

Anotações C++

Versão 7.1.0

: Frank B. Brokken
Computing Center, University of Groningen
Nettelbosje 1,
P.O. Box 11044,
9700 CA Groningen
The Netherlands

Publicado na Universidade de Groningen

ISBN 90 367 0470 7 : 1994 - 2005

Traduzido ao português do Brasil por Sergio Bacchi sob licença do autor
s.bacchi@gmail.com

Este documento está destinado aos usuários com conhecimentos da linguagem C (ou outra linguagem com gramática tipo C, como Perl ou Java) que gostariam de estender seus conhecimentos ou transitar a C++. Este documento é o livro de texto principal dos cursos de programação de Frank, organizados anualmente na Universidade de Groningen. As Anotações C++ (C++ Annotations) não cobrem todos os aspectos de C++. Serve, melhor, como um adicional a outros documentos, cobrindo as linguagens baseadas em C.

Se necessita de uma cópia das Anotações C++, estão disponíveis nos formatos postscript, pdf e outros em:

<ftp://ftp.rug.nl/contrib/frank/documents/annotations> ,

em arquivos cujos nomes começam por cplusplus. A última versão de Anotações C++ (formato html) sempre pode ser encontrado em:

<http://www.icce.rug.nl/documents/>

ÍNDICE

Capítulo 1: Visão Geral dos Capítulos.....	22
Capítulo 2: Introdução.....	25
2.1: O que há de novo nas Anotações C++.....	26
2.2: História da Linguagem de Programação C++	31
2.2.1: História das Anotações C++.....	32
2.2.2: Compilando um programa em C usando um compilador C++	33
2.2.3: Compilando um programa C++	34
2.2.3.1: C++ sob o MS-Windows.....	34
2.2.3.2: Compilando um texto fonte em C++.....	35
2.3: C++: Vantagens e aclamações.....	35
2.4: O que é Programação Orientada ao Objeto (OOP)?.....	37
2.5: Diferenças entre C e C++.....	39
2.5.1: Namespaces.....	39
2.5.2: Comentário de fim-de-linha.....	40
2.5.3: Ponteiros NULL vs. Ponteiros 0.....	40
2.5.4: Exame estrito de tipo.....	40
2.5.5: Uma nova sintaxe para os casts.....	41
2.5.5.1: O operador `static_cast'.....	42
2.5.5.2: O operador `const_cast'.....	43

2.5.5.3: O operador `reinterpret_cast`.....	43
2.5.5.4: O operador `dynamic_cast`.....	44
2.5.6: O parâmetro `void`.....	44
2.5.7: A declaração `#define __cplusplus`.....	44
2.5.8: Usando funções do padrão C.....	44
2.5.9: Arquivos cabeçalhos para ambas C e C++.....	45
2.5.10: Definindo variáveis locais.....	47
2.5.11: Sobrecarregando (Overloading) uma Função.....	50
2.5.12: Argumentos padrão de uma função.....	52
2.5.13: A palavra chave `typedef`.....	53
2.5.14: Funções como parte de uma estrutura.....	53
CAPÍTULO 3: Uma primeira impressão da C++.....	55
3.1: Mais extensões à C em C++.....	55
3.1.1: O operador resolução do escopo `::`.....	55
3.1.2: `cout`, `cin` e `cerr`.....	56
3.1.3: A palavra chave `const`.....	58
3.1.4: referências.....	61
3.2: Funções como parte de estruturas.....	66
3.3: Diversos novos tipos de dados.....	67
3.3.1: O tipo de dados `bool`.....	68
3.3.2: O tipo de dados `wchar_t`.....	69

3.3.3: O tipo de dados <code>`size_t'</code>	69
3.4: Palavras chave em C++.....	70
3.5: Encobrimento de dados: <code>public</code> , <code>private</code> e <code>class</code>	71
3.6: Estruturas em C versus estruturas em C++.....	73
3.7: Namespaces.....	74
3.7.1: Definindo espaços nomeados.....	75
3.7.1.1: Declarando entidades em espaços nomeados.....	76
3.7.1.2: Um espaço nomeado fechado.....	76
3.7.2: Referenciando entidades.....	77
3.7.2.1: A diretiva <code>`using'</code>	78
3.7.2.2: O <code>`Koenig lookup'</code>	78
3.7.3: O espaço nomeado padrão.....	81
3.7.4: Aninhando espaços nomeados e apelidos de espaços nomeados.....	82
3.7.4.1: Definindo entidades fora de seus espaços nomeados.....	84
Capítulo 4: O tipo de dados <code>`string'</code>	86
4.1: Operações com strings.....	87
4.2: Visão Geral das Operações com Cadeias de Caracteres.....	99
4.2.1: Iniciadores.....	100
4.2.2: Iteradores.....	101
4.2.3: Operadores.....	101
4.2.4: Funções Membro.....	102

Capítulo 5: A Biblioteca IO-stream.....	111
5.1: Arquivos Cabeçalho Especiais.....	114
5.2: O fundamento: a classe <code>`ios_base'</code>	115
5.3: Interfaceando objetos <code>`streambuf'</code> : a classe <code>`ios'</code>	116
5.3.1: Condição de estados.....	117
5.3.2: Formatando entradas e saídas.....	121
5.3.2.1: As flags de formatação.....	121
5.3.2.2: Funções membro modificadoras do formato.....	124
5.4: Saída.....	126
5.4.1: Saída Básica: A classe <code>`ostream'</code>	126
5.4.1.1: Escritura nos objetos <code>`ostream'</code>	127
5.4.1.2: Posicionamento numa <code>`ostream'</code>	128
5.4.1.3: Evacuação das <code>`ostream'</code>	129
5.4.2: Saída para arquivos: A classe <code>`ofstream'</code>	129
5.4.2.1: Modos para abrir objetos stream.....	131
5.4.3: Saída para a memória: A classe <code>`ostream'</code>	132
5.5: Entrada.....	133
5.5.1: Entrada Básica: A classe <code>`istream'</code>	134
5.5.1.1: Lendo de objetos <code>`istream'</code>	134
5.5.1.2: Posicionamento com <code>`istream'</code>	137
5.5.2: Entrada de streams: A classe <code>`ifstream'</code>	137

5.5.3: Entrada da memória: A classe <code>'stringstream'</code>	139
5.6: Manipuladores.....	140
5.7: A classe <code>'streambuf'</code>	143
5.7.1: Membros Protegidos de <code>'streambuf'</code>	145
5.7.2: A classe <code>'filebuf'</code>	150
5.8: Tópicos Avançados.....	150
5.8.1: Copiando streams.....	150
5.8.2: Acoplando streams.....	152
5.8.3: Redirecionando streams.....	153
5.8.4: Lendo e Escrevendo em streams.....	155
Capítulo 6: Classes.....	165
6.1: O construtor.....	167
6.1.1: Um Primeiro Aplicativo.....	169
6.1.2: Construtores: com e sem argumentos.....	172
6.1.2.1: A ordem da construção.....	175
6.2: Funções membro constantes e objetos constantes.....	176
6.2.1: Objetos anônimos	178
6.2.1.1: Subtítulos com objetos anônimos	180
6.3: A palavra chave <code>'inline'</code>	182
6.3.1: Funções <code>'inline'</code> dentro das interfaces de classe.....	183
6.3.2: Funções <code>'inline'</code> fora das interfaces de classe.....	183

6.3.3: Quando usar funções 'inline'	184
6.4: Objetos dentro de objetos: composição.....	185
6.4.1: Composição e objetos constantes: iniciadores de membros compostos.....	186
6.4.2: Objetos compostos e de referência: iniciadores de membros de referência.....	188
6.5: Organização do Arquivo Cabeçalho.....	190
6.5.1: Usando espaços nomeados nos arquivos cabeçalho.....	195
6.6: A palavra chave 'mutable'	197
Capítulo 7: Classes e alocação de memória.....	199
7.1: Os operadores 'new' e 'delete'.....	200
7.1.1: Alocação de conjuntos.....	201
7.1.2: Eliminação de conjuntos.....	202
7.1.3: Aumentando conjuntos.....	203
7.2: O Destrutor.....	204
7.2.1: 'New' e 'delete' e apontadores a objetos.....	207
7.2.2: A função set_new_handler().....	211
7.3: O operador adjudicação.....	212
7.3.1: Sobrecarga do operador adjudicação.....	214
7.3.1.1: O 'operator=()' membro.....	216
7.4: O ponteiro 'this'.....	218
7.4.1: Evitando a auto-destruição com o uso de 'this'.....	218
7.4.2: Associatividade de operadores e 'this'.....	220

7.5: A construção de cópia: iniciação versus adjudicação.....	221
7.5.1: Similaridades entre o construtor de cópias e o operator=()......	225
7.5.2: Evitando o uso de certos membros.....	227
7.6: Conclusões.....	228
Capítulo 8: Exceções.....	229
8.1: Usando as exceções: elementos de sua sintaxe.....	230
8.2: Um exemplo usando exceções.....	230
8.2.1: Anacronismos: <code>`setjmp()'</code> e <code>`longjmp()'</code>	232
8.2.2: Exceções: A alternativa preferida.....	235
8.3: Lançando exceções.....	237
8.3.1: O comando <code>'throw'</code> vazio.....	241
8.4: O bloco <code>'try'</code>	243
8.5: Apanhando exceções.....	243
8.5.1: O Receptor Padrão.....	246
8.6: Declarando Receptores de Exceções.....	247
8.7: Iostreams e Exceções.....	249
8.8: Exceções em construtores e destrutores.....	250
8.9: Os blocos <code>'try'</code> das Funções.....	256
8.10: Exceções Estandartes.....	258
Capítulo 9: Mais Sobrecarga a Operadores.....	260
9.1: Sobrecarga do <code>`operator[]()'</code>	260

9.2: Sobrecarga dos Operadores Inserção e Extração.....	264
9.3: Conversão de Operadores.....	266
9.4: A palavra chave `explicit'.....	270
9.5: Sobrecarga dos operadores incrementar e decrementar.....	273
9.6: Sobrecarga de operadores binários.....	275
9.7: Sobrecarga do 'operador new(size_t)'.....	280
9.8: Sobrecarga do `operador delete(void *)'.....	283
9.9: Os Operadores `new[]' e `delete[]'.....	284
9.9.1: Sobrecarga de 'new[]'.....	285
9.9.2: Sobrecarga de `delete[]'.....	286
9.9.2.1: `delete[](void *)'.....	286
9.9.2.2: `delete[](void *, size_t)'.....	287
9.9.2.3: Formas alternativas de sobrecarga do operador `delete[]'.....	287
9.10: Objetos Funções.....	288
9.10.1: Construção de manipuladores.....	291
9.10.1.1: Manipuladores com argumentos.....	293
9.11: Operadores Sobrecarregáveis.....	294
Capítulo 10: Funções e Dados Estáticos.....	296
10.1: Dados Estáticos.....	296
10.1.1: Dados Privados estáticos.....	297
10.1.2: Dados Públicos estáticos.....	299

10.1.3: Iniciando Dados Estáticos Constantes.....	299
10.2: Funções Membro Estáticas.....	301
10.2.1: Convenções de Chamada.....	303
Capítulo 11: Amigos (Friends).....	305
11.1: Funções 'Friend'.....	306
11.2: 'Inline friends'.....	308
Capítulo 12: Recipientes Abstratos (Containers).....	311
12.1: Notações usadas neste capítulo.....	313
12.2: O recipiente 'pair'.....	314
12.3: Recipientes Sequenciais.....	315
12.3.1: O Recipiente 'vector'.....	315
12.3.2: O recipiente 'list'.....	318
12.3.3: O recipiente 'queue'(fila).....	327
12.3.4: O Recipiente 'priority_queue'.....	329
12.3.5: O recipiente 'deque'.....	332
12.3.6: O recipiente 'map'.....	334
12.3.7: O Recipiente 'multimap'.....	344
12.3.8: O Recipiente 'set'.....	346
12.3.9: O Recipiente 'multiset'.....	349
12.3.10: O Recipiente 'stack'.....	352
12.3.11: O Recipiente 'hash_map' e outros Recipientes Aleatórios.....	354

12.4: O Recipiente `complex`.....	363
Capítulo 13: Herança.....	367
13.1: Tipos Relativos.....	368
13.2: O construtor de uma classe derivada.....	371
13.3: O destrutor de uma classe derivada.....	372
13.4: Redefinindo funções membro.....	373
13.5: Herança Múltipla.....	375
13.6: Derivação pública, protegida e privada.....	379
13.7: Conversões entre classes de base e classes derivadas.....	380
13.7.1: Conversões na adjudicação de objetos.....	380
13.7.2: Conversões na adjudicação de ponteiros.....	381
Capítulo 14: Polimorfismo.....	383
14.1: Funções Virtuais.....	383
14.2: Destrutores Virtuais.....	385
14.3: Funções Virtuais Puras.....	386
14.3.1: Implementação de Funções Virtuais Puras.....	388
14.4: Funções Virtuais em Herança Múltipla.....	389
14.4.1: Ambigüidade em Herança Múltipla.....	390
14.4.2: Classes de Base Virtuais (Derivação Virtual).....	392
14.4.3: Quando a derivação virtual não é apropriada.....	395
14.5: Identificação de Tipo em Tempo de Execução.....	396

14.5.1: O operador “dynamic_cast”.....	397
14.5.2: O Operador `typeid`.....	399
14.6: Derivando classes de `streambuf`.....	401
14.7: Uma classe Polimórfica por Exceção.....	406
14.8: Como o Polimorfismo é implantado.....	408
14.9: Referência a 'vtable' Indefinida.....	410
14.10: Construtores Virtuais.....	411
Capítulo 15: Classes com membros apontadores.....	416
15.1: Ponteiros a Membros: Um Exemplo.....	416
15.2: Definindo ponteiros a membros.....	417
15.3: Usando ponteiros a membros.....	419
15.4: Ponteiros a membros estáticos.....	422
15.5: Tamanhos do Ponteiro.....	423
Capítulo 16: Classes Aninhadas.....	426
16.1: Definindo membros de classes aninhadas.....	428
16.2: Declarando classes aninhadas.....	429
16.3: Acessando membros privados em classes aninhadas.....	430
16.4: Aninhando Enumerações.....	434
16.4.1: Enumerações Vazias.....	435
16.5: Revendo construtores virtuais	436
Capítulo 17: A Biblioteca de Modelos Padrão: Algoritmos Genéricos.....	439

17.1: Objetos Função Predefinidos.....	440
17.1.1: Objetos Funções Aritméticos.....	441
17.1.2: Objetos Funções Relacionais	446
17.1.3: Objetos Função Lógicos.....	447
17.1.4: Adaptadores de Funções.....	448
17.2: Iteradores.....	452
17.2.1: Iteradores de Inserção.....	455
17.2.2: Iteradores para objetos 'istream'.....	457
17.2.3: Iteradores para objetos 'istreambuf'.....	458
17.2.4: Iteradores para objetos 'ostream'.....	459
17.2.4.1: Iteradores para objetos 'ostreambuf'.....	459
17.3: A Classe 'auto_ptr'.....	460
17.3.1: Definindo variáveis 'auto_ptr'.....	461
17.3.2: Apontando para um objeto recém alocado.....	462
17.3.3: Apontando para outro 'auto_ptr'.....	463
17.3.4: Criando um 'auto_ptr' pleno.....	464
17.3.5: Operadores e membros.....	465
17.3.6: Construtores e membros de dados ponteiros.....	465
17.4: O Algoritmo Genérico.....	467
17.4.1: accumulate().....	470
17.4.2: adjacent_difference().....	471

17.4.3: adjacent_find()	472
17.4.4: binary_search()	474
17.4.5: copy()	475
17.4.6: copy_backward()	476
17.4.7: count()	478
17.4.8: count_if()	478
17.4.9: equal()	479
17.4.10: equal_range()	481
17.4.11: fill()	483
17.4.12: fill_n()	484
17.4.13: find()	485
17.4.14: find_end()	486
17.4.15: find_first_of()	488
17.4.16: find_if()	490
17.4.17: for_each()	491
17.4.18: generate()	494
17.4.19: generate_n()	495
17.4.20: includes()	496
17.4.21: inner_product()	498
17.4.22: inplace_merge()	501
17.4.23: iter_swap()	502

17.4.24: <code>lexicographical_compare()</code>	503
17.4.25: <code>lower_bound()</code>	506
17.4.26: <code>max()</code>	507
17.4.27: <code>max_element()</code>	509
17.4.28: <code>merge()</code>	510
17.4.29: <code>min()</code>	512
17.4.30: <code>min_element()</code>	513
17.4.31: <code>mismatch()</code>	514
17.4.32: <code>next_permutation()</code>	516
17.4.33: <code>nth_element()</code>	518
17.4.34: <code>partial_sort()</code>	519
17.4.35: <code>partial_sort_copy()</code>	520
17.4.36: <code>partial_sum()</code>	522
17.4.37: <code>partition()</code>	523
17.4.38: <code>prev_permutation()</code>	524
17.4.39: <code>random_shuffle()</code>	526
17.4.40: <code>remove()</code>	528
17.4.41: <code>remove_copy()</code>	530
17.4.42: <code>remove_if()</code>	531
17.4.43: <code>remove_copy_if()</code>	532
17.4.44: <code>replace()</code>	533

17.4.45: <code>replace_copy()</code>	534
17.4.46: <code>replace_if()</code>	536
17.4.47: <code>replace_if()</code>	537
17.4.48: <code>reverse()</code>	538
17.4.49: <code>reverse_copy()</code>	539
17.4.50: <code>rotate()</code>	539
17.4.51: <code>rotate_copy()</code>	540
17.4.52: <code>search()</code>	541
17.4.53: <code>search_n()</code>	543
17.4.54: <code>set_difference()</code>	545
17.4.55: <code>set_intersection()</code>	546
17.4.56: <code>set_symmetric_difference()</code>	548
17.4.57: <code>set_union()</code>	550
17.4.58: <code>sort()</code>	551
17.4.59: <code>stable_partition()</code>	552
17.4.60: <code>stable_sort()</code>	554
17.4.61: <code>swap()</code>	557
17.4.62: <code>swap_ranges()</code>	558
17.4.63: <code>transform()</code>	559
17.4.64: <code>unique()</code>	561
17.4.65: <code>unique_copy()</code>	563

17.4.66: upper_bound().....	564
17.4.67: Algoritmos das Pilhas.....	566
17.4.67.1: A função 'make_heap()'.....	567
17.4.67.2: A função 'pop_heap()'.....	568
17.4.67.3: The 'push_heap()' function.....	568
17.4.67.4: A função 'sort_heap()'.....	569
17.4.67.5: Um exemplo usando as funções de pilha.....	570
Capítulo 18: Modelos de Funções.....	572
18.1: Definição de modelo de funções.....	573
18.2: Dedução de Argumentos.....	578
18.2.1: Transformações em 'lvalue'.....	580
18.2.2: Conversões de Qualificação.....	581
18.2.3: Conversão a uma classe de base.....	582
18.2.4: O algoritmos de dedução de modelos de argumentos.....	583
18.3: Declarando modelos de funções.....	584
18.3.1: Instanciação de declarações.....	585
18.4: Instanciando modelos de funções.....	587
18.5: Usando modelos explícitos de tipos.....	590
18.6: Sobrecarregando modelos de funções.....	591
18.7: Especializando os modelos para tipos com desviação.....	595
18.8: O mecanismo de seleção de modelos de funções.....	598

18.9: Compilando definições de modelos e instâncias.....	600
18.10: Sumário da sintaxe da declaração do modelo.....	601
Capítulo 19: Modelos de classes.....	602
19.1: Definindo modelos de classes.....	602
19.1.1: Parâmetros padrão dos modelos de classe.....	609
19.1.2: Declarando modelos de classes.....	610
19.1.3: Distingüindo membros e tipos formais de tipos de classes.....	611
19.1.4: Parâmetros sem tipificação.....	613
19.2: Modelos de Membros.....	616
19.3: Membros de dados estáticos.....	619
19.4: Especializando modelos de classes para desviação de tipos.....	620
19.5: Especializações Parciais.....	624
19.6: Instanciando modelos de classes.....	631
19.7: Processando modelos de classes e instâncias.....	633
19.8: Declarando friends.....	634
19.8.1: Funções sem modelo ou classes como amigas.....	635
19.8.2: Modelos instanciados para tipos específicos como amigos.....	637
19.8.3: Modelos como amigos sem limite.....	640
19.9: Derivação de modelos de classe.....	643
19.9.1: Derivando classes concretas de modelos de classes.....	645
19.9.2: Derivando modelos de classes de modelos de classes.....	646

19.9.3: Derivando modelos de classes de classes concretas.....	649
19.9.4: Resolução de tipo em membros de classe de base	655
19.10: Modelos de classes e aninhamento.....	657
19.10.1: Retorno de tipos aninhados em modelos de classes.....	659
19.11: Construindo iteradores.....	661
19.11.1: Implantando um 'RandomAccessIterator'.....	663
19.11.2: Implantando um 'iterador reverso'.....	668
Capítulo 20: Aplicações de Modelos Avançados.....	671
20.1: Subtleties (sutilezas).....	672
20.1.1: A palavra chave 'typename'.....	672
20.1.2: Retornando tipos aninhados sob modelos de classes.....	675
20.1.3: Resolução de tipos em membros de classes básicas.....	677
Capítulo 21: Exemplos Concretos em C++.....	731
21.1: Usando descritores de arquivos com as classes 'streambuf'.....	731
21.1.1: Classes para operações de saída.....	731
21.1.2: Classes para operações de entrada.....	736
21.1.2.1: Usando um bufer de um caracter.....	736
21.1.2.2: Usando um bufer de n caracteres.....	738
21.1.2.3: Deslocando posições em objetos 'streambuf'.....	741
21.1.2.4: Chamadas múltiplas a 'unget()' em objetos 'streambuf'	743
21.2: Extração de campos de tamanho fixo de objetos 'istream'.....	748

21.3: O sistema de chamadas 'fork()'	753
21.3.1: Re-direção re-visitada	757
21.3.2: O programa 'Daemon'	758
21.3.3: A classe 'Pipe'	759
21.3.4: A classe 'ParentSlurp'	762
21.3.5: Comunicando-se com múltiplos filhos	764
21.3.5.1: A classe 'Select'	764
21.3.5.2: A classe 'Monitor'	769
21.3.5.3: A classe 'Child'	776
21.4: Objetos funções que realizam operações sobre bits	778
21.5: Implantando um 'reverse_iterator'	780
21.6: Um conversor para converter textos em qualquer coisa	782
21.7: Envoltórios para os algoritmos STL	784
21.7.1: Estruturas do contexto local	787
21.7.2: Funções membro chamadas por funções objetos	788
21.7.3: Modelo de função objeto com um argumento configurável	788
21.7.4: Modelo de função objeto com dois argumentos configurável	794
21.8: Usando 'bisonc++' e 'flex'	796
21.8.1: Uso do 'flex' para criar um 'scanner'	797
21.8.1.1: A classe derivada 'Scanner'	798
21.8.1.2: Implantando a classe 'Scanner'	803

21.8.1.3: Usando um objeto 'Scanner'.....	807
21.8.1.4: Construindo o programa.....	808
21.8.2: Usando ambos 'bisonc++' e 'flex'.....	809
21.8.2.1: O arquivo de especificação do 'bisonc++'.....	810
21.8.2.2: O arquivo de especificação do 'flex'.....	819
21.8.2.3: Gerando o código.....	820

Capítulo 1: Visão Geral dos Capítulos

Os capítulos das Anotações C++ cobrem os seguintes tópicos:

- Capítulo 1: Esta visão geral dos capítulos.
- Capítulo 2: Uma introdução geral à C++.
- Capítulo 3: Uma primeira impressão: diferenças entre C e C++.
- Capítulo 4: O tipo de dados 'string'.
- Capítulo 5: A biblioteca C++ de E/S .
- Capítulo 6: O conceito de 'classe': estruturas com funções. O conceito 'object': variáveis de uma classe.
- Capítulo 7: Alocação e devolução de memória não usada: novo, elimine, e a função 'set_new_handler'.
- Capítulo 8: Exceções: manipulação de erros onde apropriado, melhor onde ocorrem.
- Capítulo 9: Dê seu próprio significado aos operadores.
- Capítulo 10: Dados e funções estáticos: membros de uma classe fora das fronteiras dos objetos.
- Capítulo 11: Ganhando acesso a partes privadas: funções e classes friend.

- Capítulo 12: Recipientes Abstratos para se depositar coisas.
- Capítulo 13: Construindo classes sobre classes: criando hierarquias de classes
- Capítulo 14: Mudando o comportamento de funções membro acessadas através de apontadores da classe básica.
- Capítulo 15: Classes com apontadores a membros: apontando a locais dentro de objetos.
- Capítulo 16: Construindo classes e enumeradores em classes.

- Capítulo 17: A Biblioteca Padrão de Modelos, algoritmos genéricos .
- Capítulo 18: Funções Modelo: usando moldes para funções de tipo independente.
- Capítulo 19: Classes Modelo: usando moldes para classes de tipo independente.
- Capítulo 20: Aplicações avançadas de modelos
- Capítulo 21: Diversos exemplos de programas escritos em C++.

Capítulo 2: Introdução

Este documento oferece uma introdução à linguagem de programação C++. é um guia para cursos de programação C/C++, apresentado anualmente por Frank na Universidade Groningen. Este documento não é um manual C/C++ completo, já que muito do ambiente C presente na C++ não é coberto. Para isto deve-se referir a outras fontes (p.ex., the Dutch book De programmeertaal C, Brokken and Kubat, University of Groningen, 1996).

Aqui assume-se que o leitor tenha esse extensivo conhecimento da linguagem de programação C. As Anotações C++ continua de onde os tópicos da linguagem de programação C termina, tais como controle básico de fluxo de ponteiros e a construção de funções.

A versão das Anotações C++ (atualmente em 6.2.3) é atualizada quando o conteúdo do documento muda. O primeiro número é o principal, e provavelmente não mudará por um tempo: indica modificações maiores. O número do meio é incrementado quando se adiciona informações ao documento. O último número indica pequenas mudanças; é incrementado quando, por exemplo, são corrigidos erros de digitação.

Este documento é publicado pelo Centro de Computação da Universidade de Groningen, Holanda. Foi digitado no formato yodl.

Todos os direitos reservados. Nenhuma parte deste documento pode ser publicada ou modificada sem o consentimento prévio do autor. Toda correspondência concernente a sugestões, adições, melhorias ou modificações a este documento devem ser dirigidas ao autor:

Frank B. Brokken

Computing Center, University of Groningen
Nettelbosje 1,
P.O. Box 11044,
9700 CA Groningen

(email: f.b.broken@rc.rug.nl)

Neste capítulo uma primeira impressão da C++ é apresentada. Algumas poucas extensões à C são apresentadas. Algumas poucas extensões são revistas e os conceitos de baseado em objeto e programação orientada ao objeto (OOP) são brevemente introduzidos.

2.1: O que há de novo nas Anotações C++

Esta seção é modificada quando quando a primeira ou segunda parte do número de versão muda (e algumas vezes com a terceira também).

- A versão 7.1.0 agrega a descrição do membro `type_info::before()` (cf. seção 14.5.2). Ainda mais, diversas correções tipográficas foram feitas.
- A versão 7.0.1. foi uma distribuição rápida depois da distribuição da versão 7.0.0, como resultado de um retorno muito extenso recebido de Eric S. Raymond (`esr at thyrsus dot com`) e Edward Welbourne (`eddy at chaos dot org dot uk`). Considerando a extensão do retorno recebido, é apropriado mencionar explicitamente uma sub-sub-distribuição aqui. Muitas mudanças textuais foram feitas e a seção 4.2.4 foi completamente reorganizada.
- A versão 7.0.0 vem com um capítulo novo que discute aplicações avançadas de modelos. Além disso, a terminologia geral usada com moldes evoluiu. 'Template' é considerado agora um conceito-nuclear, que se reflete pelo uso de 'template' como um substantivo, antes que um adjetivo. Assim, de agora em diante temos a 'classe template' antes que 'uma classe de modelos'. A adição de outro capítulo, junto com a adição de diversas seções novas aos capítulos existentes assim como várias reescritas de seções já existentes fez-nos promover a liberação da edição principal seguinte. O capítulo recentemente adicionado não visa exemplos concretos de modelos. Antes discute possibilidades de modelos além dos básicos da função e da classe. Além deste capítulo novo, diversas seções novas foram adicionadas: a seção 6.5 introduz classes locais; a seção 7.1.4 discute a colocação do operador 'new'; a seção 13.6.1 discute como disponibilizar alguns membros de classes herdadas em privado e a seção 13.8 discute por como os objetos criados por 'new[]' podem ser iniciados por construtores não padrões. Além de tudo isto, Elwin Dijck (e ponto dijck arroba gmail ponto com), um dos estudantes da edição 2006-2007 do curso

de C++, fez um trabalho magnífico convertendo todas as imagens a vetores gráficos (que alertame a começar usar vetores gráficos também: -). Agradecimentos a Elwin por um trabalho bem feito!

- A versão 6.5.0 mudou o tipo sem sinal a `size_t` onde apropriado, e os tipos explicitamente mencionados derivados de `int` como `int16_t`. As definições dos membros da função da in-class foram movidas para fora (abaixo) de suas definições de classe como membros definidos inline. Foram adicionados parágrafos sobre execução das funções membro virtuais puras. Vários erros e erros de compilação foram corrigidos. Adicionou a tradução ao português à distribuição, depois de receber a aceitação de Sergio Bacchi.
- A Versão 6.4.0 teve aumentada uma nova seção (19.10.1) que discute melhor o uso da Palavra chave `template` para distinguir tipos aninhados em modelos de classes de modelos de membros. Ainda mais, Sergio Bacchi s dot bacchi at gmail dot com fez um impressionante trabalho quando traduziu as Anotações ao português. Sua tradução (que pode estar atrasada uma distribuição ou duas em relação à última versão das Anotações) Também pode ser retirada de:
<ftp://ftp.rug.nl/contrib/frank/documents/annotations>
- A versão 6.3.0 teve agregada novas seções sobre objetos anônimos (seção 6.2.1) e resolução de tipo em modelos de classes (seção 19.9.4). Também a descrição do algoritmo de dedução do modelo de parâmetro foi reescrito (seção 18.2.4) e numerosas modificações requeridas devido que o compilador foi sintonizado melhor ao estandarte C++, entre as quais relançamento de exceções de blocos `try` de funções construtoras e destrutoras. Também, todas as correções dos textos recebidas dos leitores desde a versão 6.2.4 foram processadas.
- Na versão 6.2.4 foram feitas muitas melhorias no texto. Recebi extensas listas de erros de digitação e sugestões para clarificar o texto, em particular de Nathan Johnson e Jakob van Bethlehem. Igualmente valiosas foram as sugestões recebidas de vários outros leitores das **anotações C++**: todas foram processadas nesta distribuição. O conteúdo da matéria da **C++**, desta distribuição não foi substancialmente modificado, comparado com a versão 6.2.2.
- Versão 6.2.2 oferece uma implementação melhorada das classes modelo configuráveis (seções 20.7.3 e 20.7.4).
- Versão 6.2.0 foi liberada como uma Atualização Anual, ao fim de Maio de 2005. Suas características, além da usual correção de erros de digitação se acrescentaram diversas seções novas e algumas foram removidas: no capítulo das Exceções (8) uma seção foi acrescentada cobrindo as exceções padrão e seus significados; no capítulo que cobre membros estáticos (10) uma seção foi acrescentada que discute membros de dados estáticos constantes; e o capítulo final cobre classes modelo configuráveis usando estruturas do contexto local (substituindo as classes

anteriores `ForEach`, `UnaryPredicate` e `BinaryPredicate`). Ainda mais, a seção final (cobrindo um gerador de parser (analisador) C++) agora usa o programa `bison++`.

- Versão 6.1.0 liberada pouco depois da 6.0.0. Seguindo as sugestões de Leo Razoumov <LEOR@winmain.nutgers.edu> e Paulo Tribolet, e depois de recebidas inúmeras sugestões úteis e ajuda extensiva de Leo, os arquivos .pdf navegáveis desde agora são distribuídos com as Anotações C++. Algumas seções também foram ligeiramente adaptadas.
- Versão 6.0.0 liberada depois de uma atualização completa do texto, removendo muitas inconsistências e erros de digitação. Parece mesmo apropriado dizer que a atualização afetou o texto completo das Anotações C++, fazendo jus, parece, a uma versão principal. Diversas novas seções foram acrescentadas: sobrecarregando operadores binários (seção 9.6); lançamento de exceções em construtores e destrutores (seção 8.8); função try-blocos (seção 8.9); convenções de chamada de funções estáticas e globais (seção 10.2.1) e construtores virtuais (seção 14.10). O capítulo sobre modelos foi completamente reescrito e dividido em dois capítulos separados : o capítulo 18 discute a sintaxe e uso de funções modelo; o capítulo 19 discute classes modelo. Vários exemplos concretos foram modificados; novos exemplos foram incluídos também (Capítulo 20).
- Na versão 5.2.4 a descrição do algoritmo genérico `random_shuffle` (seção 17.4.39) foi modificada.
- Na versão 5.2.3 a seção 2.5.10 sobre variáveis locais foi estendida e a seção 2.5.11 respeito à sobrecarga de funções foi modificada explicitando a discussão dos efeitos do modificador `const` com funções sobrecarregadas. A descrição da função `compare()` no capítulo 4 continha um erro que também foi reparado.
- Na versão 5.2.2 um resto, na seção 9.4, da versão anterior foi removido e o texto correspondente foi atualizado. Alguns erros de digitação também foram corrigidos.
- Na versão 5.2.1 vários erros de digitação foram reparados e alguns parágrafos foram reescritos clarificando-os. Além disso uma seção foi adicionada ao capítulo 18 sobre modelos, sobre a criação de diversos tipos de iteradores. Este tópico foi posteriormente elaborado no capítulo 20, onde a seção sobre a construção de um iterador reverso (seção 20.5) foi completamente reescrita. No mesmo capítulo, um texto sobre um conversor a nada é discutido (seção 20.6). Também desde então estão disponíveis versões em LaTeX, PostScript e PDF para o formato de papel carta, bem como versões: `cplusplus.latex`, `cplusplus.ps` e `cplusplus.pdf`. O formato A4, sem dúvida, foi mantido e continua disponível como arquivos `cplusplus.latex`, `cplusplus.ps` e `cplusplus.pdf`.
- A versão 5.2.0 foi liberada depois da adição de uma seção sobre a palavra chave mutável (seção 6.6) e depois de uma meticulosa mudança na discussão da classe abstrata `Fork()` (seção 20.3).

Todos os exemplos podem agora serem atualizados com respeito ao uso do espaço nomeado `std`.

- Nesse meio tempo, contudo, o compilador Gnu g++ versão 3.2 foi liberado\ (<http://www.gnu.org>). Nesta versão foram incluídas extensões aos recipientes abstratos (veja a seção 12.3.11) num espaço nomeado separado, `__gnu_cxx`. Este espaço nomeado pode ser usado junto com aqueles recipientes. Mas isto pode parar compilações de fontes com g++ versão 3.0. Nesse caso uma compilação pode ser executada condicionalmente nas versões do compilador 3.2 e 3.0, definindo `__gnu_cxx` para a versão 3.2. Alternativamente, o truque sujo `#define __gnu_cxx std` pode ser usado imediatamente antes do arquivo cabeçalho onde o espaço nomeado é usado. Isto pode resultar numa eventual colisão de nomes e é um truque sujo segundo qualquer padrão, portanto não conte a ninguém que eu escrevi isto.
- A versão 5.1.1 foi liberada depois das modificações nas seções relativas à chamada ao sistema `fork()` no capítulo 20. Sob o padrão ANSI/ISO muitas das extensões previamente disponíveis (como `procbuf` e `vform()`) aplicadas a streams foram descontinuadas. A partir da versão 5.1.1 as formas de construir estas facilidades sob o padrão ANSI/ISO são discutidas nas Anotações C++. Considero o objetivo suficientemente complexo para garantir uma nova sub-versão.
- Com o aparecimento do compilador Gnu g++ versão 3.00 uma implementação mais estrita do padrão ANSI/ISO C++ se torna possível. Isto resultou na versão 5.1.0 das Anotações, aparecida pouco depois da versão 5.0.0. Na versão 5.1.0 o capítulo 5 foi modificado e outras mudanças cosméticas tiveram lugar (p.ex. removendo classes das listas de tipos de parâmetros, veja o capítulo 18). As versões intermediárias (como 5.0.0a, 5.0.0b) não foram mais publicadas, pois eram apenas versões intermediárias aguardando a versão 5.1.0. Gradualmente os exemplos serão adaptados ao novo compilador. Entretanto o leitor deve estar preparado para inserir usando o espaço nomeado `std`; em muitos exemplos, justo depois do processador de diretivas `#include`, como uma medida temporária para o compilador aceitar o exemplo.
- Novos desenvolvimentos aparecem todo o tempo, resultando na versão 5.0.0 das Anotações. Nesta versão se limpou muito código e erros de digitação foram corrigidos. De acordo ao padrão atual, se requerem espaços nomeados nos programas C++, assim foram introduzidos muito cedo (na seção 2.5.1) nas Anotações. Uma nova seção sobre o uso de programas externos foi acrescentada às Anotações (e removida na versão 5.1.0), uma nova classe `stringstream`, substituindo a classe `strstream` agora é coberta também (seções 5.4.3 e 5.5.3). Atualmente o capítulo sobre entradas e saídas foi completamente reescrito. Ainda mais, os operadores `new` e `delete` são discutidos no capítulo 7, onde fica melhor que um capítulo sobre classes que anteriormente eram discutidas aí.
- Alguns capítulos foram desmembrados e reorganizados, assim os assuntos podem geralmente ser introduzidos sem referências posteriores. Finalmente as Anotações C++ dispõem agora versões em

formato html, PostScript e pdf com índice (soa importante?). Considerando o volume e a natureza das modificações parece justo atualizar a uma versão principal maior. Assim que aqui está. Considerando o volume das Anotações, estou seguro que haverá erros de digitação sempre, agora e sempre. Por favor, não exite em mandar um mail declarando qualquer erro encontrado ou correções que gostaria de sugerir.

- As mailing lists das Anotações pararam na versão 4.4.1d. A partir deste ponto somente modificações menores eram esperadas, que geralmente não são anunciadas.
- Na versão 4.4.1b o tamanho da página no arquivo LaTeX foi definido como din A4. Nos países onde outro padrão de tamanho de página é usado a melhor escolha é uma conversão. Para o fim, remova a opção din A4 do cplusplus.tex (ou cplusplus.yo se usa o yodl) e reconstrua as Anotações com o arquivo Tex ou arquivos Yodl.
- Em algum ponto considerei que a versão 4.4.1 seria a versão final das Anotações C++. Contudo, uma seção especial com funções de entrada e saída foi acrescentada para cobrir entradas e saídas não formatadas e a seção sobre tipo de dados string foi melhorada e devido ao seu volume, lhe dei um capítulo (capítulo 4). Tudo isto resultou eventualmente na versão 4.4.2.
- A versão 4.4.1 novamente contém novo material e reflete o padrão ANSI/ISO (bem, intentei refletir o padrão ANSI/ISO). Na versão 4.4.1 foram acrescentados vários capítulos e seções, entre os quais um capítulo sobre a Biblioteca de Modelos Padrão (Standard Template Library STL) e algoritmos genéricos.
- A versão 4.4.0 (e sub-letas) era apenas uma versão em construção e nunca ficou disponível.
- A versão 4.3.1a é a precursora da 4.3.2. Na 4.3.1a a maior parte dos erros de digitação que recebi desde a última atualização foi processada. Na versão 4.3.2 foi posta atenção especial à sintaxe nos endereços das funções e apontadores a membros das funções.
- A decisão de atualização da versão 4.2.* à 4.3.* foi tomada depois de realizar o escaneador léxico a função yylex() pode ser definida na classe scanner que é derivada de yyFlexLexer. Sob esta visão a função yylex() pode acessar os membros das classes derivadas de yyFlexLexer e os membros de yyFlexLexer, tanto os públicos como os protegidos. O resultado disto é uma implementação limpa das regras definidas na especificação do arquivo flex++.
- A atualização da versão 4.1.* para 4.2.* foi o resultado da inclusão da seção 3.3.1 sobre tipos de dados booleanos no capítulo 3. A distinção das diferenças entre C e C++ e extensões da linguagem de programação C estão (embora um pouco vago) refletidas no capítulo de introdução e no

capítulo das primeiras impressões da C++: O capítulo de introdução abrange certas diferenças entre C e C++, já o capítulo sobre as primeiras impressões da C++ abrange algumas extensões da linguagem de programação C, como encontradas em C++.

- A versão 4 representa uma reescritura da versão anterior 3.4.14: o documento foi reescrito de SGML a Yodl e muitas seções novas foram adicionadas. Todas as seções foram melhoradas. A base da distribuição, contudo, não mudou: veja a introdução.

As modificações nas versões 1.*.*, 2.*.* e 3.*.* (substitua as estrelinhas por qualquer número aplicável) não foram documentadas.

- As sub-versões como 4.4.2a, etc. contém correções de erros e correções tipográficas.

2.2: História da Linguagem de Programação C++

A primeira implementação de C++ foi desenvolvida nos anos oitenta do século XX nos Laboratórios da AT&T Bell, onde foi criado o Sistema Operacional Unix.

A linguagem C++ era originalmente um pré-compilador, similar ao pré-processador da linguagem C, que converte construções especiais em seu pleno código C. Este código era então compilado por um compilador C normal. O pré-código, lido pelo pré-compilador C++, usualmente estava em arquivos com extensão .cc, .C ou cpp. Estes arquivos eram então convertidos em fontes C e postos em arquivos com extensão .c, que eram, então, compilados e ligados.

A nomenclatura dos arquivos fonte C++ permanece: as extensões .cc e .cpp são ainda usadas. Contudo, a elaboração preliminar de um pré-compilador C++ nos compiladores modernos usualmente está incluída no processo de compilação. Frequentemente os compiladores determinam o tipo de fonte pela extensão do arquivo. Isto é verdade para os compiladores C++ da Borland e Microsoft, que assumem que as fontes C++ estão em arquivos com extensão .cpp. O compilador Gnu g++, disponível em muitas arquiteturas Unix assume a extensão .cc.

O fato de que a C++ é compilada para código C é também visível do fato que C++ é um super-conjunto da C: A C++ oferece todas as possibilidades da C e mais. Isto faz a transição da C é C++ bem fácil. Os programadores familiarizados com C podem começar a programar usando os arquivos fonte

com extensão .cc ou .cpp no lugar de .c e e então, confortavelmente, introduzir-se em todas as possibilidades oferecidas pela C++. Não necessita de abruptas mudanças de costumes.

2.2.1: História das Anotações C++

A versão original das Anotações C++ foi escrita por Frank Brokken e Karel Kubat em holandês usando LaTeX. Depois de algum tempo Karel reescreveu o texto e o converteu num guia num formato mais amigável e (claro está) em inglês em Setembro de 1994.

A primeira versão do guia apareceu na rede em Outubro de 1994. Então convertido a SGML.

Gradualmente foram acrescentados novos capítulos e o conteúdo foi modificado e melhorado (graças a inumeráveis leitores que enviaram comentários).

Na transição da versão principal três à quatro realizada por Frank: outra vez se acrescentou novos capítulos e o documento fonte foi convertido de SGML a Yodl.

As Anotações C++ não são livremente distribuíveis. Assegure-se de ler as notas legais.

Continuando a leitura das Anotações além deste ponto implica concordância com as restrições impostas.

Se este documento for de seu agrado indique-o a seus amigos. Melhor ainda, faça-nos saber enviando um mail a Frank.

Na Internet existem muitos hyperlinks úteis a C++. Sem mesmo sugerir completude (e sem examinar regularmente sua existência: podem haver desaparecido no momento em que se leia isto), os seguintes merecem serem visitados:

○ <http://www.cplusplus.com/ref/> : Um lugar de referência à C++.

o (<http://www.csci.csusb.edu/dick/c++std/cd2/index.html>) :
Publica uma versão de 1996 do padrão ANSI/ISO da C++)

2.2.2: Compilando um programa em C usando um compilador C++

Em benefício da completude é imperativo mencionar que C++ é "quase" um super-conjunto de C. Existem algumas diferenças que se pode encontrar quando simplesmente se renomeia um arquivo com extensão .cc e o compilamos com um compilador C++:

- Em C `sizeof('c')` é igual a `sizeof(int)`, 'c' sendo qualquer caracter ASCII. A idéia subjacente é, provavelmente, que quando passamos como argumentos de funções, são passados de qualquer forma como inteiros. Ainda mais, o compilador C manipula um caracter constante como 'c' como um inteiro constante. Daí que, em C, as chamadas à função:

```
putchar(10);
```

e

```
putchar('\n');
```

são sinônimos.

Em contraste, em C++, `sizeof('c')` é sempre 1 (mas veja também a seção 3.3.2), enquanto um `int` ainda é um `int`. Como veremos mais tarde (veja seção 2.5.11), as duas chamadas à função:

```
umafunc(10);
```

e

```
umafunc('\n');
```

Podem ser manipuladas por funções completamente diferentes: C++ distingue as funções não só pelo

seus nomes, mas também pelo tipo de seus argumentos, que são diferentes nessas duas chamadas: uma usando um argumento inteiro, a outra usando um caracter.

- A C++ requer protótipos de funções externas muito estritos. Por exemplo, um protótipo como

```
extern void func();
```

Em C significa que uma função `func()` existe, que não retorna nada. A declaração não especifica que argumentos (se existir algum) a função toma.

Em contraste, tal declaração em C++ significa que aquela função `func()` não toma argumento algum: a passagem de argumentos a ela produz erro de compilação.

2.2.3: Compilando um programa C++

Para compilar um programa C++ é necessário um compilador C++. Considerando a natureza livre deste documento, não deve ser uma surpresa o fato de que sugerimos um compilador livre aqui. A Fundação para Software Livre (FSF - Free Software Foundation) entrega em <http://www.gnu.org> um compilador livre, entre outros parte do Debian (<http://www.debian.org>) distribuição do Linux (<http://www.linux.org>).

2.2.3.1: C++ sob o MS-Windows

Para o MS-Windows a Cygnus (<http://sources.redhat.com/cygwin>) fornece os fundamentos para instalar o compilador Gnu g++ no Windows.

Quando no local do URL acima para obter o g++ livre clique em instalar agora. Esta ação baixará o arquivo `setup.exe`, que instalará cygwin. O software a ser instalado pode ser baixado da internet. Existem alternativas (como por exemplo usando um CD) descritas na página de Cygwin. A instalação continua interativamente. Os padrões oferecidos são normalmente aquilo que se busca.

O compilador Gnu g++ pode ser obtido de:

<http://gcc.gnu.org>

Se o compilador oferecido na distribuição Cygnus não corresponde à última versão, as fontes da última versão podem ser baixadas e o compilador pode ser construído usando com o compilador da distribuição Cygnus. A página da web do compilador (mencionada acima) contém instruções detalhadas de como proceder. Em nossa experiência, a construção de um novo compilador com o ambiente Cygnus funciona fluentemente.

2.2.3.2: Compilando um texto fonte em C++

Em geral o seguinte comando é usado para compilar um arquivo fonte C++ 'fonte.cc':

```
g++ fonte.cc
```

Isto produz um programa binário (a.out ou a.exe). Se não se deseja um nome padrão, o nome do executável pode ser especificado usando-se a opção -o (aqui produzindo o programa fonte):

```
g++ -o fonte fonte.cc
```

Se é requerida uma mera compiladorção pode-se compilar o módulo usando a opção -c:

```
g++ -c fonte.cc
```

Esta opção produz o arquivo fonte.o que pode ser lincado a outros módulos posteriormente.

Usando o programa icmake um escrito de manutenção pode ser usado para assistir na construção e manutenção de programas C++. Um escrito genérico de manutenção, testado durante anos em plataformas Linux, está disponível. Sua descrição e componentes podem ser encontrados num arquivo de nome icmake-C1.61.tar.gz (ou semelhante), guardado no mesmo lugar do programa icmake. Alternativamente o programa make padrão pode ser usado para manutenção de programas C++. é altamente recomendável começar usando escritos ou programas de manutenção no estudo da linguagem de programação C++.

2.3: C++: Vantagens e aclamações

Freqüentemente é dito que programar em C++ conduz a programas 'melhores'. Algumas das vantagens de C++ proclamadas são:

- Novos programas poderiam ser desenvolvidos em menos tempo porque antigos códigos podem ser reusados.
- Criar e usar novos tipos de dados seria mais fácil que em C.

- O gerenciamento de memória sob C++ seria mais fácil e transparente.
- Os programas estariam menos propensos a erros, já que C++ usa uma sintaxe estrita e exame de tipo.
- O encobrimento de dados ('Data hiding') - o uso de dados por uma parte do programa enquanto outras partes não podem acessar os dados seria mais fácil de implementar.

Quais destas afirmações são verdadeiras? Originalmente nossa impressão era de que a linguagem C++ estava um pouco superestimada; igual que toda a programação voltada para o objeto (OOP). O entusiasmo respeito à linguagem C++ assemelha-se ao havido em relação às linguagens voltadas à Inteligência Artificial (AI) Lisp e Prolog: se supunha que através destas linguagens se resolveriam os problemas mais difíceis em AI 'quase sem esforço'. Claro que estórias muito promissoras respeito a linguagens de programação precisam ser examinadas; finalmente qualquer problema pode ser codificado em qualquer linguagem de programação (seja BASIC ou assembler). As vantagens ou desvantagens de uma dada linguagem de programação não está em 'o que se pode ou não pode com ela', mas em 'que ferramentas a linguagem oferece para se chegar a uma solução compreensível de um problema de programação'.

Sobre os elogios é C++, nós apoiamos o seguinte:

- O desenvolvimento de programas novos, quando existe código reusável, também pode ser realizado em C, usando-se as bibliotecas de funções. As funções podem ser coletadas numa biblioteca e não necessitam ser reinventadas em cada novo programa. A linguagem C++ , contudo, oferece possibilidades sintéticas específicas na reutilização de código além das bibliotecas de funções (veja o capítulo 13).
- A criação e utilização de novos tipos de dados é também possível em C; usando-se estruturas (struct) e definição de tipos (typedef), etc.. Destes tipos outros tipos podem derivar, isto leva a estruturas que contém estruturas e assim por diante. Em C++ estas facilidades estão argumentadas na definição de tipos de dados completamente 'auto suportados', cuidando do gerenciamento de sua memória automaticamente (sem recorrer a um gerenciamento da memória do sistema operacional como o usado, por exemplo, em Java).
- O gerenciamento de memória em C++, é em princípio, tão fácil ou tão difícil como em C. Especialmente quando usamos funções C dedicadas tais como xmalloc() e xrealloc() (alocando a memória ou abortando quando esta se exauriu). Contudo, com funções do tipo malloc() é fácil cair-se em erro: calculando mal o número de bytes numa chamada a malloc() freqüentemente

conduz a erro. Já em C++ as facilidades oferecidas para alocação de memória são até certo ponto seguras, através de seu operador `new`.

- No concernente à 'propensão a erro' podemos dizer que a C++ usa de fato um exame mais estrito de tipo que a C. Contudo a maioria dos compiladores modernos possuem um sistema de 'níveis de alarme'; é por escolha do programador ignorar ou não os alarmes gerados. Em C++ muitos desses alarmes se transformam em erros fatais (a compilação para).
- No que concerne ao 'encobrimento de dados', a linguagem C oferece algumas ferramentas. Pode-se usar variáveis locais ou estáticas onde é possível e tipos especiais de dados como estruturas manipuladas por funções dedicadas. Usando tais técnicas mesmo em C a defesa de dados pode ser realizada; apesar disso, deve ser admitido que a linguagem C++ oferece construções sintéticas especiais, fazendo a 'ocultação de dados' muito mais fácil de realizar que em linguagem C.

A linguagem C++ em particular (e OOP em geral) está claro que não é a solução para todos os problemas de programação. Contudo, a linguagem oferece várias facilidades novas e elegantes que vale a pena investigar. Ao mesmo tempo o nível de complexidade gramatical de C++ aumentou significativamente comparado com C. Esta pode ser considerada uma desvantagem séria da linguagem. Contudo enfrentamos este nível de complexidade e a transição não foi rápida e sem penas. Com as Anotações C++ esperamos ajudar o leitor fazer a transição de C a C++ entregando nossas anotações, que sem dúvida, se encontram em alguns livros de texto sobre C++. É nossa esperança que você goste deste documento e se beneficie com ele: Divirta-se e boa sorte em sua jornada em C++!

2.4: O que é Programação Orientada ao Objeto (OOP)?

Programação orientada ao objeto (e baseada no objeto) propõe uma maneira claramente diferente de resolver os problemas de programação que a estratégia usualmente usada em programas em C. Na programação em linguagem C os problemas normalmente são resolvidos usando-se um 'método procedural': um problema é decomposto em sub-problemas e este processo é repetido até que as sub-tarefas possam ser codificadas. Assim um conglomerado de funções é criado, comunicadas através de argumentos e variáveis, globais ou locais (ou estáticas).

Em contraste com (ou melhor: em adição a) isto a maneira baseada no objeto identifica palavras-chave no problema. Estas palavras-chave são representadas num diagrama e se desenham flechas entre essas palavras-chave para definir uma hierarquia interna. As palavras-chave serão os objetos na execussão e a hierarquia define as relações entre os objetos. O termo objeto é aqui usado para descrever uma estrutura

limitada e bem-definida, contendo toda informação sobre um ente: tipos de dados e funções para manipular os dados. Como exemplo de uma forma orientada ao objeto segue uma ilustração:

Os funcionários e o proprietário de um negócio de distribuição de carros e uma companhia de garagem de autos são pagos como segue. Primeiro, os mecânicos que trabalham na garagem recebem mensalmente. Segundo, o proprietário da companhia recebe uma soma fixa por mês. Terceiro, os vendedores de autos recebem salários mensais e um bônus por carro vendido. Finalmente, os funcionários que compram carros de segunda-mão e recorrem a região recebem salários mensais, um bônus por carro e a devolução de despesas de viagem.

Quando representamos a administração dos salários acima, as palavras-chave poderiam ser mecânicos, proprietário, vendedores e compradores. As propriedades de cada unidade são: salário mensal, às vezes bônus por compra ou venda e às vezes restituições de despesas de viagens. Ao analisarmos o problema desta maneira chegamos à seguinte representação:

- O proprietário e os mecânicos podem ser representados pelo mesmo tipo, recebem salário mensal. A informação relevante deste tipo seria a quantia mensal. Adicionalmente este objeto conteria dados, tais como: nomes, endereços, etc..
- Os vendedores de carros podem ser representados com o mesmo tipo acima mas com alguma funcionalidade extra: o número de transações (vendas) e bônus por transação.

Na hierarquia dos objetos podemos definir a dependência entre os primeiros dois objetos deixando os vendedores de carros como 'derivado' do proprietário e mecânicos.

- Finalmente estão os compradores de carros de segunda-mão. Estes compartilham a funcionalidade dos vendedores, exceto pelas despesas de viagens. A funcionalidade adicional consistirá, portanto, nas despesas feitas e este tipo pode derivar do tipo vendedor.

A hierarquia dos objetos identificados está ilustrada na Figura 1.



Figura 1 Hierarquia dos objetos na administração de salários.

O processo total na definição da hierarquia, tal como acima, começa com a descrição dos tipos mais simples. Em seguida os tipos mais complexos são derivados e cada derivação agrega um pouco de funcionalidade. Destes tipos derivados outros tipos podem ser derivados e assim por diante, até que a representação de todo o problema possa ser feita.

Em C++ cada objeto pode ser representado por uma classe que contenha a funcionalidade necessária para ser útil na elaboração das variáveis (chamadas objetos) dessas classes. Não toda a funcionalidade e não todas as propriedades de uma classe usualmente está presente nos objetos de outras classes. Como veremos, as classes tendem a esconder suas propriedades, de tal maneira que não são modificáveis diretamente pelo mundo exterior. Em vez disso, funções dedicadas são usadas para acessar e modificar as propriedades dos objetos. Também os objetos tendem a ser auto-contidos. Eles encapsulam toda a funcionalidade e dados requeridos para realizar suas tarefas e defender a integridade dos objetos.

2.5: Diferenças entre C e C++

Nesta seção mostramos alguns exemplos de código C++. Algumas diferenças entre C e C++ são ressaltadas.

2.5.1: Namespaces

A linguagem C++ introduz a noção de espaço nomeado (namespace): todos os símbolos são definidos num grande contexto, chamado espaço nomeado. O espaço nomeado é usado para evitar conflitos entre nomes que pode advir quando o programador quer definir como `sin()`, que opere restritamente mas não quer perder a capacidade de usar a função padrão `sin()`, operando em radianos.

Os espaços nomeados são cobertos extensivamente na seção 3.7. Por agora devemos notar que a maioria dos compiladores requerem a declaração explícita de um padrão namespace: `std`. Assim, a menos de indicação contrária, fica subentendido que nas Anotações todos os exemplos agora usam declaração: `namespace std`.

2.5.2: Comentário de fim-de-linha

De acordo com a definição ANSI, os 'comentários de fim-de-linha' são postos com a sintaxe da C++. Estes comentários começam com `//` e terminam no fim da linha. O padrão C para comentários, delimitados por `/*` e `*/` podem ainda serem utilizados em C++:

```
int main()
{
    // este é um comentário de fim-de linha
    // um comentário por linha

    /*
        este é um comentário padrão C, cobre
        muitas linhas.
    */
}
```

Apesar do exemplo, é recomendado não usar comentários tipo C dentro do corpo das funções. às vezes é necessário suprimir pedaços de código. Nestes casos é bastante prático usar o padrão dos comentários C. Se o código suprimido contém tais comentários, resultará em linhas de comentário aninhadas, o que conduzirá a erros de compilação. Daí a regra de não usar comentários tipo C no corpo de funções C++.

2.5.3: Ponteiros NULL vs. Ponteiros 0

Na linguagem C++ todo valor zero é codificado como 0. Na linguagem C, no concernente aos apontadores, freqüentemente o valor NULL é usado. Esta diferença é puramente estilística. Em C++ já não há necessidade de se usar NULL e o uso de 0 é preferível para indicar um valor nulo de um ponteiro.

2.5.4: Exame estrito de tipo

C++ usa uma verificação de tipo muito estrita. Um protótipo deve ser declarado para cada função antes de sua chamada, e a chamada deve coincidir com o protótipo. O programa:


```
int main()
{
    printf("Olá Mundo\n");
}
```

compila frequentemente sob C, apesar de que com um aviso que o `printf()` não é uma função conhecida. Muitos compiladores da C++ não produzirão código em tal situação. O erro está na falta, naturalmente, de `#include <stdio.h>` a diretriz orientadora.

Embora que nela: em C++ a função `main()` usa sempre o valor de retorno `int`. é possível definir `int main()`, sem uma indicação de retorno explícita, mas uma indicação de retorno sem uma expressão não pode existir na função `main()`: uma indicação de retorno no `main()` deve sempre ser dada como uma expressão `int`. Para o exemplo:

```
int main()
{
    return;        // não compilará: espera uma expressão int
}
```

2.5.5: Uma nova sintaxe para os casts

Tradicionalmente, a C oferece o seguinte modelo de construção de casts:

(nome-do-tipo)expressão - onde nome-do-tipo é o nome de um tipo válido, e expressão uma expressão. Aparte do estilo de cast C (depreciado agora) C++ suporta também casts com notação de chamada de função:

nome-do-tipo(expressão) Esta notação de chamada de função não é atualmente um modelo, mas o pedido ao compilador para construir uma variável (anônima) do tipo nome-do-tipo da expressão expressão. Esta forma é atualmente usada muito frequentemente em C++, mas não deve ser usada para os casts. Em seu lugar quatro novos modelos foram introduzidos:

- O cast estandarte para converter um tipo em outro é:

```
static_cast<tipo>(expressão)
```

- Existe um cast especial para cuidar da mudança de tipo de uma constante:

```
const_cast<tipo>(expressão)
```

- O terceiro cast é usado para mudar a interpretação de informações:

```
reinterpret_cast<tipo>(expressão)
```

- E finalmente há uma forma de cast usado em combinação com polimorfismo (veja o capítulo 14):

```
dynamic_cast<tipo>(expressão)
```

Assim, em tempo de execução há uma conversão de um apontador a um objeto de certa classe num apontador a um objeto abaixo dentro de sua hierarquia de classe. Neste ponto dentro das Anotações é um pouco cedo para discutir o `dynamic_cast`, mas voltaremos ao tópico na seção 14.5.1.

2.5.5.1: O operador `'static_cast'`

O operador `static_cast<tipo>(expressão)` é usado para converter um tipo em outro tipo aceitável. Por exemplo, `double` em `int`. Um exemplo deste cast é : seja `d` de tipo `double` e `a`, `b` sejam do tipo `int`. Nesta situação, o cálculo do quociente em ponto flutuante de `a` e `b` requer um cast:

```
d = static_cast<double>(a) / b;
```

Se omitimos o cast o operador divisão omitirá o resto, já que seus operandos são do tipo `int`. Note-se que a operação poderia se encontrar fora do cast. Se não a divisão (inteira) será realizada antes do cast ter a oportunidade de converter o tipo de um operando em `double`. Outro bom exemplo de código onde é uma boa idéia usar o operador `static_cast<>()` é onde damos um valor a uma variável de um tipo que vem de uma variável de outro tipo. Seja a expressão (`doubleVar` é uma variável do tipo `double`):

```
intVar += doubleVar;
```

A declaração deve ser avaliada assim:

```
intVar = static_cast<int>(static_cast<double>(intVar) + doubleVar);
```

Primeiro a variável `intVar` é convertida em `double` então somada como `double` a `doubleVar`. Em seguida a soma é transformada pelo cast em `int`. Estas duas conversões estão um pouco exageradas. O mesmo resultado se obtém fazendo-se um cast de `doubleVar` a `int`, obtendo-se assim um valor `int` para a parte direita da expressão:

```
intVar += static_cast<int>(doubleVar);
```

2.5.5.2: O operador `const_cast`

O operador `const_cast<tipo>(expressão)` é usado para desfazer o tipo constante de um tipo (apontador). Digamos que uma função `fun(char *s)` seja real, que faça alguma operação sobre seu parâmetro `char *s`. Ainda mais, digamos que a função não altera a string `s` que recebe como argumento. Como podemos usar a função com uma string como `char const hello[] = "Alô mundo"`?

Ao passar `hello` à `fun()` produz a chamada de atenção (warning): passando `'const char *'` como argumento 1 de `'fun(char *)'`; o que pode ser prevenido usando a chamada:

```
fun(const_cast<char *>(hello));
```

2.5.5.3: O operador `reinterpret_cast`

O operador `reinterpret_cast<tipo>(expressão)` é usado para reinterpretar ponteiros. Por exemplo, usando um `reinterpret_cast<>()` podemos facilmente fazer de um valor `double` bytes individuais. Se `doubleVar` for uma variável de tipo `double` então podemos acessar a seus bytes usando:

```
reinterpret_cast<char *>(&doubleVar)
```

Este exemplo particular também sugere o perigo do `cast`: parece, contudo, que uma string C é produzida, mas sem o final com um byte 0. É um meio de acessar os bytes individuais da memória mantendo o valor `double`.

Mais geralmente: Usar os operadores `cast` é um hábito perigoso, já que suprime o mecanismo de exame de tipo normal do compilador. é aconselhável evitar os casts se possível. Se sob determinadas circunstâncias é imperativo o uso de casts, documente bem as razões para tal uso no código, para assegurar duplamente que o `cast` não seja eventualmente tomado como a causa de um mau comportamento do programa.

2.5.5.4: O operador `dynamic_cast`

O operador `dynamic_cast<>()` é usado no contexto do polimorfismo. A sua discussão fica posposta à seção 14.5.1.

2.5.6: O parâmetro `void`

Na linguagem C, o protótipo de uma função com um parâmetro vazio, tal como:

```
void func();
```

significa que o argumento da função declarada não foi prototipado: o compilador não detetará uso impróprio de argumento. Em C, para declarar uma função sem argumentos a palavra chave `void` é usada:

```
void func(void);
```

Como C++ força um estrito exame de tipos, um parâmetro vazio indica a ausência de qualquer parâmetro. A palavra chave `void` pode, portanto, ser omitida: Em C++ as duas declarações da função `func()` acima são equivalentes.

2.5.7: A declaração `#define __cplusplus`

Cada compilador C++ que segue os standartes ANSI/ISO define o símbolo `__cplusplus`: é como se cada arquivo fonte fosse prefixado com a diretiva de pré-processamento `#define __cplusplus`.

Veremos exemplos do uso deste símbolo nas seções posteriores.

2.5.8: Usando funções do padrão C

As funções normais C, p.ex., coletadas em tempo de execução de uma biblioteca, Também podem ser usadas num programa C++. Tais funções, contudo, devem ser declaradas como funções C.

Como exemplo podemos citar o seguinte fragmento de código que declara `xmalloc()`:

```
extern "C" void *xmalloc(size_t size);
```

Esta declaração é análoga à declaração em C, exceto que o protótipo está prefixado com `extern "C"`.

Um modo um pouco diferente de declarar funções C é o que segue:

```
extern "C"
{
    // As declarações C entram aqui
}
```

Também é possível colocar diretivas ao preprocessador no lugar das declarações. Por exemplo, um arquivo cabeçalho em C, `myheader.h`, que declara as funções C pode ser incluído numa fonte C++ como segue:

```
extern "C"
{
    #include <myheader.h>
}
```

Apesar de que estas duas formas podem ser usadas, atualmente raramente são encontradas em fontes C++. Encontramos mais frequentemente o método para declarar funções C externas da próxima seção.

2.5.9: Arquivos cabeçalhos para ambas C e C++

A combinação do prefixo `__cplusplus` e a possibilidade de definir funções externas `"C"` oferece a habilidade de criar arquivos cabeçalho para ambas as linguagens C e C++. Tais arquivos declaram um grupo de funções que são usáveis em programas tanto em C como em C++.

A configuração de tais arquivos é como segue:

```
#ifdef __cplusplus
extern "C"
{
#endif
    // declarações de dados e funções C são postas aqui. P.ex., void
    // *xmalloc(size_t size);
```

```
#ifndef __cplusplus
}
#endif
```

Usando esta configuração um arquivo cabeçalho C é posto sob `extern "C" {` que está no início do arquivo e por `}` no fim do arquivo. As diretivas `#ifdef` testam o tipo de compilação: C ou C++. Os arquivos cabeçalho 'padrão' C, tais como `stdio.h`, são construídos desta maneira e portanto aproveitáveis para ambas C e C++.

Além disso os cabeçalhos C++ suportam guardas de inclusão. Em C++, em geral, é indesejável incluir o mesmo cabeçalho duas vezes no mesmo arquivo fonte. Podemos evitar facilmente múltiplas inclusões com a diretiva `#ifndef` nesses arquivos. Por exemplo:

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_
    // declarações de cabeçalhos aqui,
    // colocando diretivas #ifdef __cplusplus etc.
#endif
```

Quando este arquivo é percorrido por primeira vez pelo preprocessador, o símbolo `_MYHEADER_H_` ainda não está definido. A condição `#ifndef` é verdadeira e todas as declarações são percorridas. Em consequência o símbolo `_MYHEADER_H_` é definido.

Quando este arquivo for percorrido por uma segunda vez durante a mesma compilação, o símbolo `_MYHEADER_H_` já está definido e conseqüentemente toda informação entre as diretivas `#ifndef` e `#endif` é saltada pelo compilador.

Neste contexto o nome do símbolo `_MYHEADER_H_` serve somente para reconhecimento. Por exemplo, o nome do cabeçalho pode ser usado para este propósito, em maiúsculas e sublinhado no lugar de ponto.

Além de tudo isto os arquivos cabeçalho em C possuem a extensão `.h` e em C++ tais arquivos não possuem extensão. Por exemplo, os `iostreams` `cin`, `cout` e `cerr` ficam disponíveis depois de incluir a diretiva ao preprocessador `#include <iostream>`, no lugar de `#include <iostream.h>` numa fonte. Nas Anotações esta convenção é usada com os cabeçalhos padrão C++, mas não em toda parte (Francamente temos a tendência de não seguir esta convenção: nossos cabeçalhos C++ ainda têm a extensão `.h` e aparentemente ninguém se importa...).

Há mais a ser dito sobre arquivos cabeçalho. Na seção 6.5 a organização preferencial dos cabeçalhos é discutida.

2.5.10: Definindo variáveis locais

Em C as variáveis locais somente podem ser definidas no topo de uma função ou no início de um bloco aninhado. Em C++ as variáveis locais podem ser criadas em qualquer posição dentro do código, mesmo entre declarações (statements).

Ainda mais, pode-se criar variáveis locais dentro de certas declarações, imediatamente antes de seu uso. Um exemplo típico é o comando for:

```
#include <stdio.h>

int main()
{
    for (register int i = 0; i < 20; i++)
        printf("%d\n", i);
    return 0;
}
```

Neste fragmento de código a variável *i* é criada dentro do comando for. De acordo com o estandarte ANSI, a variável não existe antes do comando for e tão pouco além do comando for. Com alguns compiladores mais antigos a variável continua existindo depois da execução do comando for, mas uma chamada de atenção (warning) como:

```
warning: name lookup of `i' changed for new ANSI `for' scoping
using obsolete binding at `i'
(Cuidado: a busca do nome 'i' mudou para o novo 'for' ANSI evitando
conexões obsoletas a 'i')
```

aparecerá quando a variável for usada fora do laço for. A implicação parece clara: definir a variável antes do comando for se esta for usada depois do comando, se não a variável pode ser definida dentro do próprio comando for.

A definição de variáveis quando sejam necessárias requer um pouco de prática. Contudo, eventualmente tende a produzir um código mais legível e a miúdo mais eficiente que definir variáveis no início de declarações compostas. Sugerimos as seguintes regras práticas para definir variáveis:

- As variáveis locais devem ser criadas nos lugares 'intuitivamente corretos', como no exemplo acima. Isto não só vincula o comando for mas também todas as situações onde uma variável só é necessária ali, digamos em meio de uma função.

- Mais geralmente, as variáveis devem ser definidas de tal forma que seus escopos sejam o mais limitado e localizado possível. As variáveis locais não necessariamente devem ser definidas no início de uma função, em seguida à primeira {.
- É considerada boa prática evitar variáveis globais. É muito fácil perder o controle de qual variável está sendo usada com qual propósito. Em C++ raramente se necessita variáveis globais e localizando as variáveis o fenômeno bem conhecido de usar a mesma variável para muitos propósitos invalidando cada propósito individual, facilmente pode ser evitado.

Se considerarmos apropriado, pode-se usar blocos aninhados para localizar as variáveis auxiliares. Existem situações que variáveis locais são consideradas apropriadas dentro de comandos aninhados. O que acabamos de mencionar sobre comandos é certo, mas variáveis locais também podem ser definidas em cláusulas condicionais nos comandos if-else, em cláusulas de seleção do comando switch e cláusulas condicionais do comando while. As variáveis assim definidas serão válidas em toda a extensão do comando, incluindo em seus comandos aninhados. Por exemplo, considere o seguinte comando switch:

```
#include <stdio.h>

int main()
{
    switch (char c = getchar())
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("é vogal %c\n", c);
            break;

        case EOF:
            printf("é EOF\n");
            break;

        default:
            printf("é outro caracter, valor hexa 0x%2x\n", c);
    }
}
```

Note o lugar da definição do caracter 'c': está definido na parte da expressão do comando switch(). Isto implica que aquele 'c' está disponível somente no comando switch, incluindo seus (sub)comandos aninhados, mas não fora do escopo de switch.

O mesmo critério pode ser usado nos comandos `is` e `while`: uma variável definida na parte condicional de um comando `if` ou `while` está disponível em suas declarações aninhadas. Contudo devemos fazer o seguinte:

- A definição da variável deve resultar numa variável iniciada com um valor numérico ou lógico;
- A definição da variável não pode ser aninhada (p. ex. Usando parênteses) numa expressão mais complexa.

O ponto anterior pode não ser uma grande surpresa: para estar em condições de avaliar a condição lógica de um comando `if` ou `while`, o valor da variável deve ser interpretado como zero (falso) ou não-zero (verdadeiro). Usualmente isto não é problema, mas nos objetos C++ (como objetos do tipo `std::string` (capítulo 4) freqüentemente são retornados por funções. Estes objetos podem ou não serem interpretáveis como valor numérico. Se não (como no caso com objetos `std::string`), então tal variável não pode ser definida como condição ou partes da expressão condicional - ou declaração de repetição. O exemplo a seguir, por isso, não compilará

```
    if (std::string myString = getString())          // assume getString()
retorna
    {                                              // um valor std::string
        // processa myString
    }
```

O dito acima merece um esclarecimento. Frequentemente podemos dar a uma variável um escopo local, mas um exame extra é requerido depois de sua iniciação. Ambos a iniciação e o exame não podem ser combinados numa expressão, necessita dois comandos aninhados. Assim o exemplo seguinte tão pouco compilará:

```
    if ((char c = getchar()) && strchr("aeiou", c))
        printf("é uma vogal\n");
```

Se ocorrer uma situação como essa, ou se usa dois comandos aninhados ou se localiza a definição de `char c` usando um comando composto aninhado. Atualmente são possíveis, como usando exceções (exceptions, capítulo 8) e funções especializadas, mas estamos saltando um pouco longe. Neste ponto de nossa discussão sugerimos uma das soluções para remediar o problema introduzido pelo último exemplo:

```
    if (char c = getchar())                        // comandos if aninhados
        if (strchr("aeiou", c))
            printf("é uma vogal\n");

    {                                              // comando composto aninhado
        char c = getchar();
```

```

        if (c && strchr("aeiou", c))
            printf("E uma vogal\n");
    }

```

2.5.11: Sobrecarregando (Overloading) uma Função

Em C++ é possível definir funções com idênticos nomes mas realizando ações diferentes. As funções devem diferir em sua lista de parâmetros (e/ou em seus atributos const). Veja um exemplo abaixo:

```

#include <stdio.h>

void show(int val)
{
    printf("Inteiro: %d\n", val);
}

void show(double val)
{
    printf("Double: %lf\n", val);
}

void show(char *val)
{
    printf("String: %s\n", val);
}

int main()
{
    show(12);
    show(3.1415);
    show("Alô Mundo\n!");
}

```

No fragmento acima são definidas três funções `show()`, que diferem somente por seus parâmetros: `int`, `double` e `char *`. As funções com nomes idênticos são chamadas 'funções sobrecarregadas' ('function overloading').

É interessante a forma, bastante simples, como o compilador C++ implementa funções sobrecarregadas. Apesar de que as funções compartilhem o mesmo nome no texto fonte (neste exemplo `show()`), o compilador (e portanto o lincador) usa nomes completamente diferentes. A conversão do nome do arquivo fonte ao usado internamente é chamada 'name mangling'. O compilador C++ converte o nome

`void show(int)` ao nome interno `VshowI`, enquanto uma função análoga com `char *` será chamada `VshowCP`. Os nomes usados internamente dependem do compilador e não são relevantes ao programador, exceto onde esses nomes mostrados acima que participam de uma lista de uma biblioteca.

Algumas notas respeito a funções sobrecarregadas são:

- Não use sobrecarga de funções que realizam diferentes tarefas. No exemplo acima as funções `show()` são, de certa forma, correlatas (imprimem informação na tela)

Também é bem possível definir duas funções `lookup()`, uma que busca um nome numa lista, enquanto a outra determina o modo de vídeo. Neste caso as duas funções não têm nada em comum, exceto o nome. Portanto seria mais prático usar nomes sugestíveis com a ação; digamos `findname()` e `vidmode()`.

- A linguagem C++ não permite nomes de funções idênticos, que difiram somente no valor retornado, já que é sempre uma escolha do programador ignorar ou não o valor de retorno de uma função. P. ex. O fragmento:

```
printf("Alô Mundo!\n");
```

Não carrega informação sobre o valor retornado da função `printf()`. Duas funções `printf()` que se distinguiriam somente pelo valor retornado não poderiam ser distinguidas pelo compilador.

- A sobrecarga de função pode produzir surpresas. P.Ex. Imagine um comando como:

```
show(0);
```

Dadas as três funções `show()` acima. O zero poderia, aqui, ser interpretado como um apontador `NULL` a um `char`, isto é, um `(char *)0`, ou como um inteiro com valor zero. Aqui a C++ chamará uma função esperando um argumento inteiro, que poderia não ser o que se espera.

- No capítulo 6 a noção de membros constantes de funções será introduzida (seção 6.2). Aqui somente é mencionado que as classes normalmente têm, as assim chamadas, funções associadas (veja o capítulo 4 para uma introdução informal ao conceito).

Além de funções membro sobrecarregadas em suas listas de parâmetros, é possível sobrecarregar uma função através de seus atributos constantes. Nestes casos, as classes podem possuir pares de funções membro, tendo listas de parâmetros idênticas. Então estas funções são sobrecarregadas através de seus atributos constantes: uma destas funções deve ter o atributo constante e a outra não.

2.5.12: Argumentos padrão de uma função

Em C++ é possível fornecer 'argumentos padrão' ao definir uma função. Estes argumentos são entregados pelo compilador quando não são especificados pelo programador. Por exemplo:

```
#include <stdio.h>

void showstring(char *str = "Alô Mundo!\n");

int main()
{
    showstring("Aqui está um argumento explícito.\n");

    showstring();           // de fato isto diz:
                           // showstring("Alô Mundo!\n");
}
```

A possibilidade de omitir argumentos em situações onde os argumentos padrão são definidos é um excelente objetivo: o compilador suprirá o argumento omitido, a menos de que seja explicitamente especificado na chamada. O código do programa fica sempre menor ou mais eficiente.

As funções podem ser definidas com mais de um argumento padrão:

```
void two_ints(int a = 1, int b = 4);

int main()
{
    two_ints();           // argumentos: 1, 4
    two_ints(20);         // argumentos: 20, 4
    two_ints(20, 5);      // argumentos: 20, 5
}
```

Quando a função `two_ints()` é chamada o compilador entrega um ou dois argumentos quando necessário. Um comando como `two_ints(6)`, contudo, não é permitido: quando se omitem argumentos deve ser do lado direito.

Os argumentos padrão devem ser conhecidos pelo compilador quando o código seja gerado os argumentos devem ser fornecidos. Para isto, os argumentos padrão são usualmente mencionados na declaração da função:

```
// amostra de arquivo cabeçalho
extern void two_ints(int a = 1, int b = 4);

// código da função em, digamos, two.cc
void two_ints(int a, int b)
{
```

```
    ...  
}
```

Note que fornecer os argumentos na definição da função no lugar do arquivo cabeçalho não é correto: O compilador lerá o cabeçalho e não a definição da função quando esta for usada em outras fontes. Consequentemente, nestes casos o compilador não poderá determinar os argumentos padrão da função.

2.5.13: A palavra chave `typedef`

A palavra chave typedef é permitida em C++, mas não é mais requerida nas definições de união, estrutura ou enumeração (union, struct e enum).

Este fato está ilustrado no seguinte exemplo:

```
struct somestruct  
{  
    int      a;  
    double   d;  
    char     string[80];  
};
```

Quando uma estrutura, união ou outro tipo composto é definido, o rótulo deste tipo pode ser usado como o nome do tipo (como somestruct no exemplo acima):

```
somestruct what;  
  
what.d = 3.1415;
```

2.5.14: Funções como parte de uma estrutura

Em C++ é permitido definir funções como parte de estruturas. Aqui encontramos o primeiro exemplo de um objeto: como descrito anteriormente (veja seção 2.4), um objeto é uma estrutura contendo todo o código e dados envolvidos.

Uma definição de struct point é dada no fragmento de código abaixo. Nesta estrutura dois campos de dados e uma função draw() são declarados.

```
struct point          // definição de um ponto  
{                    // na tela:  
    int x;            // coordenadas
```

```

    int y;                // x/y
    void draw(void);      // função de desenho
};

```

Uma estrutura semelhante poderia ser parte de um programa de desenho, p. ex., poderia representar um ponto no desenho. Respeito a esta estrutura deve-se notar o seguinte:

- A função draw(), mencionada na definição da estrutura é uma mera declaração. O código da função, ou em outras palavras, as ações que desenvolve estão em outra parte. Descreveremos as definições das funções dentro de estruturas mais tarde (veja seção 3.2).
- O tamanho da estrutura point é igual a seus dois inteiros. Uma função declarada dentro de uma estrutura não afeta seu tamanho. O compilador executa este comportamento permitindo que a função draw() seja conhecida somente no contexto de point.

A estrutura point poderia ser usada como segue:

```

point a;                // dois pontos
point b;                // na tela

a.x = 0;                // define o primeiro ponto
a.y = 10;               // e o desenha
a.draw();

b = a;                  // copia a em b
b.y = 20;               // redefine a coordenada y
b.draw();                // e desenha-o

```

A função que é parte da estrutura é selecionada de maneira similar é dos campos de dados; isto é, usando o operador de seleção de campo (.). Quando usamos apontadores para estruturas pode-se usar (->).

A idéia por trás desta construção sintética é que muitos tipos podem conter funções com nomes idênticos. Por exemplo, uma estrutura que represente um círculo deveria conter três valores inteiros: dois valores para as coordenadas do centro do círculo e um valor para o raio. Analogamente à estrutura point a função draw() pode ser declarada que desenharia o círculo.

CAPÍTULO 3: Uma primeira impressão da C++

Neste capítulo exploraremos a C++ mais profundamente. A possibilidade de se declarar funções dentro de estruturas está ilustrada em vários exemplos. O conceito de classe é introduzido.

3.1: Mais extensões à C em C++

Antes de continuar com a solução para programação com objetos 'reais', primeiro queremos introduzir mais algumas extensões à linguagem de programação C: não meras diferenças entre C e C++, mas construções sintáticas e palavras chave que não pertencem à C.

3.1.1: O operador resolução do escopo ::

A linguagem C++ introduz diversos novos operadores, entre os quais o operador resolução do escopo (::). Este operador pode ser usado em situações onde existe uma variável global com o mesmo nome de uma variável local:

```
#include <stdio.h>

int counter = 50;                                // variável global

int main()
{
    for (register int counter = 1; // isto se refere à
         counter < 10;             // variável local
         counter++)
    {
        printf("%d\n",
                ::counter           // a variável global
                /                   // dividida pela
                counter);           // variável local
    }
    return 0;
}
```

Neste fragmento de código o operador de escopo é usado para endereçar uma variável global em lugar da variável local com o mesmo nome. Em C++ o operador de escopo é usado extensivamente,

mas raramente é usado para acessar uma variável global ofuscada por uma local com nome idêntico. Seu propósito principal será descrito no capítulo 6.

3.1.2: `cout`, `cin` e `cerr`

Analogamente à C, a C++ define streams padrão de entrada e saída que são abertas quando um programa é executado. As streams são:

- `cout`, análoga a `stdout`,
- `cin`, análoga a `stdin`,
- `cerr`, análoga a `stderr`.

Sintaticamente estas streams não são usadas como funções: em vez disso os dados escritos ou lidos delas usam o operador `<<`, chamado operador inserção e `>>`, chamado operador extração. Isto está ilustrado no seguinte exemplo:

```
#include <iostream>

using namespace std;

int main()
{
    int    ival;
    char   sval[30];

    cout << "Entre um número:" << endl;
    cin >> ival;
    cout << "E agora uma string:" << endl;
    cin >> sval;

    cout << "O número é: " << ival << endl
         << "E a string é: " << sval << endl;
}
```

Este programa lê um número e uma string da stream `cin` (usualmente o teclado) e imprime estes dados em `cout`. Com respeito às streams, por favor note:

- As streams padrão estão declaradas no cabeçalho `iostreams`. No exemplo das Anotações este arquivo cabeçalho freqüentemente não está mencionado explicitamente. é indispensável incluí-lo

(direta ou indiretamente) quando estas streams são usadas. Comparável ao uso do namespace std; o leitor espera a cláusula `#include <iostream>` em todos os exemplos onde as streams padrão são usadas.

- As streams `cout`, `cin` e `cerr` são de fato 'objetos' de uma dada classe (mais sobre classes adiante) que processa as entradas e saídas de um programa. Note que o termo 'objeto', como usado aqui, significa 'variável' de uma classe particular.
- A stream `cin` extrai dados de uma stream e copia-os a variáveis (p. ex., `ival` no exemplo acima) usando o operador extrator (dois caracteres consecutivos `>`: `>>`). Descreveremos mais tarde como os operadores em C++ podem realizar ações muito diferentes para as quais foram definidos na linguagem, como é o caso aqui. A sobrecarga de funções já foi aqui mencionada. Em C++ os operadores também podem ter definições múltiplas, que se chama sobrecarga de operadores.
- Os operadores que manipulam `cin`, `cout` e `cerr` (i.e., `>>` e `<<`) também manipulam variáveis de diferentes tipos. No exemplo acima `cout << ival` resulta na impressão de um valor inteiro, já em `cout << "Entre um número"` resulta na impressão de uma string. As ações dos operadores dependem dos tipos das variáveis.
- O operador extração (`>>`) executa uma adjudicação segura a uma variável, 'extraíndo' seu valor de uma stream de texto. Normalmente o operador extração saltará todos os caracteres de espaços em branco que precedem os valores a serem extraídos.
- Constantes simbólicas especiais são usadas para situações especiais. A terminação de uma linha escrita por `cout` é usualmente feita inserindo-se o símbolo `endl`, no lugar de `"\n"`. O `"\n"` ainda pode ser inserido, mas este não fará a descarga do bufer, ao contrário, o bufer será descarregado inserindo-se `endl`.

As streams `cin`, `cout` e `cerr` não fazem parte da gramática de C++, como definidas no compilador quando faz a varredura dos arquivos fonte. As streams fazem parte das definições do arquivo cabeçalho `iostream`. Como ocorre com funções como `printf()` que não fazem parte da gramática de C, mas originalmente escrita por alguém que considerava tal função importante e a pôs na biblioteca de tempo de execução.

Um programa ou usa o estilo antigo como `printf()` e `scanf()` ou usa o novo estilo das streams, é uma questão de gosto. Ambos estilos não devem ser misturados. Um número de vantagens e desvantagens é dado abaixo:

- Comparado com o padrão C, as funções `printf()` e `scanf()`, o uso dos operadores inserção e

extração é mais seguro. Os formatos de strings que são usados com `printf()` e `scanf()` pode definir mal os especificadores de formatos para seus argumentos, por isso o compilador às vezes não pode gerar uma chamada de atenção. Em contraste, o exame de tipos com `cin`, `cout` e `cerr` é feito pelo compilador. Conseqüentemente é impossível um erro com os argumentos nos lugares onde deve aparecer um `int` aparecer uma string.

- As funções `printf()` e `scaNf()` e outras funções que usam formato string, de fato introduzem uma mini-linguagem interpretada em tempo de execução. Em contraste o compilador C++ sabe exatamente qual entrada ou saída dado argumento usa.
- O uso de operadores shift-esquerdo e shift-direito no contexto das streams ilustra as possibilidades da C++. Novamente, é necessário um pouco de prática para ascender da C, mas depois disso estes operadores sobrecarregados nos fazem sentir mais confortáveis.
- As iostreams são extensíveis: nova funcionalidade pode ser facilmente acrescentada às já existentes, um fenômeno chamado herança. A herança é discutida em detalhe no capítulo 13.

A biblioteca `iostream` tem muito mais a oferecer que somente `cin`, `cout` e `cerr`. No capítulo 5 cobriremos `iostream` em grande detalhe. Mesmo `printf()` e amigas podem ainda serem usadas em programas C++, as streams estão praticamente substituindo o antigo estilo da C de funções de E/S como `printf()`. Se pensamos que ainda se necessita usar `printf()` e funções correlatas, pensemos outra vez: nesse caso, provavelmente, não se compreendeu ainda completamente as possibilidades dos objetos stream.

3.1.3: A palavra chave `'const'`

A palavra chave `'const'` é vista muito a miúdo em programas C++. Contudo `const` é parte da gramática C, em C `const` é usada muito menos freqüentemente.

A palavra chave `const` é um modificador que estabelece o valor de uma variável ou de um argumento que não pode ser modificado. No exemplo a seguir a intenção é mudar o valor da variável `ival`, que falha:

```
int main()
{
    int const ival = 3;           // uma constante int
                                  // iniciada em 3

    ival = 4;                     // a adjudicação produz
                                  // uma mensagem de erro
```

```
}
```

Este exemplo mostra como ival pode ser iniciada com um valor dado em sua definição; intentos de mudar seu valor mais tarde (numa adjudicação) não são permitidos.

As variáveis declaradas com o tipo const podem, em contraste com a C, ser usadas como a especificação do tamanho de um conjunto (array), como no seguinte exemplo:

```
int const size = 20;  
char buf[size];           // tamanho de 20 chars
```

Outro uso da palavra chave const é visto na declaração de ponteiros, p. ex., argumentos ponteiro. Na declaração:

```
char const *buf;
```

buf é uma variável ponteiro que aponta para chars. Nada mais pode ser apontado por buf, já que é declarada const, não pode ser mudada: Os chars são declarados como const. O ponteiro buf pode ser mudado. Uma adjudicação como *buf = 'a'; por isso não é permitido, mas buff++ sim.

Na declaração:

```
char *const buf;
```

A variável buf é um ponteiro const que não pode ser mudado. Mas o que seja apontado por buf pode mudar é vontade.

Finalmente a declaração:

```
char const *const buf;
```

Também é possível, aqui nenhum pode ser mudado, nem o apontador nem o apontado.

O papel da regra na localização da palavra chave const é como segue: seja o que for que ocorra é esquerda da palavra chave não pode ser mudado.

Contudo esta regra não é usada muito seguidamente. Por exemplo:

Bjarne Stroustrup afirma (em http://www.research.att.com/~bs/bs_faq2.html#constplacement)

Posso por "const" antes ou depois do tipo?

Eu ponho antes, mas é uma questão de gosto. "const T" e "T const" sempre foi (ambos)

permitidos e equivalentes. Por exemplo:

```
const int a = 1;           // ok
int const b = 2;           // também ok
```

Meu conselho é usar a primeira versão será menos confuso aos programadores ("é mais idiomático").

Abaixo veremos um exemplo onde esta simples localização 'anterior' da palavra chave `const` produz um resultado inesperado (i.é., indesejado). Aparte disto, a localização 'idiomática' anterior conflita com a noção de funções `const`, que encontramos na seção 6.2, onde a palavra chave `const` também é escrita atrás do nome da função.

A definição ou declaração onde `const` é usada deve ser lida do identificador da variável ou função para trás ao tipo de identificador:

```
``Buf é um ponteiro const a caracteres const"
```

Esta regra prática é especialmente útil em casos de possíveis confusões. Em exemplos de código C++, a miúdo encontramos o contrário: `const` precedendo aquilo que não pode ser alterado. Isto pode ser num código desleixado, como indicado pelo nosso segundo exemplo acima:

```
char const *buf;
```

O que deve ficar constante aqui? De acordo com a interpretação desleixada o ponteiro não pode ser alterado (já que `const` precede o ponteiro). De fato os valores de `char` são entidades constantes aqui, como aclararemos ao intentarmos compilar o seguinte programa:

```
int main()
{
    char const *buf = "Olá!";

    buf++;                // aceito pelo compilador
    *buf = 'u';           // rejeitado pelo compilador

    return 0;
}
```

A compilação falhou no comando `*buf = 'u';`, não em `buf++;`.

Marshall Cline's

Do C++ FAQ a mesma regra (parágrafo 18.5), num contexto similar:

[18.5] Qual é a diferença entre "`const Fred* p`", "`Fred* const p`" e "`const Fred* const p`"?

lê-se a declaração do ponteiro da direita para a esquerda.

A afirmação de Marshal Clines precisa ser comprovada, digamos: Começar a leitura das definições de apontadores (e declarações) a partir do nome da variável, lendo até o fim da definição. Uma vez chegando num parênteses fechado, a leitura continua para trás do ponto inicial da leitura, da direita para a esquerda, até o parênteses aberto ou o início da definição. Por exemplo, considere a declaração complexa seguinte:

```
char const *(* const (*ip)[])[]
```

Aqui vemos:

- a variável ip, sendo um
- (lendo para trás) ponteiro modificável de um
- (lendo para frente) conjunto de
- (lendo para trás) ponteiros constantes a um
- (lendo para frente) conjunto de
- (lendo para trás) ponteiros modificáveis a caracteres constantes

3.1.4: referências

Além dos meios bem conhecidos de definir variáveis, variáveis plenas ou ponteiros a c++ permite 'referências' definidas como sinônimo de variáveis. Uma referência a uma variável é como um apelido; o nome da variável e sua referência podem participar de comandos que envolvam a variável:

```
int int_value;  
int &ref = int_value;
```

No exemplo acima a variável int_value é definida. Subseqüentemente uma referência ref é definida que (devido à sua iniciação) se refere à mesma localização de memória que int_vlue. Na definição de ref, o operador referência & indica que ref não é por si um inteiro, mas a referência a um inteiro. Os dois comandos:

```
int_value++;           // alternativa 1  
ref++;                // alternativa 2
```

têm o mesmo efeito, como esperado: Em algum lugar da memória um valor inteiro será incrementado por uma unidade. Essa localidade de memória pode ser referida como `int_valeu` ou `ref`, sem distinção.

As referências desenvolvem um papel importante em C++, como um meio de passar argumentos que podem ser modificados. P. ex., em C padrão, uma função que incrementa o valor de seu argumento em 5 unidades mas não retorna nada (`void`), necessita um parâmetro apontador.

```
void increase(int *valp)    // espera um ponteiro
{                          // a um inteiro
    *valp += 5;
}

int main()
{
    int x;

    increase(&x)            // o endereço de x é
    return 0;              // passado como argumento
}
```

Esta construção também pode ser usada em C++, mas o mesmo efeito pode ser obtido usando uma referência:

```
void increase(int &valr)    // espera uma referência
{                          // a um inteiro
    valr += 5;
}

int main()
{
    int x;

    increase(x);            // referência a x é
    return 0;              // passada como argumento
}
```

Pode-se argumentar qual seria dos códigos acima mais claro: o comando `increase(x)` na função `main()` sugere que não é passado o próprio `x` mas uma sua cópia. Mas o valor de `x` muda devido à forma como `increase()` está definida.

As referências são realizadas usando ponteiros. Portanto, referências em C++ são justamente ponteiros, no que concerne ao compilador. Contudo, o programador não necessita saber ou incomodar-se sobre níveis de indireção. Tão pouco pode-se distinguir entre ponteiros e referências: uma vez iniciadas as referências nunca mais podem referenciar outra variável, ao passo que os valores de um ponteiro

podem mudar, que resultaria num apontador a uma variável apontando a outra localidade de memória. Por exemplo:

```
extern int *ip;
extern int &ir;

ip = 0;      // revaloriza ip, agora é um ponteiro 0
ir = 0;      // ir sem mudança, a variável int a que se referia
              // agora é 0.
```

Afim de prever confusão sugerimos a aderir às regras seguintes:

- Nasquelas situações onde uma chamada a uma função não altera seus argumentos de tipos primitivos, uma cópia das variáveis pode ser passada:

```
void some_func(int val)
{
    cout << val << endl;
}

int main()
{
    int x;

    some_func(x);      // uma cópia é passada, assim
    return 0;          // x não será mudada
}
```

- Quando uma função muda o valor de seus argumentos, um parâmetro apontador é preferível. Estes parâmetros apontadores de preferência devem ser os parâmetros iniciais da função. Isto é conhecido como 'retorno por argumento'.

```
void by_pointer(int *valp)
{
    *valp += 5;
}
```

- Quando uma função não altera o valor de seus argumentos de classe ou de tipo estrutura, ou ainda se a modificação do argumento é de efeito trivial de efeito colateral (p. ex. O argumento é uma stream), deve-se usar referências. As referências constantes devem ser usadas se a função não altera o argumento:

```
void by_reference(string const &str)
{
    cout << str;
}
```

```

int main ()
{
    int x = 7;
    string str("Alô");

    by_pointer(&x);          // é passado um ponteiro
    by_reference(str);       // str não é alterada
    return 0;                // x ode ser alterada
}

```

As referências jogam um papel importante nos casos onde o argumento não será alterado pela função, mas onde é indesejável que o argumento inicie o parâmetro. Esta situação ocorre quando uma variável muito grande, p. ex. uma estrutura, é passada como argumento, ou é retornada pela função. Nestes casos a cópia tende a ser um fator importante, já que toda a estrutura deve ser copiada. Assim, nesses casos é preferível usar referências. Se o argumento não é mudado pela função ou quem chamou não muda a informação de retorno, o uso de uma palavra chave const pode ser usada. Considere o seguinte exemplo:

```

struct Person                                // uma grande estrutura
{
    char    name[80],
    char    address[90];
    double  salary;
};

Person person[50];                          // base de dados de pessoas
// printperson espera uma
void printperson (Person const &p)
{
    // referência a uma estrutura
    // mas não a altera
    cout << "Name: " << p.name << endl <<
        "Address: " << p.address << endl;
}

// apanha uma pessoa pelo valor do índice
Person const &person(int index)
{
    return person[index]; // é retornada uma referência,
}                          // não uma cópia de person[index]

int main()
{
    Person boss;

    printperson (boss); // não é passado um ponteiro,
                        // porque a variável não será

```



```

// alterada pela função
printperson(person(5));
// referências, não cópias
// são passadas aqui
return 0;
}

```

Ainda mais, deve-se notar que há outra razão para usar referências ao passar objetos como argumentos de funções: passando a referência a um objeto evita-se a ativação do afamado construtor de cópias. Os construtores de cópias serão vistos no capítulo 7.

As referências podem resultar num código extremamente 'feio'. Uma função pode retornar a referência a uma variável, como no exemplo seguinte:

```

int &func()
{
    static int value;
    return value;
}

```

Isto permite as seguintes construções:

```

func() = 20;
func() += func();

```

Provavelmente é supérfluo notar que esta construção normalmente não se usará. Apesar disso, existem situações onde é útil retornar uma referência. Já vimos um exemplo deste fenômeno na discussão anterior de streams. Num comando como `cout <<"Olá" << endl;`, o operador inserção retorna uma referência a `cout`. Neste comando primeiro é inserido em `cout` "Olá", produzindo uma referência a `cout`. Através desta referência `endl` é, então, inserido no objeto `cout`, produzindo outra vez uma referência a `cout`. Esta última referência não é usada.

Uma série de diferenças entre ponteiros e referências está listada abaixo:

- Uma referência não pode existir por si só, i.e., sem algo a que referir-se. Uma declaração como `int &ref;` não é permitida a que se refere `ref`?
- Contudo pode ser declarada como externa. Estas referências são referidas mais adiante.
- As referências podem ser parâmetros de funções: são iniciadas quando a função é chamada.
- As referências podem ser mudadas entre os tipos de retorno de funções. Nestes casos a função determina a que se refere o valor de retorno.

- As referências podem ser usadas como membros de classes. Retornaremos a este uso mais tarde.
- Em contraste com as referências, os ponteiros são variáveis por si. Apontam a algo ou a 'nada'.
- As referências são apelidos de outras variáveis e não podem mudar de variável. Uma vez definida a referência se referirá a uma variável particular.
- Em contraste, os apontadores podem apontar a diferentes variáveis.
- Quando um endereço do operador & é usado com uma referência, a expressão contém o endereço da variável para a qual a referência foi aplicada. Em contraste os ponteiros ordinários são variáveis por si sós, assim, o endereço de uma variável ponteiro nada tem a ver com o endereço da variável apontada.

3.2: Funções como parte de estruturas

Acima foi mencionado que as funções podem ser parte de estruturas (veja seção 2.5.14). Tais funções são chamadas funções membros ou métodos.

Esta seção discute como definir tais funções.

O fragmento de código abaixo ilustra uma estrutura com campos de dados nomeados com nome e endereço. Uma função print() está incluída na definição da estrutura:

```
struct Person
{
    char name[80],
    char address[80];

    void print();
};
```

A função membro print() é definida usando-se o nome da estrutura (person) e o operador escopo de resolução (::):

```
void Person::print()
{
    cout << "Name:      " << name << endl
         << "Address:   " << address << endl;
}
```

Na definição desta função membro, o nome da função é precedido pela palavra chave struct

seguida por ::. O código da função mostra como os campos da estrutura podem se endereçados sem o uso do nome do tipo: neste exemplo a função print() imprime o nome de uma variável. Enquanto print() é parte da estrutura person, o nome da variável implicitamente se refere ao mesmo tipo.

Esta estrutura poderia ser usada como segue:

```
Person p;  
  
strcpy(p.name, "Karel");  
strcpy(p.address, "Rietveldlaan 37");  
p.print();
```

A vantagem das funções membro está no fato de que ao se chamar a função automaticamente se endereça os campos de dados da estrutura que foram invocados. Como no comando p.print() a estrutura p é o 'substrato'; os nomes das variáveis usadas no código da print() se referem à mesma struct p.

3.3: Diversos novos tipos de dados

Em C os tipos básicos de dados são: void, char, short, int, long, float e double. A C++ estende estes cinco tipos com diversos novos tipos: os tipos bool, wchar_t, long e long double (Cf. ANSI/ISO (1995), par. 27.6.2.4.1 para exemplos desses tipos muito longos). O tipo longo longo é meramente um tipo double-long double. Além destes tipos básicos um tipo padrão string é definido. O tipo bool e wchar_t são vistos nas próximas seções e o tipo string no capítulo 4.

Agora que estes novos tipos foram introduzidos, permita-nos refrescar sua memória sobre as letras que podem ser usadas em constantes literais de vários tipos. Elas são:

- E ou e: caracter de exponenciação em valores literais em ponto flutuante. Por exemplo: 1.23E+3. Aqui E deve ser pronunciado e interpretado como: expoente vezes 10. Portanto 1.23E+3 representa o valor 1230.
- F é usado como pós-fixado de uma constante numérica fracionária para indicar um valor do tipo flutuante, melhor dito double, que é o padrão. Por exemplo: 12.F (o ponto transforma o 12 num valor em ponto flutuante); 1.23E+3F (veja o exemplo anterior. 1.23E+3 é um valor double, já 1.23E+3F é um valor de ponto flutuante).
- L pode ser usado como prefixo para indicar uma string de caracteres cujos elementos são do tipo de caracter wchar_t. Por exemplo: L"Alô Mundo".

- L pode ser usado como pós-fixado de um valor inteiro do tipo longo, em vez de int, que é o padrão. Note que não há uma letra que indique o tipo short. Para isto é necessário usar `static_cast<short>()`.
- U pode ser usado como pós-fixado para indicar um valor sem sinal, no lugar de int. Pode também ser combinado com o pós-fixado L para produzir um valor longo sem sinal.

3.3.1: O tipo de dados 'bool'

Nesta seção o tipo bool será introduzido.

O tipo bool representa valores booleanos (lógicos), para isso os valores (agora reservados) verdadeiro (true) e falso (false) podem ser usados. Além desses valores reservados valores inteiros também podem ser adjudicados às variáveis do tipo bool, que serão implicitamente convertidos em verdadeiro ou falso de acordo com as regras de conversão (assume que intValue é uma variável int e boolValue é uma variável bool):

```
// de int a bool:

boolValue = intValue ? true : false;

// de bool a int:

intValue = boolValue ? 1 : 0;
```

Ainda, quando valores bool são inseridos em, p.ex., cout, então 1 é escrito para valores verdadeiros e 0 para valores falsos. Considere o seguinte exemplo:

```
cout << "A valor verdadeiro: " << true << endl
      << "A valor falso: " << false << endl;
```

Os dados tipo booleanos são encontrados em outras linguagens de programação também. O Pascal tem seu tipo Boolean e Java o tipo boolean. A diferença destas linguagens, o tipo bool da C++ atua como uma espécie de inteiro: sua primeira documentação tinha somente dois valores verdadeiro e falso. Agora estes valores podem ser interpretados como valores de enumeração para 1 e 0. Ao fazermos isto podemos negligenciar o conceito por trás do tipo booleana, mas: adjudicando um valor verdadeiro a uma variável int não produz nem chamada de atenção e nem erros.

O uso do tipo bool geralmente é mais intuitivamente claro que usar int. Considere os seguintes protótipos:

```
bool exists(char const *fileName); // (1)
int  exists(char const *fileName); // (2)
```

No primeiro protótipo (1) muitos esperarão que a função retorne verdadeiro se o nome de arquivo dado é o nome de um arquivo existente. Contudo usando o segundo protótipo aparecem algumas ambigüidades: intuitivamente o valor retornado pareceria ser 1, já que se subentende de construções como:

```
if (exists("myfile"))
    cout << "myfile exists";
```

Por outro lado muitas funções (como access(), stat(), etc.) retornam 0 para indicar um sucesso, reservando outros valores para indicar vários tipos de erros.

Como regra prática sugerimos o seguinte: se uma função informa a quem a chamou sobre um sucesso ou falha de seu trabalho faça a função retornar um tipo bool. Se a função pode retornar sucesso ou vários tipos de erros faça a função retornar valores enum, documentando a situação do retorno da função. Somente quando a função retorna um valor inteiro significativo (como a soma de dois valores inteiros) faça a função retornar um valor inteiro.

3.3.2: O tipo de dados 'wchar_t'

O tipo 'wchar_t' é uma extensão ao tipo básico char, para acomodar valores grandes de caracteres, tal como o conjunto de caracteres Unicode. O compilador g++ (versão 2.95 ou maior) reporta sizeof(wchar_t) como 4, que facilmente acomoda todos os 65.536 valores diferentes dos caracteres Unicode.

Note que uma linguagem de programação como Java possui um tipo char comparável ao tipo wchar_t da C++. O tipo char da Java possui 2 bytes. Por outro lado, o tipo de dado byte da Java é comparável ao tipo char da C++. Muito conveniente....

3.3.3: O tipo de dados 'size_t'

O tipo size_t não é realmente um tipo de dados primitivo interno, mas um tipo de dados promovido pelo POSIX como um nome de tipo a ser usado para valores não-negativos inteiros. Não é um tipo específico C++, também disponível em, p. ex., C. Deve ser usado no lugar de unsigned int. Usualmente é definido implicitamente quando um sistema de arquivos cabeçalhos é incluído. O arquivo cabeçalho que 'oficialmente' define size_t no contexto C++ é cstdint.

Usar o `size_t` tem a vantagem de ser um tipo conceitual, antes que um tipo padrão que seja modificado então por um modificador. Assim, melhora a auto-documentação do código de fonte.

O tipo `size_t` deve ser usado em todas as situações onde os valores inteiros não-negativos são pretendidos. Às vezes as funções requerem explicitamente o uso de `unsigned int`. Por exemplo, em arquiteturas amd a função das X-windows, `XqueryPointer`, requer explicitamente um ponteiro a uma variável inteira sem sinal como um de seus argumentos. Nesta situação particular um ponteiro a uma variável `size_t` não pode ser usada. Esta situação é excepcional. Geralmente `size_t` pode ser usado (e deve) onde valores sem sinal são pretendidos.

Outros tipos bastante úteis existem também. Por exemplo, `uns32_t` é garantido para guardar valores de 32-bits sem sinal. Analogamente, `int32_t` guarda valores sinalizados de 32-bits. Os tipos correspondentes existem para valores de 8, 16 e 64 bits. Estes tipos são definidos no arquivo cabeçalho `stdint.h`.

3.4: Palavras chave em C++

As Palavras chave da C++ conformam um super-conjunto daquelas da C. Eis uma lista de todas as palavras chave da linguagem:

<code>and</code>	<code>const</code>	<code>float</code>	<code>operator</code>	<code>static_cast</code>	<code>using</code>
<code>and_eq</code>	<code>const_cast</code>	<code>for</code>	<code>or</code>	<code>struct</code>	<code>virtual</code>
<code>asm</code>	<code>continue</code>	<code>friend</code>	<code>or_eq</code>	<code>switch</code>	<code>void</code>
<code>auto</code>	<code>default</code>	<code>goto</code>	<code>private</code>	<code>template</code>	<code>volatile</code>
<code>bitand</code>	<code>delete</code>	<code>if</code>	<code>protected</code>	<code>this</code>	<code>wchar_t</code>
<code>bitor</code>	<code>do</code>	<code>inline</code>	<code>public</code>	<code>throw</code>	<code>while</code>
<code>bool</code>	<code>double</code>	<code>int</code>	<code>register</code>	<code>true</code>	<code>xor</code>
<code>break</code>	<code>dynamic_cast</code>	<code>long</code>	<code>reinterpret_cast</code>	<code>try</code>	<code>xor_eq</code>
<code>case</code>	<code>else</code>	<code>mutable</code>	<code>return</code>	<code>typedef</code>	
<code>catch</code>	<code>enum</code>	<code>espaço nomeado</code>	<code>short</code>	<code>typeid</code>	
<code>char</code>	<code>explicit</code>	<code>new</code>	<code>signed</code>	<code>typename</code>	
<code>class</code>	<code>extern</code>	<code>not</code>	<code>sizeof</code>	<code>union</code>	
<code>compl</code>	<code>false</code>	<code>not_eq</code>	<code>static</code>	<code>unsigned</code>	

Note as palavras chave operadores: `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor` and `xor_eq` são alternativas simbólicas para, respectivamente, `&&`, `&=`, `&`, `!`, `!=`, `||`, `!|`, `^` e `^=`.

3.5: Encobrimento de dados: *public*, *private* e *class*

Como mencionamos antes (veja a seção 2.3), a C++ contém possibilidades sintáticas especiais para introduzir o encobrimento de dados. Encobrimento de dados é a habilidade de uma parte de um programa encobrir seus dados de outras partes; evitando endereçamentos impróprios ou colisões de nomes.

A C++ tem três palavras chave especiais relativas ao encobrimento de dados: *private*, *protected* e *public*. Estas palavras chave podem ser usadas na definição de uma estrutura. A palavra chave *public* define que todos os campos subseqüentes da estrutura são acessíveis por todo código; a palavra chave *private* define que todos os campos subseqüentes somente são acessíveis pelo código que é parte da estrutura (i.e., somente acessíveis às funções membro). A palavra chave *protected* é discutida no capítulo 13 e está além do escopo da discussão corrente.

Numa estrutura todos os campos são *public*, a menos do estabelecimento explícito do contrário. Usando este conhecimento podemos expandir a estrutura *Person*:

```
struct Person
{
    private:
        char d_name[80];
        char d_address[80];
    public:
        void setName(char const *n);
        void setAddress(char const *a);
        void print();
        char const *name();
        char const *address();
};
```

Os campos *d_name* e *d_address* somente são acessíveis às funções membro definidas na estrutura: nas funções *setName()*, *setAddress()* etc.. Isto pelo fato de que os campos *d_name* e *d_address* estão precedidos pela palavra chave *private*. Como ilustração consideremos a fração de código:

```
Person x;

x.setName("Frank");           // ok, setName() é public
strcpy(x.d_name, "Knarf");    // erro, name é private
```

O encobrimento dos dados é feito como segue: Os dados da estrutura *Person* são mencionados na definição da estrutura. O mundo exterior acessa os dados usando funções especiais, que também fazem parte da definição. Essas funções membro controlam todo o tráfego entre os campos de dados e outras partes do programa e por isso também são chamadas funções 'interface'. O encobrimento dos dados assim realizado está ilustrado na figura 2.

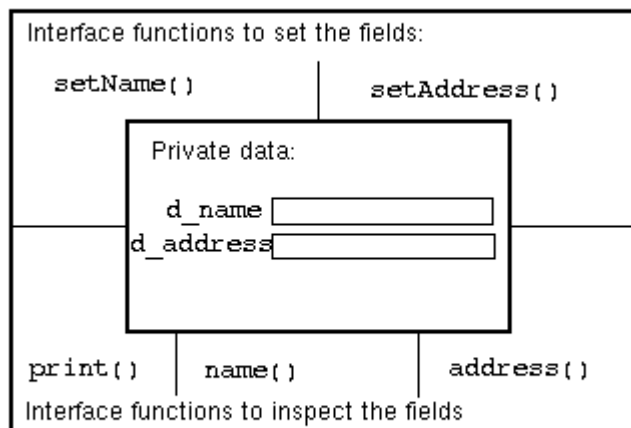


Figura 2 (Funções interface e dados private e public da classe `Person`.)

Note também que as funções `setName()` e `setAddress()` estão declaradas como tendo um argumento `char const*`. Isto significa que as funções não alteram as strings que lhes são passadas. No mesmo sentido as funções `name()` e `address()` retornam `char const *`: quem chama não pode modificar as strings apontadas pelos valores de retorno.

Dois exemplos das funções membro da estrutura `Person` são mostrados abaixo:

```
void Person::setName(char const *n)
{
    strncpy(d_name, n, 79);
    d_name[79] = 0;
}

char const *Person::name()
{
    return d_name;
}
```

Em geral, o poder das funções membro e o conceito de emcobrimento de dados repousa no fato de que as funções interface podem realizar tarefas especiais, p. ex., examinar a validade dos dados.

No exemplo acima setName() copia somente até 79 caracteres de seu argumento ao membro de dados name, evitando, assim, ultrapassar a capacidade do buffer.

Outro exemplo do conceito de encobrimento de dados é o seguinte. Como alternativa às funções membro que guardam seus dados em memória (como fazem os exemplos acima), uma biblioteca em tempo de execução poderia ser desenvolvida com funções interface que guardassem seus dados em disco. A conversão de um programa que armazene estruturas Person em memória a um que guarde em arquivo poderia não requerer qualquer modificação nos programas usuários das estruturas Person. Depois da recompilação e linkagem o novo módulo objeto a uma nova biblioteca, o programa usará a nova estrutura Person.

O encobrimento de dados pode ser realizado por estruturas, apesar de que mais frequentemente (quase sempre) são usadas classes. Uma classe faz referência ao mesmo conceito que as estruturas, exceto que uma classe usa acesso privado como padrão, enquanto as estruturas usam acesso público como padrão. A definição de uma classe Person teria exatamente a mesma aparência como mostrado acima, exceto pelo fato de que no lugar da palavra chave struct se usaria class e a cláusula private inicial poderia ser omitida. Nossa sugestão tipográfica para os nomes de classe é o uso de letra maiúscula como primeiro caracter seguido de minúsculas para o resto do nome (p. ex., Person).

3.6: Estruturas em C versus estruturas em C++

A seguir queremos ilustrar a analogia entre C e C++ no concernente às estruturas. Em C é comum definir muitas funções para processar uma estrutura. Um fragmento imaginário de um cabeçalho em C é dado abaixo:

```
// definição de uma estrutura PERSON_
typedef struct
{
    char name[80];
    char address[80];
} PERSON_;

// algumas funções para manipular estruturas PERSON_

// inicia campos com um nome e endereço
void initialize(PERSON_ *p, char const *nm,
               char const *adr);

// imprime informações
void print(PERSON_ const *p);
```

```
// etc..
```

Em C++ as declarações das funções envolvidas estão dentro da definição da estrutura ou classe. O argumento que evidencia a qual estrutura se refere não é necessário.

```
class Person
{
    public:
        void initialize(char const *nm, char const *adr);
        void print();
        // etc..
    private:
        char d_name[80];
        char d_address[80];
};
```

O argumento da estrutura é implícito em C++. Uma chamada a uma função em C como:

```
PERSON_ x;

initialize(&x, "some name", "some address");
```

Em C++ se tornaria:

```
Person x;

x.initialize("some name", "some address");
```

3.7: Namespaces

Imagine um professor de matemática que queira desenvolver um programa interativo de matemática. Para este programa funções como `cos()`, `sin()`, `tan()`, etc. devem aceitar argumentos em graus e não em radianos. Infelizmente os nomes da função `cos()` já está em uso e essa função aceita radianos em seus argumentos antes que graus.

Problemas como estes usualmente são resolvidos definindo-se outro nome, p.ex. `CosGraus()`. A C++ oferece uma solução alternativa: permitindo-nos usar espaços nomeados. Namespaces pode ser considerados como áreas ou regiões no código onde são definidos identificadores que normalmente não entram em conflito com nomes já definidos noutro lugar.

Agora, depois da aplicação do padrão ANSI/ISO em extenso grau nos compiladores recentes, o uso de espaços nomeados está mais difundido que em versões prévias dos compiladores. Isto tem certas conseqüências na configuração de arquivos cabeçalho para classes. Nesta altura das Anotações não podemos discutir este tema em detalhe, mas na seção 6.5.1 a construção de cabeçalhos usando as entidades espaços nomeados é discutida.

3.7.1: Definindo espaços nomeados

Namespaces são definidos com a seguinte sintaxe:

```
namespace identificador
{
    // entidades declaradas ou definidas
    // (região de declaração)
}
```

O identificador usado na definição de um espaço nomeado é o de um identificador padrão C++.

Na região de declaração, introduzida no exemplo de código acima, funções, variáveis, estruturas, classes e mesmo espaços nomeados (aninhados) podem ser definidos ou declarados. Os espaços nomeados não podem ser definidos dentro de um bloco. Assim não é possível definir um espaço nomeado numa função. Contudo é possível definir um espaço nomeado usando múltiplas declarações de espaços nomeados. Os espaços nomeados são chamados 'abertos'. Isto significa que um espaço nomeado CppAnnotations poderia ser definido num arquivo arquivo1.cc e também num arquivo arquivo2.cc. As entidades definidas no espaço nomeado CppAnnotations do arquivo1.cc e arquivo2.cc estão unidas numa região espaço nomeado CppAnnotations. Por exemplo:

```
// no arquivo1.cc
namespace CppAnnotations
{
    double cos(double argInDegrees)
    {
        ...
    }
}

// no arquivo2.cc
namespace CppAnnotations
```

```

{
    double sin(double argInDegrees)
    {
        ...
    }
}

```

Ambas `sin()` e `cos()` são definidas no mesmo espaço nomeado `CppAnnotations`.

Entidades espaço nomeado podem ser definidas fora de seus espaços nomeados. Este tópico será discutido na seção 3.7.4.1.

3.7.1.1: Declarando entidades em espaços nomeados

Além de definições de entidades em espaços nomeados, podemos também declarar entidades em espaços nomeados. Isto permite todas as declarações de um espaço nomeado estarem num arquivo cabeçalho e assim serem incluídas em fontes onde as entidades do espaço nomeado são usadas. Tal cabeçalho poderia conter, p. ex.:

```

namespace CppAnnotations
{
    double cos(double degrees);
    double sin(double degrees);
}

```

3.7.1.2: Um espaço nomeado fechado

Os espaços nomeados podem ser definidos sem nome. Tal espaço nomeado é anônimo e de visibilidade restrita às entidades aí definidas, no arquivo fonte onde o espaço nomeado é definido.

As entidades definidas num espaço nomeado anônimo são comparáveis às variáveis e funções estáticas da linguagem C. Em C++ a palavra chave `static` ainda pode ser usada, mas seu uso é mais comum em definições de classes (veja o capítulo 6). Nos casos onde variáveis ou funções são necessárias o uso de espaço nomeado anônimo é preferível.

O espaço nomeado anônimo é um espaço nomeado fechado: não é possível adicionar entidades ao mesmo espaço nomeado anônimo em diferentes arquivos fonte.

3.7.2: Referenciando entidades

Dado um espaço nomeado e entidades definidas ou declaradas nele, o operador de resolução de escopo pode ser usado para referenciar as entidades definidas no espaço nomeado. Por exemplo, para usar a função `cos()` definida no espaço nomeado `CppAnnotations` pode-se usar o seguinte código:

```
// assume-se que o espaço nomeado CppAnnotations está declarado no
// arquivo cabeçalho seguinte:
#include <CppAnnotations>

int main()
{
    cout << "O coseno de 60 graus é: " <<
        CppAnnotations::cos(60) << endl;
}
```

Este é um modo incômodo de referência a função `cos()` no espaço nomeado `CppAnnotations`, especialmente se a função for usada com frequência.

Nestes casos uma forma abreviada (somente `cos()`) pode ser usada especificando-a numa declaração `using`. Como segue:

```
using CppAnnotations::cos; // note: sem protótipo da função,
                           // somente o nome da entidade
                           // é requerido.
```

`Cos()` referenciará a função `cos()` no espaço nomeado `CppAnnotations`. Isto implica que a função padrão `cos()`, aceitando radianos, não poderá mais ser usada automaticamente. O operador de resolução de escopo pleno deve ser usado para acessar a função genérica `cos()`:

```
int main()
{
    using CppAnnotations::cos;
    ...
    cout << cos(60)           // usa CppAnnotations::cos()
        << ::cos(1.5)        // usa a função padrão cos()
        << endl;
}
```

Note-se que uma declaração `using` pode ser posta dentro de um bloco. A declaração `using` fará com que a referência ao nome que esteja em `using` seja o acessado ao usá-lo: não é possível colocar uma declaração `using` para uma variável no espaço nomeado `CppAnnotations` e definir ou declarar um objeto com nome idêntico no bloco onde a declaração `using` foi posta:

```
int main()
{
```

```

    using CppAnnotations::value;
    ...
    cout << value << endl; // usa CppAnnotations::value

    int value;              // erro: value já definida.
}

```

3.7.2.1: A diretiva *'using'*

Uma alternativa generalizada à declaração using é a diretiva using:

```
using namespace CppAnnotations;
```

Seguindo esta diretiva todas as entidades no espaço nomeado CppAnnotations são usadas como se cada uma tivesse sido declarada por uma declaração using.

Ao mesmo tempo que a diretiva using é um meio rápido de importar todos os nomes do espaço nomeado CppAnnotations (assumindo que as entidades foram declaradas ou definidas separadamente da diretiva), é um meio algo enganoso de se fazer, já que não fica claro quais entidades serão usadas no bloco de código.

Se, p. ex., `cos()` é definido no espaço nomeado CppAnnotations a função `CppAnnotations::cos()` será usada quando `cos()` for chamada no código. Contudo, se `cos()` não estiver no espaço nomeado CppAnnotations, a função padrão será usada. A diretiva using não documenta claramente qual entidade será usada como a declaração using o faz. Por esta razão a diretiva using é algo desprezada.

3.7.2.2: O *'Koenig lookup'*

Se o 'Koenig lookup' fosse chamado de 'princípio Koenig', poderia ter sido o título de um novo Ludlum novell. Contudo não o é. Em lugar disso se refere a uma técnica C++.

O 'Koenig lookup' é relativo ao fato de que se uma função é chamada sem referenciar a um espaço nomeado então os espaços nomeados de seus argumentos são usados para encontrar o espaço nomeado da função. Se o espaço nomeado onde estão os argumentos contém tal função então essa função é usada. A esta técnica chamamos 'Koenig lookup'.

No seguinte exemplo isto é ilustrado. A função `FBB::fun(FBB::value v)` é definida no espaço

nomeado FBB. Como é mostrado ela pode ser chamada sem a menção explícita a um espaço nomeado:

```
#include <iostream>

namespace FBB
{
    enum Value          // define FBB::Value
    {
        first,
        second,
    };

    void fun(Value x)
    {
        std::cout << "fun chamada com " << x << std::endl;
    }
}

int main()
{
    fun(FBB::first);    // Koenig lookup: sem espaço nomeado
                       // para fun()
}
/*
saída gerada:
fun chamada com 0
*/
```

Note que o intento de enganar o compilador não funciona: se no espaço nomeado FBB Value fosse definido como `typedef int Value`, então `FBB::Value` seria reconhecido como `int`, isto faria o Koenig lookup falhar.

Como outro exemplo considere o seguinte programa. Aqui há dois espaços nomeados envolvidos, cada um definindo sua própria função `fun()`. Não há ambigüidade aqui, já que o argumento define o espaço nomeado. Portanto `FBB::fun()` é chamada:

```
#include <iostream>

namespace FBB
{
    enum Value          // define FBB::Value
    {
        first,
        second,
    };
}
```

```

};

void fun(Value x)
{
    std::cout << "FBB::fun() chamada com " << x << std::endl;
}

namespace ES
{
    void fun(FBB::Value x)
    {
        std::cout << "ES::fun() chamada com " << x << std::endl;
    }
}

int main()
{
    fun(FBB::first);    // Não há ambigüidade: o argumento determina
                        // o espaço nomeado
}
/*
    Saída gerada:
FBB::fun() chamada com 0
*/

```

Finalmente um exemplo onde há uma ambigüidade: fun() tem dois argumentos, um de cada espaço nomeado individual. Aqui a ambigüidade deve ser resolvida pelo programador:

```

#include <iostream>

espaço nomeado ES
{
    enum Value        // define ES::Value
    {
        first,
        second,
    };
}

espaço nomeado FBB
{
    enum Value        // define FBB::Value
    {
        first,
        second,
    };
}

```



```

        void fun(Value x, ES::Value y)
        {
            std::cout << "FBB::fun() chamada\n";
        }
    }

    espaço nomeado ES
    {
        void fun(FBB::Value x, Value y)
        {
            std::cout << "ES::fun() chamada\n";
        }
    }

    int main()
    {
        /*
                                fun(FBB::first, ES::first); // ambigüidade: deve ser
resolvida                                // mencionando explicitamente
                                           // o espaço nomeado

        */
        ES::fun(FBB::first, ES::first);
    }
    /*
        saída gerada:
        ES::fun() chamada
    */

```

3.7.3: O espaço nomeado padrão

Muitas entidades de software disponível em tempo de execução (p. ex. `cout`, `cin`, `cerr` e modelos definidos na biblioteca Standard Template Library, veja o capítulo 17) agora estão definidas no espaço nomeado `std`.

Observando a discussão da seção anterior, poder-se-ia usar a declaração `using` para estas entidades. Por exemplo, para se usar a stream `cout` o código deveria começar com algo assim:

```

#include <iostream>
using std::cout;

```

Costumeiramente, contudo, os identificadores definidos no espaço nomeado `std` podem ser aceitos sem muita conversa. Assim, frequentemente encontramos uma diretiva `using`, no lugar de uma

declaração `using`, com o espaço nomeado `std`. Portanto no lugar da declaração `using` mencionada encontramos uma construção como:

```
#include <iostream>
using namespace std;
```

Se devemos ou não encorajar o uso desta forma é tema para disputa. Longas declarações `using`, claro, é inconveniente também. Portanto, como regra prática, devemos estender a lista de declarações até o ponto em que se torne inconvenientemente longa a partir daí a diretiva `using` se torna considerável.

3.7.4: Aninhando espaços nomeados e apelidos de espaços nomeados

Os espaços nomeados podem ser aninhados. O código seguinte mostra a definição de espaço nomeado aninhado:

```
namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;
    }
}
```

Relativamente à variável ponteiro definida no espaço nomeado `Virtual`, aninhada sob o espaço nomeado `CppAnnotations`, para nos referirmos a esta variável existem as seguintes opções:

- O nome por extenso pode ser usado. Um nome completamente qualificado de uma entidade é uma lista de todos os espaços nomeados que são visitados até chegarmos à definição da entidade, unidos pelo operador de resolução de escopo:

```
int main()
{
    CppAnnotations::Virtual::pointer = 0;
}
```

- Uma declaração `using` para `CppAnnotations::Virtual` pode ser usada. Assim `Virtual` pode ser usado sem qualquer prefixo, mas o ponteiro tem que ser usado com o prefixo `Virtual::`:

```
...
using CppAnnotations::Virtual;

int main()
```

```
{
    Virtual::pointer = 0;
}
```

- Uma declaração using para CppAnnotations::Virtual::pointer pode ser usada. Agora o ponteiro pode ser usado sem prefixo:

```
...
using CppAnnotations::Virtual::pointer;

int main()
{
    pointer = 0;
}
```

- Uma diretiva using ou diretivas podem ser usadas:

```
...
using namespace CppAnnotations::Virtual;

int main()
{
    pointer = 0;
}
```

Alternativamente duas diretivas using separadas podem ser usadas:

```
...
using namespace CppAnnotations;
using namespace Virtual;

int main()
{
    pointer = 0;
}
```

- Uma combinação de declarações using e diretivas using pode ser usada. P.ex. Uma diretiva using para o espaço nomeado CppAnnotations e uma declaração using para a variável Virtual::pointer:

```
...
using namespace CppAnnotations;
using Virtual::pointer;

int main()
{
    pointer = 0;
}
```

Todas as entidades da diretiva using desse espaço nomeado podem ser usadas sem qualquer

prefixo. Se um espaço nomeado está aninhado então também pode ser usado sem prefixo. Contudo as entidades definidas no espaço nomeado aninhado ainda necessitarão do nome do espaço nomeado aninhado. Somente usando uma declaração ou diretiva using a qualificação do nome do espaço nomeado aninhado pode ser omitida.

Quando o nome pleno parece ser preferível e uma forma muito longa como:

```
CppAnnotations::Virtual::pointer
```

é ao mesmo tempo considerada muito longa, pode-se definir um apelido do espaço nomeado:

```
namespace CV = CppAnnotations::Virtual;
```

Assim se define CV como apelido do nome completo. Portanto para se referir à variável pointer usa-se a construção:

```
CV::pointer = 0;
```

Claro que um apelido do espaço nomeado também pode ser usado numa declaração ou diretiva using.

3.7.4.1: Definindo entidades fora de seus espaços nomeados

Não é estritamente necessário definir membros de espaços nomeados dentro de uma região do espaço nomeado. Prefixando o membro com seu espaço nomeado ou espaços nomeados um membro pode ser definido fora da região do espaço nomeado. Isto pode ser feito a nível global ou em níveis intermediários, no caso de espaços nomeados aninhados. Assim, se não for possível definir um membro A dentro da região do espaço nomeado C, é possível definir um membro do espaço nomeado A::B dentro da região do espaço nomeado A.

Note, contudo, que quando um membro de um espaço nomeado é definido fora da região do espaço nomeado ainda assim precisa estar declarado nessa região.

Imaginemos que o tipo `int8[8]` está definido no `CppAnnotations::Virtual`.

Suponha agora que queremos definir um membro da função `funny`, dentro do espaço nomeado `CppAnnotations::Virtual`, que retorna um apontador a `CppAnnotations::Virtual::INT8`. Definindo tudo dentro do espaço nomeado `CppAnnotations::Virtual` tal função pode ser definida como segue:

```

namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;

        typedef int INT8[8];

        INT8 *funny()
        {
            INT8 *ip = new INT8[1];

            for (int idx = 0; idx < sizeof(INT8) / sizeof(int);
++idx)
                (*ip)[idx] = (idx + 1) * (idx + 1);

            return ip;
        }
    }
}

```

A função funny() define um conjunto de um vetor INT8 e retorna seu endereço depois de iniciar o vetor com o quadrado dos primeiros oito números naturais.

Agora a função funny() pode ser definida fora do espaço nomeado CppAnnotations::Virtual como segue:

```

namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;

        typedef int INT8[8];

        INT8 *funny();
    }
}

CppAnnotations::Virtual::INT8 *CppAnnotations::Virtual::funny()
{
    INT8 *ip = new INT8[1];

    for (int idx = 0; idx < sizeof(INT8) / sizeof(int); ++idx)
        (*ip)[idx] = (idx + 1) * (idx + 1);

    return ip;
}

```

```
}
```

No final do fragmento de código note o seguinte:

- funny() esta declarada dentro do espaço nomeado CppAnnotations::Virtual

A definição exterior da região do espaço nomeado requer o uso do nome completo da função e o tipo de seu retorno.

- Dentro do bloco da função funny() estamos com o nome do espaço nomeado CppAnnotations::Virtual, portanto dentro da função como temos os nomes completos (p.ex. Para INT8) já não são requeridos.

Finalmente note que a função poderia ter sido definida também na região de CppAnnotations. Nesse caso teria sido necessário o espaço nomeado Virtual para o nome da função e seu tipo de retorno, enquanto o interior da função seria o mesmo:

```
namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;

        typedef int INT8[8];

        INT8 *funny();
    }

    Virtual::INT8 *Virtual::funny()
    {
        INT8 *ip = new INT8[1];

        for (int idx = 0; idx < sizeof(INT8) / sizeof(int); ++idx)
            (*ip)[idx] = (idx + 1) * (idx + 1);

        return ip;
    }
}
```

Capítulo 4: O tipo de dados `string`

A linguagem C++ oferece um grande número de recursos para chegarmos à solução de problemas comuns. A maioria desses recursos são parte da biblioteca Standard Template Library ou estão implementados como algoritmos genéricos (veja o capítulo 17).

Entre as facilidades da C++ que os programadores desenvolveram uma e outra vez estão poderosos meios de manipulação de texto, comumente chamados strings. A linguagem de programação C oferece um suporte rudimentar às strings : o ASCII-Z séries terminadas de caracteres é a fundação sobre a qual montanhas de código foram desenvolvidas (definimos uma string ASCII-Z como uma seqüência de caracteres ASCII terminada pelo caracter ASCII zero (a partir daqui -Z), que tem o valor zero e não pode ser confundido com '0', que usualmente tem o valor 0x30)

A C++ agora oferece o tipo padrão string. Para se poder usar objetos tipo string o cabeçalho string precisa ser incluído na fonte.

Hoje em dia os objetos string são variáveis de uma classe e essa classe está formalmente introduzida no capítulo 6. Contudo, para se usar uma string não é necessário saber o que é uma classe. Nesta seção os operadores disponíveis para o uso com strings e diversas outras operações são discutidos. As operações fatíveis de se realizar com strings têm a forma:

```
stringVariable.operation(argumentList)
```

Por exemplo, se a string1 e a string2 são variáveis do tipo string, então:

```
string1.compare(string2)
```

Pode ser usada para comparar ambas strings. Uma função como compare(), que é parte da classe string é chamada função membro. A classe string oferece um longo número dessas funções membro bem como extensões de alguns operadores bem conhecidos, como adjudicação (=). Estes operadores e funções são discutidos nas próximas seções.

4.1: Operações com strings

Algumas das operações possíveis de se realizar com strings retornam índices com as strings. Sempre que a operação falha em encontrar um índice apropriado, o valor string::npos é retornado. Este valor (simbólico) é do tipo string::size_type, que é (para todos os fins práticos) um int (sem sinal).

Note que em todas as operações com strings ambas string objeto char const * e variáveis podem ser usadas.

Algumas strings membro usam iteradores. Os iteradores serão vistos na seção 17.2. As funções membro usando iteradores estão listadas na seção seguinte (4.2), não estão ilustradas abaixo.

Pode-se realizar as seguintes operações com strings:

- **Iniciação:** As strings objeto podem ser iniciadas. Para iniciar uma string plena-z, outra string objeto, ou uma iniciação implícita podem ser usadas. No exemplo note que a iniciação implícita não leva argumento, e não pode usar uma lista de argumentos. Nem mesmo vazia.

```
#include <string>

using namespace std;

int main()
{
    string stringOne("Olá Mundo"); // usando plena ascii-Z
    string stringTwo(stringOne);    // usando outra string objeto
    string stringThree;             // iniciação implícita com "".
                                    // não usa a forma
`stringThree() '
    return 0;
}
```

- **Adjudicação:** As strings objeto podem ser adjudicadas umas às outras. Para isso o operador de adjudicação (i.e., o operador '=') é usado, o qual aceita uma string objeto e uma cadeia de caracteres estilo C como seu argumento à direita:

```
#include <string>
using namespace std;

int main()
{
    string stringOne("Olá Mundo");
    string stringTwo;

    stringTwo = stringOne; // adjudica stringOne à stringTwo
    stringTwo = "Hello world"; // adjudica uma string C à StringTwo

    return 0;
}
```

- **Conversão de uma string C:** No exemplo anterior uma string padrão C (uma string ASCII-Z) foi implicitamente convertida a uma string objeto. A conversão reversa (conversão de uma string objeto a uma string C) não foi feita automaticamente. Para se obter a string C que será guardada na string objeto pode-se usar a função membro `c_str()`, que retorna `char const *`:


```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Olá Mundo");
    char const *cString = stringOne.c_str();

    cout << cString << endl;

    return 0;
}

```

- Elementos de uma string: os elementos individuais de uma string podem ser acessados para leitura ou escrita. Para esta operação se dispõe do operador subscrive ([]), mas não há apontador de referência (*). O operador de subscrição não realiza exame de tamanho. Se for necessário um exame de tamanho usa-se a função membro string::at():

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Olá Mundo");

    stringOne[4] = 'm';           // agora "Olá mundo"
    if (stringOne[0] == 'O')
        stringOne[0] = 'o';      // agora "olá mundo"

    // *stringOne = 'H';          // NÃO COMPILA

    stringOne = "Hello World";    // agora usando a

                                   // função membro at():
    stringOne.at(6) =
        stringOne.at(H);          // agora "Hello Horld"
    if (stringOne.at(0) == 'H')
        stringOne.at(0) = 'W';    // agora "Wello Horld"

    return 0;
}

```

Quando um índice ilegal é passado à at() função membro o programa aborta (atualmente é gerada uma exceção, que pode ser apanhada. As exceções são vistas no capítulo 8).

- **Comparações:** pode-se comparar duas strings para determinar igualdade ou ordem, usando-se os operadores `==`, `!=`, `<`, `<=`, `>` e `>=` ou a função membro `string::compare()`. A função membro `compare()` vem em distintas variedades (veja a seção 4.2.4 para detalhes). Por exemplo:
 - `int string::compare(string const &other)`: Esta variante oferece um pouco mais de informação que os operadores de comparação. o valor de retorno da função membro `string::compare()` pode ser usado para ordenar lexicamente: um valor negativo se a string objeto que usa `compare()` (no exemplo: `stringOne`) está colocada antes na sequência ASCII que a string argumento.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Olá Mundo");
    string stringTwo;

    if (stringOne != stringTwo)
        stringTwo = stringOne;

    if (stringOne == stringTwo)
        stringTwo = "Algo mais";

    if (stringOne.compare(stringTwo) > 0)
        cout << "stringOne depois de stringTwo no alfabeto\n";
    else if (stringOne.compare(stringTwo) < 0)
        cout << "stringOne antes da stringTwo no alfabeto\n";
    else
        cout << "Ambas strings são iguais\n";

    // Alternativamente:

    if (stringOne > stringTwo)
        cout <<
            "stringOne depois de stringTwo no alfabeto\n";
    else if (stringOne < stringTwo)
        cout <<
            "stringOne antes da stringTwo no alfabeto\n";
    else
        cout << "Ambas strings são iguais\n";

    return 0;
}
```

Note que não há função membro para realizar uma comparação insensível aos casos.

- `int string::compare(string::size_type pos, size_t n, string const &other)`: o primeiro argumento indica a posição na string corrente que deve ser comparada; o segundo argumento indica o número de caracteres a serem comparados (se esse número excede o número de caracteres disponíveis a partir dessa posição, somente os disponíveis são comparados); o terceiro argumento indica a string que será comparada à corrente.
- Mais variantes de `string::compare()` estão disponíveis na seção 4.2.4 em detalhe.

O exemplo seguinte ilustra a função `compare()`:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Hello World");

    // comparando a partir de uma posição de stringOne
    if (!stringOne.compare(1, stringOne.length() - 1, "ello World"))
        cout << "comparando 'Hello world' do índice 1"
              << " com 'ello World': ok\n";

    // o número de caracteres a comparar (2o argumento)
    // pode exceder o número de caracteres disponíveis:
    if (!stringOne.compare(1, string::npos, "ello World"))
        cout << "comparando 'Hello world' do índice 1"
              << " a 'ello World': ok\n";

    // comparando a partir de certo caracter de stringOne com
    // certo número de caracteres em "World and more"
    // Esta fallha, já que todos os caracteres da stringOne
    // a partir do índice 6 são comparados, não apenas
    // 3 caracteres em "World and more"
    if (!stringOne.compare(6, 3, "World and more"))
        cout <<
            "comparando 'Hello World' a partir do índice 6 para cima"
            " 3 posições de 'World and more': ok\n";
    else
        cout << "(sub)strings desiguais\n";

    // Esta reportará uma coincidência, já que 5 caracteres são
    // comparados entre as duas strings
```

```

    if (!stringOne.compare(6, 5, "World and more", 0, 5))
        cout <<
            "comparando 'Hello World' a partir do índice 6"
            " em 5 posições com 'World and more': ok\n";
    else
        cout << "(sub)strings desiguais \n";
}
/*
    Saídas Geradas:

    comparando 'Hello world' do índice 1 com 'ello World': ok
    comparando 'Hello world' do índice 1 com 'ello World': ok
    (sub)strings desiguais
    comparando 'Hello World' do índice 6 em 5 posições com
        'World and more': ok
*/

```

- **Apendiculação (appending):** Uma string pode ser apendiculada a outra. Para isto o operador += pode ser usado, bem como a função membro string &string::append().

Como a função compare(), a função membro append() pode ter argumentos extra. O primeiro argumento é a string a ser apendiculada, o segundo especifica o índice do primeiro carácter a ser apendiculado. O terceiro argumento especifica o número de caracteres a serem apendiculados. Se o primeiro argumento for do tipo char const *, somente um segundo argumento pode ser especificado. Neste caso o segundo argumento especifica o número de caracteres que serão apendiculados à string objeto. Ainda mais o operador + pode ser usado para apendicular duas strings numa expressão:

```

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string stringOne("Hello");
    string stringTwo("World");

    stringOne += " " + stringTwo;

    stringOne = "hello";
    stringOne.append(" world");

                                // apendiculados 5 caracteres:
    stringOne.append(" ok. >Isto não é usado<", 5);

    cout << stringOne << endl;
}

```

```

    string stringThree("Hello");
                                // apendiculado " world":
    stringThree.append(stringOne, 5, 6);

    cout << stringThree << endl;
}

```

O operador + é usado em casos onde pelo menos um termo do operador + é uma string objeto (o outro termo pode ser string, char const * ou char).

No caso de nenhum operando de + ser uma string objeto, pelo menos um operando precisa ser convertido a string objeto antes. Uma maneira fácil de se fazer isto é usar uma string objeto anônima:

```

    string("hello") + " world";

```

- Inserção: A função membro string &string::insert() insere (partes de) uma string e possui pelo menos dois argumentos e quando muito quatro:
- O primeiro argumento é um deslocamento dentro da string objeto corrente onde outra string será inserida.
- O segundo argumento é a string a ser inserida.
- O terceiro argumento especifica o índice da posição de inserção na string objeto onde será inserida a outra string.
- O quarto argumento especifica o número de caracteres a serem inseridos.

Se o primeiro argumento é do tipo char const *, o quarto argumento não está disponível. Nesse caso o terceiro argumento indica o número de caracteres que serão inseridos da string char const *:

```

int main()
{
    string
        stringOne("Hell ok.");
                                // Insere "o " na posição 4
    stringOne.insert(4, "o ");

    string
        world("The World of C++");

                                // insere "World" em stringOne
    stringOne.insert(6, world, 4, 5);

    cout << "Adivinhe o quê? É: " << stringOne << endl;
}

```

```
}
```

Diversas variantes de `string::insert()` estão disponíveis. Veja a seção 4.2 para maiores detalhes.

- Substituição: Por vezes necessita-se substituir o conteúdo de uma string objeto por outra informação. Para substituir partes do conteúdo de uma string por outra a função membro `string &string::replace()` é usada.

A função membro tem pelo menos três e possivelmente cinco argumentos com os seguintes significados (veja a seção 4.2 para versões sobrecarregadas de `replace()`, usando diferentes tipos de argumentos):

- O primeiro argumento indica a posição do primeiro caracter a ser substituído.
- O segundo argumento dá o número de caracteres a serem substituídos.
- O terceiro argumento define o texto de substituição (uma string ou `char const *`).
- O quarto argumento especifica a posição do índice do primeiro caracter que será inserido da string fornecida.
- O quinto argumento é usado para especificar o número de caracteres que serão substituídos.

Se o terceiro argumento é do tipo `char const *`, então o quinto argumento não é válido. Nesse caso o quarto argumento indica o número de caracteres do `char const *` que serão inseridos.

O seguinte exemplo mostra um simples transformador de arquivos e substitui as ocorrências de 'searchstring' por 'replacestring'. São feitos exames simples para determinar o número correto de argumentos e o conteúdo das strings dadas (podem ser diferentes) são construídas usando a macro `assert()`.

```
#include <iostream>
#include <string>
#include <cassert>

using namespace std;

int main(int argc, char **argv)
{
    assert(argc == 3 &&
           "Usage: <searchstring> <replacestring> to process stdin");
```

```

    string line;
    string search(argv[1]);
    string replace(argv[2]);

    assert(search != replace);

    while (getline(cin, line))
    {
        string::size_type idx = 0;
        while (true)
        {
            idx = line.find(search, idx); // find(): outro membro de

            // string                                     // veja `searching' abaixo

            if (idx == string::npos)
                break;

            line.replace(idx, search.size(), replace);
            idx += replace.length(); // não muda a substituição
        }
        cout << line << endl;
    }
    return 0;
}

```

- Troca (swapping): A função membro `string &string::swap(string &outra)` troca o conteúdo de duas strings objeto. Por exemplo:

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Hello");
    string stringTwo("World");

    cout << "Antes: stringOne: " << stringOne << ", stringTwo: "
         << stringTwo << endl;

    stringOne.swap(stringTwo);

    cout << "Depois: stringOne: " << stringOne << ", stringTwo: "
         << stringTwo << endl;
}

```

- Apaga (erasing): A função membro `string &string::erase()` remove caracteres da string. A forma

padrão tem dois argumentos opcionais:

- Sem argumentos a string será apagada completamente; se torna uma string vazia (string() ou string("")).
- O primeiro argumento especifica a posição a partir da qual se apagará os caracteres.
- O segundo argumento especifica o número de caracteres que serão apagados.

Veja a seção 4.2 para as versões de erase() sobrecarregadas. um exemplo do uso de erase() está dado abaixo:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Olá Mundo Cruel");

    stringOne.erase(5, 6);

    cout << stringOne << endl;

    stringOne.erase();

    cout << "'" << stringOne << "'\n";
}
```

- Busca: Para encontrar subcadeias numa string usa-se a função membro string::size_type string::find(). Esta função procura pela string dada como seu primeiro argumento na string objeto e retorna o índice do primeiro caracter da subcadeia se a encontrar. Se a subcadeia não é encontrada string::npos é retornado. A função membro rfind() procura pela subcadeia a partir do fim da string objeto até o início. Um exemplo usando find() foi dado anteriormente.
- Subcadeias: Para extrair uma subcadeia de uma string objeto dispomos da função string string::substr(). A string objeto retornada contém uma cópia da subcadeia da string objeto que chamou substr(). A função membro substr() tem dois argumentos opcionais:
- Sem argumentos retorna uma cópia da própria string.
- O primeiro argumento é usado para especificar o deslocamento do primeiro caracter a ser retornado.

- O segundo argumento é usado para especificar o número de caracteres a serem retornados.

Por exemplo:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Hello World");

    cout << stringOne.substr(0, 5) << endl
         << stringOne.substr(6) << endl
         << stringOne.substr() << endl;
}
```

- Busca de conjunto de caracteres: Como find() é usada para buscar uma subcadeia, as funções find_first(), find_first_not_of(), find_last_of() e find_last_not_of() são usadas para buscar conjuntos de caracteres (Infelizmente aqui não há suporte para expressões regulares). O seguinte programa lê uma linha de texto da entrada padrão stream e mostra a subcadeia que começa pela primeira vogal, começando pela última vogal e começando pelo primeiro não dígito:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;

    getline(cin, line);

    string::size_type pos;

    cout << "Line: " << line << endl
         << "Starting at the first vowel:\n"
         << "' ' "
         << (
             (pos = line.find_first_of("aeiouAEIOU"))
             != string::npos ?
             line.substr(pos)
             :
             "*** not found ***"
         ) << "'\n"
         << "Starting at the last vowel:\n"
         << "' ' "
```

```

        << (
            (pos = line.find_last_of("aeiouAEIOU"))
            != string::npos ?
                line.substr(pos)
            :
                "*** not found ***"
        ) << "'\n"
    << "Starting at the first non-digit:\n"
    << "'"
        << (
            (pos = line.find_first_not_of("1234567890"))
            != string::npos ?
                line.substr(pos)
            :
                "*** not found ***"
        ) << "'\n";
    }

```

- **Tamanho da string:** O número de caracteres guardados na string é obtido pela função membro `size()` que igual que a função padrão da C `strlen()` não inclui o caracter de terminação da ASCII-Z. Por exemplo:

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne("Hello World");

    cout << "O tamanho da string stringOne é "
         << stringOne.size() << " caracteres\n";
}

```

- **Strings Vazias:** A função membro `size()` é usada para determinar se uma string não guarda nenhum caracter. Alternativamente pode-se usar a função membro `string::empty()`:

```

#include <iostream>
#include <string>
using namespace std;

int main()
{
    string stringOne;

    cout << "O tamanho da string stringOne é "
         << stringOne.size() << " caracteres\n"
         << "É" << (stringOne.empty() ? "" : " não ");
}

```

```

        << "vazia\n";

stringOne = "";

cout << "Depois de por um \"\"-string na string-objeto\n"
      "é " << (stringOne.empty() ? "também" : " não")
      << " vazia\n";
}

```

- Redimensionando strings: Se o tamanho de uma string não é suficiente (ou se é muito grande) a função membro void `string::resize()` é usada para fazê-la maior ou menor. Note que operadores como `+=` redimensionam automaticamente a string quando necessário.
- Lendo uma string de uma stream: A função `istream &getline(istream &istream, string &alvo, char delimitador)` é usada para ler uma linha de texto (até o primeiro delimitador ou o fim da stream) de `istream` (note que `getline()` não é uma função membro da classe `string`).

O delimitador tem o valor padrão `'\n'`. É removido da `istream` mas não é guardado na string alvo. A `istream::fail()` determina se o delimitador foi encontrado. Se retornar verdadeiro, o delimitador não foi encontrado (veja capítulo 5 para detalhes sobre objetos `istream`). A função `getline()` foi usada em diversos exemplos anteriores (p. ex. com a função membro `replace()`).

- Uma variável string pode ser extraída de uma stream usando-se a construção:

```
istr >> str;
```

onde `istr` é um objeto stream e `str` é uma string a próxima série consecutiva de caracteres não espaço em branco será adjuntada a `str`. Note que por padrão a operação de extração saltará todo espaço em branco extraído da stream.

4.2: Visão Geral das Operações com Cadeias de Caracteres

Nesta seção as operações disponíveis com strings estão sumarizadas. Existem quatro subpartes aqui: iniciadores de strings, iteradores de strings, operadores com strings e funções com strings membro.

As funções membro estão ordenadas alfabeticamente pelo nome da operação. Abaixo objeto é uma string objeto e argumento ou é `string const &` ou `char const *`, a menos de versões sobrecarregadas para parâmetros `string` e `char const *` que explicitamente estão mencionadas. Nos casos onde uma string objeto é iniciada ou dado um novo valor é usado o termo objeto. A entidade referida como argumento

sempre fica imutável.

Ainda mais, `opos` indica um deslocamento na string objeto, `apos` indica um deslocamento na cadeia argumento. Analogamente `on` indica um número de caracteres na string objeto e `an` indica um número de caracteres na argumento. Ambos `opos` e `apos` necessariamente se referem a deslocamentos existentes, do contrário será gerada uma exceção. Em contraste `an` e `on` podem exceder o número de caracteres disponíveis, no caso, somente os caracteres disponíveis serão considerados.

Quando estão envolvidas streams, `istr` indica uma stream de onde serão extraídas informações e `ostr` indica uma stream na qual serão inseridos dados.

Nas funções membro os tipos de parâmetros são dados no protótipo da função. Em muitas funções membro se usam iteradores. Neste ponto, aqui é um pouco prematuro discutir iteradores, mas como referência devemos mencioná-los. Assim, veja a seção 17.2 para uma discussão mais detalhada de iteradores. Como `apos` e `opos`, os iteradores se referem a um caracter existente ou a um nível de iteração possível para a string a que se refere.

Finalmente note que toda função membro de `string` que retornam índice, retorna a constante predefinida `string::npos` se um índice adequado pode ser encontrado.

4.2.1: Iniciadores

Os seguintes construtores de strings são fornecidos:

- `string objeto`: Inicia o objeto como uma string vazia.
- `string objeto(string::size_type no, char c)`: Inicia o objeto com o caracter `c`.
- `string objeto(string argumento)`: Inicia o objeto com o argumento.
- `string objeto = argumento`: Inicia o objeto com argumento. Esta é uma forma alternativa da iniciação anterior.
- `string objeto(string argumento, string::size_type apos, string::size_type an = apos)`: Inicia o objeto com argumento, usando o caracter do argumento, começando do índice `apos`.
- `string objeto(InputIterator begin, InputIterator end)`: Inicia o objeto com os caracteres que implicam o iterador `InputIterator`. Os iteradores são especificados na seção 17.2, mas (por agora)

podem ser interpretados como apontadores a caracteres. Veja também a próxima seção.

4.2.2: Iteradores

Veja a seção 17.2 para os detalhes sobre iteradores. Uma introdução rápida a iteradores: um iterador atua como um apontador e os apontadores podem ser usados em situações onde os iteradores são requeridos. Os iteradores quase sempre vêm em pares: o iterador inicial, que aponta a primeira entidade a ser considerada, e o iterador final que aponta justo depois da última entidade a ser considerada. Os iteradores jogam um importante papel no contexto de algoritmos genéricos (capítulo 17).

- Iteradores dianteiros são retornados pelos membros:
- `string::begin()`, apontando o primeiro caracter da string objeto.
- `string::end()`, apontando o primeiro caracter depois do último caracter da string objeto.
- Iteradores reversos: são usados para dar um passo na direção reversa. Os iteradores reversos são retornados pelos membros:
- `string::rbegin()`, que pode ser considerada um apontador iterador ao último caracter da string objeto.
- `string::rend()`, que pode ser considerada um apontador iterador que aponta para o endereço anterior ao primeiro caracter da string.

4.2.3: Operadores

As seguintes strings operadores estão disponíveis:

- `objeto = argumento` Adjudicação da argumento a uma string existente.
- `objeto = c` Adjudicação do caracter `c` ao objeto.
- `objeto += argumento` Apendiculação ao objeto. O argumento também pode ser uma expressão `char`.
- `argumento1 + argumento2` Em expressões as strings podem ser somadas. Pelo menos um termo da expressão (termo esquerdo ou direito) deve ser uma string objeto. O outro pode ser `string`, `char`, `const *` ou uma expressão `char`, como ilustra o exemplo seguinte:

```

void fun()
{
    char const *asciiz = "hello";
    string first = "first";
    string second;

    // todas as expressões compilam ok:
    second = first + asciiz;
    second = asciiz + first;
    second = first + 'a';
    second = 'a' + first;
}

```

objeto [string_size_type opos] O operador subscrito é usado para retirar do objeto caracteres individuais ou adjudicar novos valores a caracteres individuais do objeto. Não é efetuado exame de extensão. Se for necessário exame de extensão deve-se usar a função membro `at()`.

- `argumento1 == argumento2` O operador igualdade é usado para comparar a string objeto a outra string ou `char const *`. O operador desigualdade `!=` também pode ser usado. O valor de retorno de ambos é `bool`. Para duas strings idênticas `==` retorna verdadeiro e `!=` retorna falso.
- `argumento1 < argumento2` O operador menor que é usado para comparar a ordem ASCII do `argumento1` e `argumento2`. Os operadores `<=`, `>` e `>=` estão disponíveis também.
- `ostr << objeto` O operador inserção é usado com strings objeto.
- `istr >> objeto` O operador extração é usado com strings objeto. Opera analogamente à extração de caracteres de um vetor (array), mas redimensiona automaticamente o número de caracteres.

4.2.4: Funções Membro

As funções membro string abaixo estão listadas em ordem alfabética. O nome do membro, prefixado pela classe `string` é dado primeiro. Então o protótipo completo e uma descrição são dados. Os valores do tipo `'string::size_type'` representam posições dos índices numa string. Para todos os propósitos práticos, esses valores podem ser interpretados como `'unsigned'`.

O valor especial `'string::npos'`, definido pela classe `string`, representa um índice não-existente. Este valor é retornado por todos os membros que retornam índices quando não podem realizar suas tarefas requisitadas. Note que o comprimento da string não é retornado como um índice válido. P.ex., quando chamando um membro `'find_first_not_of(" ")'` (veja abaixo) numa string objeto tendo 10 espaços em

branco, 'npos' é retornado, já que a string só contém espaços. O byte-0 final que é usado em C para indicar o fim de uma string ASCII-Z não é considerado parte da string C++, e portanto a função membro retornará 'npos', antes que 'length()'.

Na seguinte revisão geral, 'size_type' deve sempre ser lido como 'string::size_type'.

- `char &string::at(size_type opos):`

o carácter (referência) da posição indicada é retornado (pode ser re-designado). A função membro realiza um exame de tamanho, provocando uma exceção (por regra aborta o programa) se um índice inválido é passado. Nenhum valor padrão é dado para 'opos'.

- `string &string::append(InputIterator begin, InputIterator end):`

usando esta função membro os caracteres, no comprimento implicado entre o início e fim 'InputIterators' são postos no fim da string objeto.

- `string &string::append(string argument, size_type apos, size_type an):`

o argumento (ou uma sub-string) é posta no fim da string objeto.

- `string &string::append(char const *argument, size_type an):`

os primeiros caracteres do argumento são postos no fim da string objeto.

- `string &string::append(size_type n, char c):`

usando esta função membro, 'n' caracteres 'c' são postos no fim da string objeto.

- `string &string::assign(string argument, size_type apos, size_type an):`

- um argumento (ou a substring) é posto na string objeto.

- se o argumento é do tipo 'char const *' e um argumento adicional o segundo argumento é interpretado como um valor inicial, usando 0 para iniciar 'apos'.

- `string &string::assign(size_type n, char c):`

usando esta função membro, 'n' caracteres 'c' podem ser postos na string objeto.

- `size_type string::capacity():`

retorna o número de caracteres que podem atualmente serem postos na string objeto.

- `int string::compare(string argument):`

esta função membro é usada para comparar (de acordo com o conjunto de caracteres ASCII) o texto guardado na string objeto e no argumento. O argumento pode também ser um (não 0) 'char const *'. É retornado 0 se os caracteres na string objeto e no argumento são 0; um valor negativo é retornado se o texto na string for lexicograficamente anterior ao texto do argumento; um valor positivo é retornado se o texto na string for lexicograficamente posterior ao texto do argumento.

- `int string::compare(size_type opos, size_type on, string argument):`

esta função membro é usada para comparar uma substring do texto guardada na string objeto com o texto guardado no argumento. No total os caracteres, partindo do ponto 'opos', são comparados com o texto do argumento. O argumento pode também ser um (não 0) 'char const *'.

- `int string::compare(size_type opos, size_type on, string argument, size_type apos, size_type an):`

esta função membro é usada para comparar uma substring de texto guardada na string objeto com uma substring de texto guardada no argumento. No total os caracteres da string objeto, começando pelo deslocamento 'opos', são comparados com os caracteres 'an' do argumento, começando pelo deslocamento 'apos'. Note que o argumento deve ser também uma string objeto.

- `int string::compare(size_type opos, size_type on, char const *argument, size_type an):`

esta função membro é usada para comparar uma substring de texto guardada na string objeto com uma substring de texto guardada no argumento. No total os caracteres da string objeto, começando pelo deslocamento 'opos', são comparados com os 'an' caracteres do argumento. O argumento deve ter pelo menos 'an' caracteres. Contudo, os caracteres podem ter valores arbitrários: o valor ASCII-Z não tem um significado especial.

- `size_type string::copy(char *argument, size_type on, size_type opos):`

o conteúdo da string objeto é (parcialmente) copiado ao argumento. O número efetivo de caracteres copiados é retornado. Note que os caracteres do objeto correspondente a este membro serão copiados ao argumento. Note também que depois da cópia, não será agregado um ASCII-Z à copia string. Um carácter ASCII-Z final pode ser agregado ao texto copiado usando-se a seguinte construção:

```
buffer[s.copy(buffer)] = 0;
```

- `char const *string::c_str():`

a função membro retorna o conteúdo da string objeto como uma string C ASCII-Z.

- `char const *string::data():`

retorna o texto bruto guardado na string objeto. Como este membro não retorna uma string ASCII-Z (como `'c_str()'` faz), pode ser usado para guardar e retirar qualquer forma de informação, incluindo, p.ex., séries de bytes 0's:

```
string s;  
s.resize(2);  
cout << static_cast<int>(s.data()[1]) << endl;
```

- `bool string::empty():`

retorna verdadeiro se a string objeto não contém dado.

- `string &string::erase(size_type opos, size_type on):`

esta função membro é usada para apagar uma (sub)string da string objeto.

- `iterator string::erase(iterator obegin, iterator oend):`

- se só 'obegin' é fornecida, o carácter da string objeto na posição do iterador 'obegin' é apagado.
- se 'oend' é fornecido também os caracteres da string objeto, dentro dos limites dos iteradores 'obegin' e 'oend', são apagados.

O iterador 'obegin' é retornado, apontando o carácter imediatamente seguinte para o último carácter apagado.

- `size_type string::find(string argument, size_type opos):`

retorna o índice da string objeto onde o argumento for encontrado.

- `size_type string::find(char const *argument, size_type opos, size_type an):`

retorna o índice da string objeto onde o argumento for encontrado. Nota: quando tres argumentos são especificados o primeiro argumento não pode ser um `'std::string const &'`.

- `size_type string::find(char c, size_type opos):`

retorna o índice da string objeto onde 'c' for encontrado.

- `size_type string::find_first_of(string argument, size_type opos):`

retorna o índice da string objeto onde qualquer carácter do argumento for encontrado.

- `size_type string::find_first_of(char const *argument, size_type opos, size_type an):`

retorna o índice da string objeto onde um carácter do argumento é encontrado, não importando qual carácter.

- Se 'opos' é fornecido se referirá ao índice da string objeto onde a busca pelo argumento deve começar. Se omitido, a string objeto é escaneada completamente.
- Se 'an' for fornecido indica o número de caracteres do argumento 'char const *' a ser usado na busca: define uma string parcial começando no início do argumento 'char const *'. Se omitido, são usados todos os caracteres do argumento.

- `size_type string::find_first_of(char c, size_type opos):`

retorna o índice da string objeto onde o carácter 'c' se encontra.

- `size_type string::find_first_not_of(string argument, size_type opos):`

retorna o índice da string objeto onde um carácter que não aparece no argumento é encontrado.

- `size_type string::find_first_not_of(char const *argument, size_type opos, size_type an):`

retorna o índice da string objeto onde qualquer carácter que não aparece no argumento é encontrado.

- `size_type string::find_first_not_of(char c, size_type opos):`

retorna o índice da string objeto onde outro carácter diferente de 'c' se encontra.

- `size_type string::find_last_of(string argument, size_type opos):`

retorna o último índice da string objeto onde um dos caracteres do argumento se encontra.

- `size_type string::find_last_of(char const* argument, size_type opos, size_type an):`

retorna o último índice da string objeto onde um dos caracteres do argumento se encontra.

- `size_type string::find_last_of(char c, size_type opos):`

retorna o último índice da string objeto onde o carácter 'c' se encontra.

- `size_type string::find_last_not_of(string argument, size_type opos):`

retorna o último índice da string objeto onde qualquer carácter que não aparece no argumento se encontra.

- `size_type string::find_last_not_of(char const *argument, size_type opos, size_type an):`

retorna o último índice da string objeto onde qualquer carácter que não aparece no argumento se encontra.

- `size_type string::find_last_not_of(char c, size_type opos):`

retorna o último índice da string objeto onde outro carácter diferente de 'c' se encontra.

- `istream &getline(istream &istr, string object, char delimiter):`

esta função (note que não é uma função membro da classe string) é usada para ler uma linha de texto de 'istr'. todos os caracteres até 'delimiter' (ou o fim da 'stream', aquele que vier primeiro) são lidos de 'istr' e guardados no objeto. O 'delimiter', quando presente, é removido da stream, mas não é guardado na linha. O valor padrão de 'delimiter' é '\n'.

Se o delimitador não é encontrado, 'istr.eof()' retorna verdadeiro (veja seção 5.3.1). Note que o conteúdo da última linha, termine ou não por um delimitador, será sempre copiada no objeto.

- `string &string::insert(size_type opos, string argument, size_type apos, size_type an):`

esta função membro é usada para inserir uma (sub)string do argumento na string objeto, na posição do índice da string objeto 'opos'. Os argumentos para 'apos' e 'an' devem ou ambos serem especificados ou ambos serem omitidos.

- `string &string::insert(size_type opos, char const *argument, size_type an):`

se o argumento é do tipo 'char const *', 'apos' não está disponível.

- `string &string::insert(size_type opos, size_type n, char c):`

usando esta função membro, 'n' caracteres 'c' são inseridos na string objeto.

- `iterator string::insert(iterator obegin, char c):`

o carácter 'c' é inserido na posição (do iterador) 'obegin' na string objeto. O iterador 'obegin' é retornado.

- `iterator string::insert(iterator obegin, size_type n, char c):`

na posição (do iterador) 'obegin' do objeto, 'n' caracteres 'c' são inseridos. O iterador 'obegin' é retornado.

- `iterator string::insert(iterator obegin, InputIterator abegin, InputIterator aend):`

os caracteres dentro da extensão implicada pelos iteradores de entrada ('InputIterators') 'abegin' e 'aend' são inseridos a partir da posição (do iterador) 'obegin' do objeto. O iterador 'obegin' é retornado.

- `size_type string::length():`

retorna o número de caracteres guardados na string objeto.

- `size_type string::max_size():`

retorna o número máximo de caracteres que podem ser guardados na string objeto.

- `string &string::replace(size_type opos, size_type on, string argument, size_type apos, size_type an):`

a substring de caracteres especificada no objeto é substituída pelo sub-conjunto de caracteres especificado no argumento.

Se 'on' for especificado como 0, a função membro insere o argumento no objeto no deslocamento 'opos'.

- `string &string::replace(size_type opos, size_type on, char const *argument, size_type an):`

a extensão indicada de caracteres do objeto serão substituídos pelo subset 'an' inicial de 'an'

caracteres fornecido como 'char const *argument'.

- `string &string::replace(size_type opos, size_type on, size_type n, char c):`

os caracteres da string objeto, começando na posição do índice 'opos', são substituídos por 'n' caracteres com valor 'c'.

- `string &string::replace (iterator obegin, iterator oend, string argument):`

aqui, a string implicada pelos iteradores 'obegin' e 'oend' é substituída por 'argument'. Se 'argument' é um 'char const *', um argumento extra 'an' pode ser usado, que especifica o número de caracteres de 'argument' a serem usados na substituição.

- `string &string::replace(iterator obegin, iterator oend, size_type n, char c):`

os caracteres da string objeto, na extensão implicada pelos iteradores 'obegin' e 'oend' são substituídos por 'n' caracteres com valor 'c'.

- `string string::replace(iterator obegin, iterator oend, InputIterator abegin, InputIterator aend):`

aqui os caracteres da string objeto, na extensão implicada pelos iteradores 'obegin' e 'oend' são substituídos pelos caracteres na extensão implicada pelos iteradores de entrada ('InputIterators') 'abegin' e 'aend'.

- `void string::resize(size_type n, char c):`

a string guardada na string objeto é re-dimensionada para 'n' caracteres. O segundo argumento é opcional, nesse caso o valor `c = 0` é usado. Se fornecido e a string for aumentada, os caracteres extra são iniciados com 'c'.

- `size_type string::rfind(string argument, size_type opos):`

retorna o índice da string objeto onde 'argument' se encontra. A busca procede ou do fim da string objeto ou de seu deslocamento 'opos' para trás até o início.

- `size_type string::rfind(char const *argument, size_type opos, size_type an):`

retorna o índice da string objeto onde 'argument' se encontra. A busca procede ou do fim da string objeto ou de seu deslocamento 'opos' para trás até o início. O parâmetro 'an' especifica o número de caracteres de 'argument' começando de seu início.

- `size_type string::rfind(char c, size_type opos):`

retorna o índice da string objeto onde 'c' se encontra. A busca procede ou do fim da string objeto (ou do deslocamento 'opos', se especificado) para trás até o início.

- `size_type string::size():`

retorna o número de caracteres guardados na string objeto. Este membro é um sinônimo de 'string::length()'.

- `string string::substr(size_type opos, size_type on):`

retorna (usando um valor tipo de retorno) uma substring da string objeto. A string objeto não é modificada por 'substr()'.

- `size_type string::swap(string argument):`

troca o conteúdo da string objeto com o de 'argument'. Neste caso, 'argument' precisa ser uma string e não pode ser 'char const *'. Claro, ambas strings (objeto e argumento) são modificadas por esta função membro.

Capítulo 5: A Biblioteca IO-stream

Como uma extensão ao método stream (ARQUIVO - FILE), bem conhecido da linguagem de programação C, a linguagem C++ oferece uma biblioteca de entrada/saída (E/S, em inglês I/O) baseada em conceitos de classe.

Atrás (no Capítulo 3) já vimos exemplos do uso da biblioteca de E/S C++, especialmente o uso dos operadores inserção(<<) e extração(>>). Neste capítulo examinaremos com mais detalhes.

A discussão dos recursos de entrada e saída entregados pela linguagem de programação C++ usam substancialmente os conceitos de classe e a noção de funções membro. Mas a construção de classes será vista no próximo Capítulo 6 e herança será introduzida no Capítulo 13, pensamos ser possível introduzir os recursos de entrada e saída (E/S) bem antes que as noções das técnicas que jazem por trás destes tópicos.

A maioria das classes da C++ de E/S levam nomes começando com basic_ (como basic_ios). Contudo esses nomes basic_ não são encontrados regularmente nos programas C++, já que a maioria das classes são também definidas usando typedef, como:

```
typedef basic_ios<char>      ios;
```

Como a C++ define ambos tipos, char e wchar_t, os recursos de E/S foram desenvolvidos usando o modelo de mecanismo. Como será mais elaborado no Capítulo 18, através deste modo era possível construir um software genérico, que poderia ser usado com ambos tipos, char e wchar_t. Assim, analogamente para a definição typedef acima existe outra

```
typedef basic_ios<wchar_t>   wios;
```

Esta definição de tipo é usada para o tipo wchar_t. Devido à existência destas definições de tipo, o prefixo basic_ pode ser omitido nas Anotações sem perda de continuidade. Nas Anotações a ênfase está preliminarmente no padrão do tipo de caracter de 8 bits.

Como efeito colateral desta implementação é necessário acentuar que já não é mais correto declarar objetos istream usando declarações adiantadas, como:

```
class ostream;           // agora errôneo
```

Em vez disso, as fontes que precisam declarar classes istream têm que proceder assim:

```
#include <iosfwd>         // modo correto de declarar classes istream
```

O uso da biblioteca C++ de E/S tem uma vantagem adicional em segurança dos tipos. Os objetos (ou valores plenos) são inseridos nas streams. Compare com a situação comumente encontrada na

linguagem C, onde a função `fprintf()` é usada para indicar que tipo de valor é esperado onde. Comparado a esta situação anterior o método da linguagem C++ em `iostream` usa os objetos imediatamente onde seus valores aparecem, como em:

```
cout << "There were " << nMaidens << " virgins present\n";
```

O compilador é notificado do tipo da variável `nMaidens` inserindo o valor próprio no lugar apropriado da sentença colocada em `cout iostream`.

Comparemos esta situação encontrada na linguagem C. Apesar de que os compiladores C estão se tornando cada vez mais espertos com o passar dos anos e apesar de que bem projetados os compiladores C podem emitir um alerta numa confusão de especificação de tipo e o tipo da variável encontrada na posição correspondente da lista de argumentos do comando `printf()`, não podem fazer muito mais que alertar da situação. A segurança de tipos na linguagem C++ previne a confusão de tipos, uma vez que não haja coincidência de tipo.

Além disto, `iostream` oferece mais ou menos o mesmo conjunto de possibilidades que o padrão baseado em ARQUIVO de E/S usado na linguagem C: pode-se abrir, fechar, posicionar, ler, escrever, etc., em arquivos. Na linguagem C++ a estrutura básica de ARQUIVO (FILE) continuam válidas. A linguagem C++ agrega E/S baseadas em classes às E/S baseadas em ARQUIVOS, resultando em segurança de tipo, extensibilidade e clareza de projeto. No padrão ANSI/ISO a intenção era a independência da arquitetura de E/S. As aplicações anteriores da biblioteca `iostream` nem sempre cumpriam os padrões, resultando em muitas extensões aos padrões. O software desenvolvido assim teve que ser parcialmente reescrito no tocante às E/S. Isto é dito para aqueles que agora são forçados a modificar softwares existentes, todas as facilidades e extensões disponíveis anteriormente podem ser reconstruídas facilmente usando o padrão ANSI/ISO no tocante à biblioteca de E/S. Nem todas estas reimplementações podem ser cobertas neste Capítulo, já que a maioria usa tópicos de herança e polimorfismo que serão vistos nos Capítulos 13 e 14 respectivamente. As reimplementações selecionadas serão vistas no Capítulo 20 e as referências abaixo a seções particulares nesse capítulo serão dadas nos lugares apropriados.

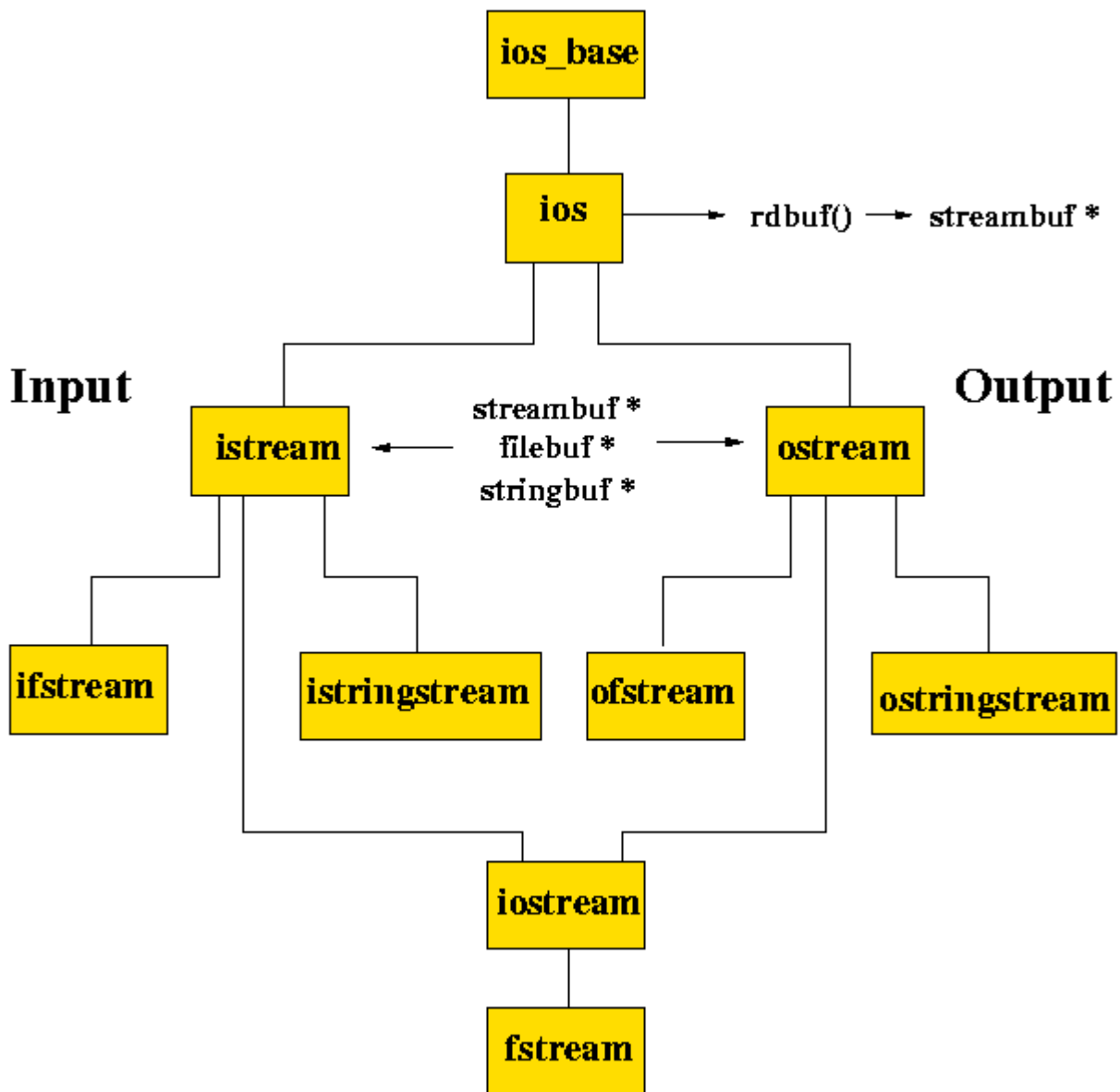


Figura 3(Classas de E/S Centrais)

Este capítulo está organizado como segue(veja também a Figura 3):

- A classe ios_base representa a fundamentação sobre a qual a biblioteca de E/S iostream foi construída. A classe ios forma o alicerce de todas as operações de E/S e define , entre outras coisas, os recursos de inspeção do estado das streams de E/S e formatação das saídas.
- A classe ios foi derivada diretamente de ios_base. Cada classe da biblioteca de E/S é derivada da classe ios e herda suas capacidades (e, por implicação, da ios_base). É indispensável que o leitor tenha isto em mente ao ler este capítulo. O conceito de herança não será discutido aqui, mas no

Capítulo 13.

Uma importante função da classe `ios` é definir a comunicação com o bufer que as streams usam. O bufer é um `streambuf` objeto (ou é derivado da classe `streambuf`) e responsável pelas entradas e/ou saídas. Isto significa que os objetos `iostream` não realizam operações de entradas/saídas por si, mas deixam aos objetos (`stream`) bufer com os que estão associados.

- Em seguida serão discutidos os recursos básicos de saída da linguagem C++. A classe básica de saída é `ostream`, define o operador inserção bem como outros recursos para escrever informações em streams. Além de inserir informações em arquivos, é possível colocar informações em buffers de memória, para o que a classe `ostream` está disponível. A formatação da saída em grande parte possível usa os recursos definidos na classe `ios`, mas também é possível inserir comandos de formatação diretamente em streams usando manipuladores. Este aspecto da linguagem C++ também é discutido.
- Os recursos básicos de entrada da linguagem C++ estão disponíveis na classe `istream`. Esta classe define o operador inserção e os recursos correlatos de entrada. Analogamente à classe `ostream`, uma classe `istream` é acessível para extrair informações dos buffers de memória.
- Finalmente diversos tópicos de E/S correlatos são discutidos: outros tópicos combinando leitura e escritura, usando streams e mistos usando C e C++ objetos `filebuf`. Outros tópicos relacionados estão vistos em outros lugares das Anotações, p.ex., no Capítulo 20.

Na biblioteca `iostream` os objetos `stream` têm um papel limitado: formam a interface entre, de um lado, os objetos a serem entrados ou saídos e, por outro lado, o `streambuf` que é responsável pelas entradas e saídas ao dispositivo para o qual o `streambuf` foi criado. Este método permite construir um novo tipo de `streambuf` para um novo tipo de dispositivo e usar esse `streambuf` em combinação com a 'antiga boa' classe `istream` – ou `ostream`. É importante entender a diferença entre os papéis da formatação dos objetos `iostream` e o armazenamento em bufer interfaceando um dispositivo externo, como feito num `streambuf`. Interfaceando a novos dispositivos (como sockets ou descritores de arquivos) requer que construamos um novo tipo de `streambuf`, não um novo objeto `istream` ou `ostream`. Uma classe envoltório pode ser construída em torno da classe `istream` ou `ostream` para facilitar o acesso a um dispositivo especial. Assim é como as classes `stringstream` foram construídas.

5.1: Arquivos Cabeçalho Especiais

Diversos arquivos cabeçalho estão definidos na biblioteca `iostream`. Dependendo da situação se pode usar os seguintes arquivos cabeçalho:

- `#include <iosfwd>`: A fonte deve usar esta diretiva ao processador se uma declaração prévia das classes de `ostream` são necessárias. Por exemplo, se uma função define um parâmetro de referência a `ostream` então, quando esta função é declarada, não haverá necessidade do compilador saber exatamente a qual `ostream` se refere. No arquivo cabeçalho, ao declarar-se tal função a classe `ostream` necessita somente ser declarada. Não se pode usar:

```
class ostream;    // declaração errônea
```

```
void someFunction(ostream &str);
```

Em seu lugar deve-se usar:

```
#include <iosfwd> // declara corretamente a classe ostream
```

```
void someFunction(ostream &str);
```

- `#include <streambuf>`: Deve-se usar esta diretiva quando se usa as classes `streambuf` ou `filebuf`. Veja as seções 5.7 e 5.7.2.
- `#include <istream>`: Deve-se usar esta diretiva ao usar a classe `istream` ou quando usar ambas classes de entrada e saída. Veja seção 5.5.1.
- `#include <ostream>`: Deve-se usar esta diretiva ao usar a classe `ostream` ou quando ambas classes de entrada e saída. Veja seção 5.4.1.
- `#include <iostream>`: Deve-se usar esta diretiva ao usar objetos `stream` globais (como `cin` e `cout`).
- `#include <fstream>`: Deve-se usar esta diretiva ao usar as classes `stream` de arquivo. Veja as seções 5.4.2, 5.5.2 e 5.8.4
- `#include <sstream>`: Deve-se usar esta diretiva ao processador quando usarmos as classes `stream` e `string`. Veja as seções 5.4.3 e 5.5.3.
- `#include <iomanip>`: Deve-se usar esta diretiva com os manipuladores parametrizados. Veja seção 5.6.

5.2: O fundamento: a classe `ios_base`

A classe `ios_base` é o alicerce de todas operações de E/S e define, entre outras coisas, os recursos de inspeção do estado das `streams` de E/S e ainda os recursos de formatação da saída. Todas as classes da biblioteca de E/S das `streams`, deriva da classe `ios` e herda suas capacidades.

A discussão da classe `ios_base` precede a introdução dos membros usados para ler e escrever as streams. Como a classe `ios_base` é o alicerce sobre o qual toda E/S em C++ foram construídas a introduzimos como a primeira classe da biblioteca C++ de E/S.

Note, contudo, que como em C, as E/S em C++ não é parte da linguagem (apesar de ser parte do padrão ANSI/ISO da linguagem C++): apesar de ser tecnicamente possível ignorar todos os recursos predefinidos, ninguém atualmente o faz e a biblioteca de E/S representa, por isso, de fato o padrão de E/S em C++. Note-se também que, como mencionado anteriormente, as classes `iostream` não realizam entradas e saídas por si, mas delegam estas funções a uma classe auxiliar: a classe `streambuf` ou suas derivadas.

Para completar, é de se notar que não é possível construir diretamente um objeto `ios_base`. Como discutido no Capítulo 13, as classes derivadas de `ios_base` (como `ios`) pode construir objetos `ios_base` usando o construtor `ios_base::ios_base()`.

A classe seguinte na hierarquia de `iostream` (veja figura 3) é a classe `ios`. Como as classes `stream` herdam de `ios_base`, na prática a distinção entre `ios_base` e `ios` é muito importante. Por isso os recursos dados atualmente pela `ios_base` são discutidos como recursos da `ios`. O leitor que esteja interessado nos recursos particulares de cada classe deve consultar os arquivos cabeçalhos de cada classe (p.ex., `ios_base.h` e `basic_ios.h`)

5.3: Interfaceando objetos 'streambuf': a classe 'ios'

A classe `ios` foi derivada diretamente da `ios_base`, e define de fato os alicerces de todas as classes `stream` da biblioteca de E/S da linguagem C++.

É possível construir objetos `ios` diretamente, às vezes isto é difícil de conseguir. O propósito da classe `ios` é entregar os recursos da classe `basic_ios` e agregar diversos recursos novos, todos relacionados ao gerenciamento dos objetos `streambuf` que são gerenciados pelos objetos da classe `ios`.

Todas as outras classes são ou direta ou indiretamente derivadas da `ios`. Isto quer dizer, como explicado no Capítulo 13, que todos os recursos oferecidos pela classe `ios_base` estão também disponíveis em outras classes `stream`. Antes de examinarmos estas classes adicionais `stream`, os recursos oferecidos pela classe `ios` (e implicitamente pela `ios_base`) são agora introduzidos.

A classe `ios` oferece diversas funções membro, a maioria das quais relativas à formatação. Outras funções membro frequentemente usadas são:

- `streambuf *ios::rdbuf()`: Esta função retorna um apontador ao `streambuf` objeto, formando uma interface entre o objeto `ios` e o dispositivo com o qual `ios` se comunica. Ver seção 20.1.2 para maiores informações sobre a classe `streambuf`.
- `streambuf *ios::rdbuf(streambuf *new)`: Esta função é usada para associar um objeto `ios` a outro `streambuf` objeto. Um apontador ao objeto `ios` original é retornado. O objeto apontado não é destruído quando o objeto `stream` sae do escopo, mas é possuído pelo que chama `rdbuf()`.
- `ostream *ios::tie()`: Esta função retorna um apontador ao objeto `stream` que está ligado (veja o membro seguinte). O objeto `ostream` retornado sempre é evacuado antes da informação ser posta ou retirada do objeto `ios` de onde o membro `tie()` é chamado. O valor 0 retornado indica que no momento nenhum objeto está associado ao objeto `ios`. Veja a seção 5.8.2 para detalhes.
- `ostream *ios::tie(ostream *new)`: Esta função é usada para associar um objeto `ios` a outro objeto `ostream`. Um apontador ao objeto original `ios` é retornado. Veja a seção 5.8.2 para detalhes.

5.3.1: Condição de estados

As operações com streams podem ser bem sucedidas ou falhar por diversas razões. Sempre que uma operação falha, as operações seguintes de leitura ou escritura com a stream são suspensas. É possível inspecionar (e possivelmente limpar) a condição de estado da stream e assim o programa pode reparar o problema, no lugar de abortar.

As condições são representadas pelas seguintes flags de condição:

- `ios::badbit`: Se verdadeiro indica requisição de operação ilegal no nível do objeto `streambuf` com o qual a stream interfaceia. Veja a função membro abaixo para alguns exemplos.
- `ios::eofbit`: Se verdadeiro o objeto `ios` detetou fim de arquivo.
- `ios::failbit`: Se verdadeiro uma operação feita pelo objeto `stream` falhou (como se espera extrair um inteiro quando caracteres numéricos não estão disponíveis na entrada). Neste caso a stream não pode realizar a operação requisitada.
- `ios::goodbit`: Se verdadeiro é porque nenhum dos outros o será.

Diversas funções membro manipulam ou determinam os estados dos objetos `ios`:

- `ios::bad()`: Esta função retorna 1 (verdadeiro) quando `ios::badbit` for 0 (falso). Se verdadeiro indica que uma operação ilegal foi requerida a nível do objeto `streambuf` com o qual a stream está associada. O que isto significa? Indica que o `streambuf` se comportou inesperadamente mal. Considere o seguinte exemplo:

```
ostream::error(0);
```

Isto constrói um objeto `ostream` sem um `streambuf` de trabalho. Como este `streambuf` jamais operará propriamente, seu `ios::badbit` será posto em 1 desde o início: `error.bad()` retorna verdadeiro.

- `ios::eof()`: Esta função membro retorna 1 (verdadeiro) quando o fim de arquivo (EOF) for encontrado (i.e., `ios::eofbit` foi posto em verdadeiro) e 0 (falso) de outra forma. Se assume que estamos lendo linha a linha de `cin`, mas a última linha não terminou com o carácter final `\n`. Nesse caso `getline()` espera ler o delimitador `\n` e aparece EOF antes. Isto põe `ios::eofbit` em verdadeiro e `cin.eof()` retorna verdadeiro. Por exemplo, assumamos que `main()` executa os seguintes comandos:

```
getline(cin, str);
cout << cin.eof();
```

Em seguida :

```
echo -n "hello world" | program
```

o valor 1 (EOF sensorado) é impresso, em seguida:

```
echo "hello world" | program
```

- valor 0 (não EOF sensorado) é impresso.
- `ios::fail()`: Esta função retorna 1 (verdadeiro) quando `ios::bad()` retorna verdadeiro ou quando `ios::failbit` for verdadeiro e 0 sob outras condições. No exemplo acima, `cin.fail()` retorna falso ou não (já que lemos uma linha). Contudo, tentando executar um segundo comando `getline()` o comando fará `ios::failbit` verdadeiro, causando `cin.fail()` retornar verdadeiro. O valor `fail()` é retornado pela interpretação booleana de um objeto stream (veja abaixo).
- `ios::good()`: Esta função retorna o valor da flag `ios::goodbit`. Retorna 1 quando nenhuma outra condição de erro (`ios::badbit`, `ios::eofbit`, `ios::failbit`) existia.

Considere este pequeno programa:

```
#include <iostream>
#include <string>
```

```

using namespace std;

void estado()
{
    cout << "\n"
           "Mau: " << cin.bad() << " "
           "Falhou: " << cin.fail() << " "
           "Eof: " << cin.eof() << " "
           "Bom: " << cin.good() << endl;
}

int main()
{
    string line;
    int x;

    cin >> x;
    estado();

    cin.clear(); //Apaga o erro
    getline(cin, line);
    estado();

    getline(cin, line);
    estado();
}

```

Quando este programa processa um arquivo com duas linhas, contendo, respectivamente 'Olá Mundo' e a segunda linha não é terminada pelo caracter \n, mostra o seguinte resultado:

```
Mau: 0 Falhou: 1 Eof: 0 Bom: 0
```

```
Mau: 0 Falhou: 0 Eof: 0 Bom: 1
```

```
Mau: 0 Falhou: 0 Eof: 1 Bom: 0
```

Assim, extraíndo x falha (good() retorna 0). Então o erro é apagado e a primeira linha é lida com sucesso (good() retorna 1). Finalmente a segunda linha é lida (incompletamente): good() retorna 0 e eof() retorna 1.

Interpretando as streams como valores booleanos:

As streams podem ser postas em expressões de valores booleanos. Alguns exemplos são:

```

if (cin)                // cin interpretada como booleana
if (cin >> x)           // cin interpretada como booleana depois de uma

```

```

// extração
if (getline(cin, str)) // getline retorna cin

```

Quando se interpreta uma stream como um valor lógico não é `ios::fail()` que é interpretada. Assim os exemplos acima podem ser reescritos como:

```

if (not cin.fail())
if (not (cin >> x).fail())
if (not getline(cin, str).fail())

```

As formas mágicas acima, contudo, são usadas quase exclusivamente.

Os seguintes membros manejam estados de erro:

- `ios::clear()`: Quando uma condição de erro ocorreu e a condição pode ser reparada, então `clear()` pode ser chamada para limpar o estado de erro do arquivo. Uma versão sobrecarregada aceita as flags de estado que depois de limpas serão modificadas em atualização: `ios::clear(int state)`
- `ios::rdstate()`: Esta função membro retorna o estado atual das flags como foram postas por um objeto `ios`. Para testar uma flag particular usa-se um operador de bit:

```

if (iosObject.rdstate() & ios::good)
{
    // state is good
}

```

- `ios::setstate(int flags)`: Este membro é usado para alterar um conjunto de flags em particular. O membro `ios::clear()` é um atalho para limpar todas as flags. Claro que limpando as flags não se corrigem automaticamente as condições de erro. A estratégia deve ser:
- É detectada uma condição de erro,
- O erro é reparado,
- O membro `ios::clear()` é chamado.

A linguagem C++ suporta um mecanismo de exceção para manipulação de situações excepcionais. De acordo com o standard ANSI/ISO, as exceções podem ser usadas com objetos streams. As exceções são vistas no Capítulo 8. Usando exceções com objetos streams é visto na seção 8.7.

5.3.2: Formatando entradas e saídas

A forma como a informação é escrita nas (ou, ocasionalmente, lidas das) streams pode ser controlada por flags de formatação.

A formatação é usada quando é necessário controlar o tamanho de um campo de saída ou bufer de entrada e tal formato é usado para determinar a maneira (p.ex., a base) na qual um valor é mostrado. Muito da formatação pertence a realm da classe ios, contudo muito da formatação agora é usada com streams, como a vindoura classe ostream. Como a formatação é controlada por flags, definidos na classe ios, foi considerado melhor discutir a formatação com a própria classe ios que considerá-la uma classe derivada, o que seria sempre algo arbitrário.

A formatação é controlada por um conjunto de flags de formatação. Estas flags podem , basicamente, serem alterados de duas maneiras: usando-se funções membros especializadas, vistas na seção 5.3.2.2 ou usando-se manipuladores, que são diretamente inseridos nas streams. Os manipuladores não se aplicam diretamente à classe ios, já que requerem o uso do operador inserção. Conseqüentemente são discutidos mais tarde (seção 5.6).

5.3.2.1: As flags de formatação

A maior parte das flags de formatação dizem respeito à informação de saída. A informação pode ser escrita nas streams de saída através de basicamente dois modos: saída binária que escreverá a informação diretamente na stream de saída, sem conversão a um formato legível aos humanos. P.ex., um valor inteiro é escrito como um bloco de quatro bytes. Alternativamente as saídas formatadas converterãoos valores que estão em bytes na memória do computador a caracteres ASCII, para criar uma forma legível aos humanos.

As flags de formatação podem ser usadas para definir a forma que esta conversão terá lugar, para controlar, p.ex., o número de caracteres que serão escritos na stream de saída.

As seguintes flags de formatação estão disponíveis (veja também as seções 5.3.2.2 e 5.6):

- `ios::adjustfield`: Máscara de valores usada com uma flag que define o modo como os valores serão ajustados a campos amplos (`ios::left`, `ios::right`, `ios::internal`). Exemplo, colocar o valor 10 alinhado à esquerda num campo de 10 caracteres:

```
cout.setf(ios::left, ios::adjustfield);  
cout << " " << setw(10) << 10 << " " << endl;
```

- `ios::basefield`: Máscara de valores usada em combinação com uma flag de acerto de uma base

para valores inteiros da saída (ios::dec, ios::hex ou ios::oc). Exemplo, imprimir o valor 57005 como um número hexadecimal:

```
cout.setf(ios::hex, ios::basefield);
cout << 57005 << endl;
// or, using the manipulator:
cout << hex << 57005 << endl
```

o ios::boolalpha: Para mostrar valores booleanos como texto 'verdadeiro' (true) e 'falso'(false). Como padrão esta flag não é colocada. Manipuladores correspondentes: boolalpha e noboolalpha. Exemplo, imprimir o valor booleano 'verdadeiro' no lugar de 1:

```
cout << boolalpha << (1 == 1) << endl;
```

- ios::dec: Para ler e mostrar valores inteiros na base decimal (i.e., base 10). Este é o padrão.

com setf() a máscara de valores ios::basefield deve ser dada. Correspondente manipulador: dec.

- ios::fixed: Para mostrar valores reais com notação fixa (p.ex., 12,25), em oposição à notação científica. Se somente uma mudança de notação é requerida a máscara de valores ios::floatfield deve ser posta quando usado com setf(). Exemplo: Veja ios::scientific abaixo. Correspondente manipulador: fixed.

Outro uso de ios::fixed é por em um número fixo de casas decimais quando se imprime valores em ponto flutuante ou duplos. Veja ios::precision na seção 5.3.2.2.

- ios::floatfield: Máscara de valores usada em combinação com uma flag para determinar o modo como números reais serão mostrados (ios::fixed ou ios::scientific). Exemplo:

```
cout.setf(ios::fixed, ios::floatfield);
```

- ios::hex: Para ler e mostrar valores inteiros como hexadecimais (i.e., base 16).

Com setf() a máscara de valores ios::basefield deve ser dada. Manipulador correspondente: hex.

- ios::internal: Para agregar caracteres de preenchimento (espaços brancos como padrão) entre o sinal de menos de números negativos e o valor.

Com setf() a máscara adjustfield deve ser dada. Manipulador correspondente: internal.

- `ios::left`: Para ajustar (inteiros) valores num campo maior que o necessário para mostrar os valores. Como padrão os valores são colocados à direita (veja abaixo).

Com `setf()` a máscara `adjustfield` deve ser dada. Manipulador correspondente: `left`.

- `ios::oct`: Para mostrar valores inteiros como octais (i.e., base 8).

Com `setf()` a máscara `ios::basefield` deve ser dada. Manipulador correspondente: `oct`.

- `ios::right`: Para ajustar à direita valores inteiros em campos maiores que o necessário para os valores. Este é o acerto padrão.

Com `setf()` a máscara `adjustfield` deve ser dada. Manipulador correspondente: `right`.

- `ios::scientific`: Para mostrar valores reais na notação científica (p.ex., `1.2e+03`).

Com `setf()` a máscara `ios::floatfield` deve ser dada. Manipulador correspondente: `scientific`.

- `ios::showbase`: Para mostrar a base numérica de um inteiro. Com hexadecimais o prefixo `0x` é usado, com octais o prefixo `0`. Com os decimais nenhum prefixo é usado. Manipulador correspondente: `showbase` e `noshowbase`.
- `ios::showpoint`: Mostra um decimal sem os zeros não significativos no início ou fim. Quando esta flag é usada uma inserção como segue é feita:

```
cout << 16.0 << ", " << 16.1 << ", " << 16 << endl;
```

que resulta em:

```
16.0000, 16.1000, 16
```

Note que o último `16` é um número inteiro e não um número real e não é mostrada a vírgula: `ios::showpoint` não tem efeito aqui.

Se não for usada `ios::showpoint` então os zeros finais são descartados. Se a parte decimal é nula então a vírgula também é descartada. Manipulador correspondente: `showpoint`.

- `ios::showpos`: Mostra o caracter `+` com números positivos. Manipulador correspondente: `showpos`.
- `ios::skipws`: Usado para extrair informações das streams. Quando esta flag é dada (que é o padrão) os espaços iniciais (brancos, tabulação, etc.) são saltados quando um valor é extraído da

stream. Se a flag não é dada esses caracteres não são saltados.

- `ios::unitbuf`: Evacua a stream depois da operação de saída.
- `ios::uppercase`: Usa letras maiúsculas na representação dos valores (hexadecimais ou científicos).

5.3.2.2: Funções membro modificadoras do formato

Diversas funções membro formatam as E/S. Frequentemente existem manipuladores correspondentes que podem ser inseridos ou extraídos das streams usando-se os operadores inserção e extração. Veja a seção 5.6 para uma discussão dos manipuladores disponíveis.

Elas são:

- `ios ©fmt(ios &obj)`: Esta função copia todas as definições de formatação do objeto para o ios objeto atual.
- `ios::fill() const`: Retorna (como char) o caracter de preenchimento. Como padrão é o espaço em branco.
- `ios::fill(char padding)`: Redefine o caracter de preenchimento. Retorna (como char) o caracter anterior. Manipulador correspondente: `setfill()`.
- `ios::flags() const`: Retorna a coleção de flags válidas atualmente que controlam a formatação da stream para a qual a função foi chamada. Para inspecionar uma flag em especial use o operador binário `&`, p.ex.:

```
if (cout.flags() & ios::hex)
{
    // saída de valores hexadecimais inteiros
}
```

- `ios::flags(fmtflags flagset)`: Retorna o conjunto de flags anterior e define um novo conjunto como flagset definido como conjunto de flags de formatação combinados pelo operador binário ou. Nota: quando definindo flags com este membro o conjunto anterior antes tem que ser retirado. Por exemplo: Para mudar a conversão de números em cout de decimal a hexadecimal usando esta função, faça:

```
cout.flags(ios::hex | fis.flags() & ~ios::dec);
```

Alternativamente um dos seguintes comandos pode ser usado:

```
cout.setf(ios::hex, ios::basefield);
cout << hex;
```

- `ios::precision()` const: Retorna (como inteiro) o número de dígitos significante usado para a saída de valores reais (padrão 6).
- `ios::precision(int signif)`: Redefine o número de dígitos significantes usados na saída de números reais, retorna (como inteiro) o número anteriormente usado. Manipulador correspondente: `setprecision()`. Exemplo:

Arredondar todos os valores duplos mostrados na tela a um número fixo de dígitos (p.ex., 3) depois da vírgula.

```
cout.setf(ios::fixed);
cout.precision(3);
cout << 3.0 << " " << 3.01 << " " << 3.001 << endl;
cout << 3.0004 << " " << 3.0005 << " " << 3.0006 << endl;
```

Note-se que o valor 3.0005 é arredondado a 3.001.

- `ios::setf(fmtflags flags)`: Retorna o conjunto das flags anteriores e põe mais uma flag de formatação (usando o operador binário ou `|()` para combinar múltiplas flags). Manipuladores correspondentes: `setiosflags` e `resetiosflags`.
- `ios::setf(fmtflags flags, fmtflags mask)`: Retorna o conjunto anterior de flags, limpa todas as flags da máscara e põe as flags especificados em flags. Os valores predeterminados da máscara são `ios::adjustfiels`, `ios::basefield` e `ios::floatfield`. (Por exemplo: ?)
- `setf(ios::left, ios::adjustfield)`: É usada para ajustar os valores à esquerda. (alternativamente `ios::right` e `ios::internal` podem ser usadas).
- `setf(ios::hex, ios::basefield)` : É usada para ativar a representação hexadecimal de valores inteiros (alternativamente `ios::dec` e `ios::oct` podem ser usadas).
- `setf(ios::fixed, ios::floatfield)`: É usada para ativar a representação de valores reais com vírgula fixa (alternativamente pode-se usar `ios::scientific`).
- `ios::unsetf(fmtflags flags)`: Retorna os flags anteriores e limpa a formatação especificada dos flags (deixando os demais flags inalterados). A limpeza de um flag limpo (p.ex., `cout.unsetf(ios::dec)`) não tem efeito.
- `ios::width()` const: Retorna (como inteiro) o comprimento do campo atual (o número de caracteres

a serem preenchidos por um valor numérico na próxima operação de inserção). Padrão: 0, significa 'tantos caracteres quantos necessários para o valor'). Manipulador correspondente: `setw()`.

- `ios::width(int nchars)`: Retorna (como inteiro) o tamanho do campo anteriormente usado, redefine o valor para `nchars` para a próxima operação de inserção. Note que o tamanho do campo é posto em 0 depois de cada operação de inserção e `width()` ai não tem efeito para o próximo valor como `char *` ou valores `string`. Manipulador correspondente: `setw(int)`.

5.4: Saída

A saída na linguagem C++ está primariamente baseada na classe `ostream`. A classe `ostream` define os operadores básicos e membros para inserir informação nas streams: O operador inserção (`<<`) e membros especiais como `ostream::write()` para escrever informação não formatada das streams.

Diversas outras classes derivam da classe `ostream` e agregam suas próprias especialidades. Nesta seção 'saída' introduziremos:

- A classe `ostream`, que oferece os recursos básicos de saída;
- A classe `ofstream` que permite abrir arquivos para escrever (comparável à `fopen(filename, "w")` da linguagem C);
- A classe `ostringstream` que permite escrever informações na memória antes que em arquivos (comparável à `sprintf()` da linguagem C).

5.4.1: Saída Básica: A classe `ostream`

A classe `ostream` é a classe que define os recursos básicos de saída. Os objetos `cout`, `clog` e `cerr` são objetos da `ostream`. Note que todos os recursos definidos na classe `ios`, no concernente à saída, estão disponíveis na classe `ostream` também, devido ao mecanismo de herança (discutido no Capítulo 13).

Podemos construir objetos `ostream` usando os seguintes construtores `ostream`:

- `ostream object(streambuf *sb)`: Este construtor é usado para construir um envoltório num

streambuf existente que pode ser a interface a um arquivo existente. Veja o Capítulo 20 para exemplos. O que isto trás à baila é que não é possível construir um objeto ostream pleno. Quando cout ou seus amigos são usados, usamos objetos ostream predefinidos que já foram criados para nós e suas interfaces, p.ex., a stream de saída padrão usa um streambuf (também predefinido) que manipula a interface.

Para se usar a classe ostream nas fontes C++ deve-se agregar a diretiva ao preprocessador `#include <ostream>`. Para usar os objetos ostream predefinidos deve-se incluir a diretiva ao preprocessador `#include <iostream>`.

5.4.1.1: Escritura nos objetos 'ostream'

A classe ostream suporta saídas binárias e formatadas.

O operador inserção (<<) é usado para se inserir valores, de modo seguro em relação aos tipos, nos objetos ostream. Esta é chamada saída formatada, já que valores binários da memória do computador são convertidos a caracteres ASCII, legíveis aos humanos, de acordo com certas regras de formatação.

Note que o operador inserção aponta para um objeto ostream lendo a informação será inserida. A associatividade normal do << continua inalterada, assim quando um comando como

```
cout << "Olá " << "Mundo";
```

É encontrado os dois primeiros operandos da esquerda são avaliados (cout << “Olá”) e um objeto ostream &, que atualmente é o próprio objeto cout, é retornado. Agora o comando se reduz a:

```
cout << "Mundo";
```

E a segunda string é inserida em cout.

O operador << possui muitas (sobrecargas) variantes, tantas quantas tipos de variáveis se pode inserir em objetos ostream. Existe um operador << sobrecarregado esperando um inteiro, um duplo, um apontador, etc. etc.. Para cada parte da informação que é inserida na stream o operador retorna o objeto ostream onde a informação foi inserida e a parte seguinte da informação é processada.

As streams não possuem os recursos de formatação da saída, como as funções da linguagem C `form()` e `vform()`. Contudo não é difícil realizar estes recursos no âmbito das streams, `form()` - como funcionalidade é ardentemente sempre requerida em programas C++. Mas, como é potencialmente insegura em relação aos tipos, é melhor evitar completamente esta função.

É verdade também que a linguagem C++ já não necessita a funcionalidade de `form()`.

Quando é necessário escrever em arquivos binários, normalmente a formatação de texto não é usada ou requerida: um valor inteiro pode ser escrito como uma série de bytes inalterados, não como séries de caracteres numéricos de 0 a 9. As seguintes funções membro são usadas para escrever 'arquivos binários':

- `ostream& ostream::put(char c)`: Esta função membro escreve um único caracter na stream de saída. Como um caracter é um byte, esta função pode também ser usada para escrever um único byte a um arquivo de texto.
- `ostream& ostream::write(char const *buffer, int length)`: Esta função membro escreve `length` bytes retidos em `char const *buffer` no objeto `ostream`. Os bytes são escritos como estão no buffer, não é feita nenhuma formatação. Note que o primeiro argumento é do tipo `char const *`: um `type_cast` é requerido para escrever qualquer outro tipo. Por exemplo, para escrever um inteiro como uma série de bytes sem formatação:

```
int x;  
out.write(reinterpret_cast<char const *>(&x), sizeof(int));
```

5.4.1.2: Posicionamento numa 'ostream'

Apesar de que nem todos os objetos `ostream` suportem posicionamento, usualmente suportam. Isto significa que é possível reescrever uma seção da stream escrita antes. O posicionamento é frequentemente usado em aplicativos de bases de dados onde é necessário ser possível acessar a informação aleatoriamente.

Os seguintes membros para posicionamento estão disponíveis:

- `pos_type ostream::tellp()`: Esta função retorna a posição (absoluta) atual onde a seguinte operação de escritura na stream terá lugar. Para todos os fins práticos `pos_type` deve ser considerado `unsigned long`.
- `ostream &ostream::seekp(off_type step, ios::seekdir org)`: Esta função membro é usada para posicionar a stream. A função espera um `off_type step`, o tamanho do passo em bytes a partir de `org`. Para todos os fins práticos `off_type` deve ser considerado do tipo `long`.

A origem de `step`, `org` é um valor `ios::seekdir`. Valores possíveis são:

- `ios::beg`: `org` é interpretado como o tamanho do passo relativo ao início da stream. Se `org` não é

especificado `ios::beg` é usado.

- `ios::cur`: org é interpretado como o tamanho do passo relativamente à posição atual (como retornou `tellp()`).
- `ios::end`: org é interpretado como o tamanho do passo relativamente do fim atual da stream.

É permitido fazer um posicionamento além do fim do arquivo.

Escrever bytes para além do fim do arquivo EOF criará um tampão aos bytes intermediários com valores ASCII-Z: null-bytes. Não é permitido posicionar antes do início do arquivo. Um posicionamento em `ios::beg` causará a flag `ios::fail` se tornar verdadeira.

5.4.1.3: Evacuação das `ostream`

A menos que a flag `ios::unitbuf` seja verdadeira, a informação escrita a um objeto `ostream` não é imediatamente escrita à stream física. Um bufer interno é preenchido durante as operações de escritura e quando estiver cheio é evacuado.

O bufer interno pode ser evacuado sob o controle do programa:

- `ostream& ostream::flush()`: Esta funo membro escreve toda a informação buferizada na stream. A chamada a `flush()` é feita quando:
- O objeto `ostream` deixa de existir,
- `endl` ou manipuladores `flush` (veja a seção 5.6) estão inseridos no objeto `ostream`,
- Uma stream derivada de `ostream` (como `ofstream`, veja seção 5.4.2) é fechada.

5.4.2: Saída para arquivos: A classe `ofstream`

A classe `ofstream` é derivada da classe `ostream`: possui os mesmos recursos que `ostream`, mas pode acessar arquivos para criar e escrever em arquivos.

Para que as fontes C++ possam usar a classe `ofstream` devem incluir a diretiva ao preprocessador `#include <fstream>`. Mesmo depois de incluir `fstream`, `cin`, `cout` etc. não ficam automaticamente declaradas. Se estes objetos também são necessários, então `iostream` deve ser incluída.

Os seguintes construtores existem para objetos ofstream :

- ofstream object: Este é o construtor básico. Cria um objeto ofstream que poderá ser associado ao arquivo posteriormente, usando o membro open() (veja abaixo).
- ofstream object(char const *name, int mode): Este construtor é usado para associar um objeto stream a um arquivo com nome name, usando o modo mode. O modo padrão de saída é ios::out. Veja 5.4.2.1 para uma visão completa dos modos de saída disponíveis.

No seguinte exemplo um objeto ofstream, associado a um arquivo recém criado /tmp/scratch é construído:

```
ofstream out("/tmp/scratch");
```

Note que não é possível abrir uma ofstream usando um descritor do arquivo. A razão para isso é (aparentemente) que a existência dos descritores de arquivo não é universal em todos os sistemas operacionais. Afortunadamente os descritores de arquivo podem ser utilizados (indiretamente) através de um objeto streambuf (e em algumas instalações: através de um objeto filebuf, que também é um streambuf). Os objetos streambuf são vistos na seção 5.7 e objetos filebuf na seção 5.7.2.

No lugar de associar diretamente um objeto ofstream a um arquivo, o objeto deve ser construído primeiro e aberto depois.

- void ofstream::open(char const *name, int mode): Tendo construído um objeto ofstream, a função membro open() pode ser usada para associar o objeto ofstream com o arquivo.
- ofstream::close(): É possível fechar um objeto ofstream explicitamente usando a função membro close(). A função põe em verdadeira a flag ios::fail do objeto fechado. Ao fechar o arquivo toda informação buferizada será evacuada para o arquivo associado. Um arquivo é automaticamente fechado quando o objeto ofstream associado deixa de existir.

Esta é uma sutileza: Assuma que uma stream foi construída, mas não está associada a um arquivo. P.ex., o comando ofstream ostr foi executado. Se examinamos seu estado através de good(), um valor diferente de zero (i.e., OK) é retornado. O estado 'bom' aqui indica que o objeto stream foi construído apropriadamente. Não significa que o arquivo esteja aberto. Para examinar se o arquivo está aberto usa-se ofstream::is_open(): se verdadeiro a stream está aberta. veja o exemplo seguinte:

```
#include <fstream>
#include <iostream>

using namespace std;
```

```

int main()
{
    ofstream of;

    cout << "of está aberta: " << boolalpha << of.is_open() << endl;

    of.open("/dev/null");          // num sistema Unix

    cout << "of está aberta: " << of.is_open() << endl;
}
/*
    Saída gerada:
of está aberta: falso
of está aberta: verdadeiro
*/

```

5.4.2.1: Modos para abrir objetos stream

Os seguintes modos ou flag de arquivos são definidos para construir ou abrir objetos ofstreams (ou istream, veja a seção 5.5.2). Os valores são do tipo `ios::openmode`:

- `ios::app`: Reposiciona no fim do arquivo antes de cada comando de saída. O conteúdo existente do arquivo é conservado.
- `ios::ate`: Começa inicialmente no fim do arquivo. O conteúdo existente do arquivo é conservado.
- `ios::binary`: Abre um arquivo binário (usado em sistemas que fazem diferença entre arquivos de texto e arquivos binários, como MS-DOS ou MS-Windows).
- `ios::in`: Abre um arquivo para leitura. O arquivo precisa existir.
- `ios::out`: Abre um arquivo. Se não existir o cria. Se existir o arquivo é sobre-escrito.
- `ios::trunc`: Começa inicialmente com um arquivo vazio. Qualquer conteúdo do arquivo é perdido.

As seguintes combinações de flags de arquivos possuem significado especial:

<code>out app:</code>	O arquivo é criado se não existir, a informação é sempre agregada no fim da stream;
<code>out trunc:</code>	O arquivo é recriado vazio para ser escrito;
<code>in out:</code>	A stream pode ser lida e escrita. Contudo, o arquivo precisa existir.

in | out | trunc: A stream pode ser lida e escrita. É recriado vazio primeiro.

5.4.3: Saída para a memória: A classe `ostream`

Para escrever informações na memória, usando os recursos das streams, se usam os objetos `ostream`. Estes objetos derivam dos objetos `ostream`. Os seguintes construtores e membros estão disponíveis:

- `ostream ostr(string const &s, ios::openmode mode)`: Quando se usar este construtor, o último ou ambos argumentos pode ser omitidos. É também um construtor com um só parâmetro de modo de abertura. Se for especificada a string `s` e o modo de abertura for `ios::ate`, o objeto `ostream` é inicialmente iniciado com a string `s` e as demais inserções são apendicadas ao conteúdo do objeto `ostream`. Se a string `s` é dada, não será alterada, já que qualquer informação inserida no objeto é guardada em memória dinamicamente alocada que será apagada quando o objeto `ostream` sair do escopo.
- `string ostream::str() const`: Esta função membro retorna a string guardada no objeto `ostream`.
- `ostream::str(string)`: Esta função membro inicia o objeto `ostream` com um conteúdo novo.

Antes de existir a classe `ostream` a classe `ostrstream` era comumente usada para se fazer saídas à memória. Esta classe anterior sofria do problema que depois que seu conteúdo fosse retirado, usando sua função membro `str()`, esse conteúdo era 'congelado', i.e., sua alocação dinâmica de memória nunca foi feita para depois que o objeto saísse do escopo. Contudo esta situação pode ser evitada (usando uma chamada ao membro de `ostrstream` `freeze()`, esta pode liberar facilmente a memória. A classe `ostream` não sofre desse risco. Por isso o uso da classe `ostrstream` é obsoleto a favor de `ostream`.

O seguinte exemplo ilustra o uso da classe `ostream`: Diversos valores são inseridos no objeto. Então o texto armazenado é posto numa string, cujo tamanho e conteúdo são impressos. Tais objetos `ostream` freqüentemente são usados para fazer conversões 'tipo a string', como converter um inteiro em string. Pode-se usar comandos de formatação com `ostream` também, já que são acessíveis nos objetos `ostream`.

Eis um exemplo mostrando o uso de um objeto `ostream`:

```
#include <iostream>
```

```

#include <string>
#include <sstream>
#include <fstream>

using namespace std;

int main()
{
    ostringstream ostr("Olá ", ios::ate);

    cout << ostr.str() << endl;

    ostr.setf(ios::showbase);
    ostr.setf(ios::hex, ios::basefield);
    ostr << 12345;

    cout << ostr.str() << endl;

    ostr << " -- ";
    ostr.unsetf(ios::hex);
    ostr << 12;

    cout << ostr.str() << endl;
}
/*
    Saída do programa:
Olá
Olá 0x3039
Olá 0x3039 -- 12
*/

```

5.5: Entrada

Na linguagem C++ as entradas estão baseadas primariamente na classe `istream`. A classe `istream` define os operadores básicos e membros para extrair informação das streams: o operador extração (`>>`) e em especial membros como `istream::read()` para se ler informação sem formatação de uma stream.

Da classe `istream` diversas outras classes derivam, todas possuindo as funcionalidades da classe `istream` e agregando suas especialidades. Na próxima seção introduziremos:

- A classe `istream` que oferece os recursos básicos de entrada;

- A classe `ifstream` que permite abrir arquivos para leitura (comparável à `fopen(nome_arq, "r")` da C);
- A classe `istream` para leitura de textos em arquivos (stream) mas é uma função em memória (comparável à `scanf()` da C).

5.5.1: Entrada Básica: A classe `'istream'`

A classe `istream` é que define os recursos básicos de entrada. O objeto `cin` é um objeto `istream` declarado quando na fonte se inclui `#include <iostream>`. Note que todos os recursos definidos na classe `ios`, ao que concerne às entradas, estão disponíveis também na classe `istream`, devido ao mecanismo de herança (discutido no Capítulo 13).

Os objetos `istream` são construídos usando-se os seguintes construtores `istream`:

- `istream object(streambuf *sb)`: Este construtor é usado para construir um envoltório para uma stream aberta, baseado num `streambuf` existente que será a interface com um arquivo existente. Veja o Capítulo 20 para exemplos.

Para se usar a classe `istream` em fontes C++ a diretiva ao preprocessador `#include <istream>` deve ser incluída. Para se usar o objeto `istream` `cin` a diretiva ao preprocessador `#include <iostream>` deve ser dada.

5.5.1.1: Lendo de objetos `'istream'`

A classe `istream` suporta entradas binárias formatadas e sem formato. O operador extração (`>>()`) é usado para extrair valores com segurança em relação aos tipo de objetos `istream`. Esta é chamada entrada formatada, pois os caracteres são convertidos de ASCII, legíveis aos humanos, de acordo com certas regras a valores binários e postos na memória do computador.

Note que o operador de extração (`>>`) aponta para objetos ou variáveis que receberão os valores. A associatividade normal de `>>` permanece inalterada, assim, comandos como:

```
cin >> x >> y;
```

é encontrado, os dois operandos mais à esquerda são avaliados primeiro (`cin >> x`) e um objeto `istream` `&` que é o próprio objeto `cin` é retornado. Assim o comando fica reduzido a:

```
cin >> y
```

e a variável `y` é extraída de `cin`.

O operador `>>` possui muitas variantes (sobrecarregadas), tantas quantos tipos de variáveis podem ser extraídos dos objetos `istream`. Existe uma variante sobrecarregada para a extração de um inteiro, de um duplo, de uma string, de um conjunto de caracteres, possivelmente para um apontador, etc. etc.. A extração de conjuntos de strings ou de caracteres saltará todos os espaços em branco (como padrão) e então todos os caracteres consecutivos diferentes do espaço. Depois de processar um operador de extração, o objeto `istream` onde a informação estava é retornado que será usado como um valor longo para a parte restante do comando, se houver.

As streams não possuem recursos para formatar as entradas (como as funções C `scanf()` e `vscanf()`). Contudo não é difícil conseguir esses recursos no âmbito das streams, já que as funcionalidades de `scanf()` é ardentemente requerida nos programas C++. Mas como essas funções não manipulam os tipos dos dados com segurança é melhor evitar completamente essas funcionalidades.

Quando é necessária a leitura de arquivos binários, a informação normalmente não estará formatada: um valor inteiro será lido como uma série de bytes inalterados, não como uma série de caracteres numéricos em ASCII de 0 a 9. As seguintes funções membro estão disponíveis em `istream` para leitura:

- `int istream::gcount()`: Esta função não lê de uma stream de entrada, mas retorna o número de caracteres lidos na última operação de entrada de uma stream.
- `int istream::get()`: Esta função retorna EOF ou o caracter lido por último como um valor `int`.
- `istream &istream::get(char &c)`: Esta função lê o caracter simples seguinte da stream de entrada e põe em `c`. Como o valor retornado é a própria stream, seu valor de retorno pode ser interrogado para determinar se a extração foi bem sucedida ou não.
- `istream& istream::get(char *buffer, int len [, char delim])`: Esta função lê uma série de tamanho `len-1` da stream de entrada para o conjunto que começa em `buffer`, que pode ser de comprimento até `len` bytes. Como máximo `len-1` caracteres são lidos para o `buffer`. Como padrão o delimitador é um caracter de nova linha (`'\n'`) O delimitador não é removido da stream de entrada.

Depois de ler a série de caracteres para o `buffer` um caracter ASCII-Z é escrito atrás do último caracter escrito no `buffer`. As funções `eof()` e `fail()` (veja seção 5.3.1) retornam 0 (falso) se o delimitador não for encontrado antes de `len-1` caracteres lidos. Além disso, um ASCII-Z pode ser usado como

delimitador: Assim, strings terminadas no caracter ASCII-Z podem ser lidas de um arquivo (binário). O programa que use esta função membro `get()` pode saber em avanço o número máximo de caracteres que serão lidos.

- `istream& istream::getline(char *buffer, int len [, char delim])`: Esta função opera analogamente à função membro anterior `get()`, mas o delimitador é removido da stream se for encontrado. No máximo `len-1` bytes são escritos no bufer e o caracter final ASCII-Z é posto no fim da string lida. O delimitador não é posto no bufer. Se o delimitador não for encontrado (antes de serem lidos `len-1` caracteres) a função membro `fail()` e possivelmente também `eof()` retornarão verdadeiro. Note que a classe `std::string` também tem suporte a uma função `getline()` que é usada mais freqüentemente que esta função membro `istream::getline()` (veja a seção 4.2.4).
- `istream& istream::ignore(int n , int delim)`: Esta função membro tem dois argumentos (opcionais). Quando chamada sem argumentos salta um caracter da stream de entrada. Quando chamada com um argumento, `n` caracteres serão saltados. O segundo argumento opcional especifica um delimitador: Depois de saltar `n` caracteres ou o delimitador (aquele que vier antes) a função retorna.
- `int istream::peek()`: Esta função retorna o caracter seguinte da entrada, mas não o remove da stream de entrada.
- `istream& istream::putback (char c)`: O caracter `c` que foi o último lido da stream é 'devolvido' à stream, para ser lido outra vez como o seguinte caracter. Se a operação falha é retornado EOF. Normalmente um caracter sempre pode ser devolvido. Note que `c` precisa ser o último caracter lido da stream. Voltar qualquer outro caracter falhará.
- `istream& istream::read(char *buffer, int len)`: Esta função lê até `len` bytes da stream de entrada para o bufer. Se EOF é encontrado antes, menos bytes serão lidos e a função membro `eof()` retornará verdadeiro. Esta função normalmente será usada para ler arquivos binários. A seção 5.5.2 contém um exemplo do uso desta função. A função membro `gcount()` pode ser usada para determinar o número de caracteres que foram retirados pela função `read()`.
- `istream& istream::readsome(char *buffer, int len)`: Esta função lê no máximo `len` bytes da stream de entrada para o bufer. Todos os caracteres disponíveis são postos no bufer, mas se EOF é encontrado antes, menos bytes serão lidos, sem modificar as flags `ios_base::eofbit` ou `ios_base::failbit`.
- `istream& istream::unget()`: Esta função faz um intento de por de volta o último caracter lido para a stream. Normalmente é bem sucedido se requerido somente uma vez depois da operação de

leitura, como é o caso de `putback()`.

5.5.1.2: Posicionamento com `'istream'`

Apesar de que nem todas os objetos stream suportem posicionamento, alguns sim. Isto significa que é possível ler a mesma seção da stream repeditamente. O Posicionamento é freqüentemente usado em aplicativos de bases de dados onde é necessário acesso aleatório à informação.

Os seguintes membros estão disponíveis:

- `pos_type istream::tellg()`: Esta função retorna a posição atual (absoluta) onde a próxima operação com a stream terá lugar. Para todos os fins práticos um `pos_type` deve ser considerado um longo sem sinal.
- `istream &istream::seekg(off_type step, ios::seekdir org)`: Esta função membro é usada para reposicionar a stream. A função espera um `off_type` como passo, o passo em bytes no reposicionamento desde `org`. Para todos os fins práticos um `pos_type` é um longo. A origem do passo, `org` é um valor `ios::seekdir`. Valores possíveis são:

`ios::beg`: `org` é interpretada como o passo relativo do início da stream. Se `org` não for especificado `ios::beg` é usado.

`ios::cur`: `org` é interpretada como o passo relativo da posição atual (como a retornada por `tellg()`).

`ios::end`: `org` é interpretada como o passo relativo ao fim da stream.

Como se pode ultrapassar o fim do arquivo, claro que ler dessa posição causará uma falha. Não é permitido um posicionamento antes do início do arquivo. Um pedido de deslocamento antes de `ios::beg` porá `ios::fail` como verdadeiro.

5.5.2: Entrada de streams: A classe `'ifstream'`

A classe `ifstream` deriva da classe `istream`: possui as mesmas capacidades que `istream` mas pode acessar arquivos para leitura. O arquivo precisa existir.

Para se usar a classe `ifstream` em fontes C++ é necessário incluir a diretiva ao preprocessador `#include <fstream>`.

Os seguintes construtores estão disponíveis para os objetos `ifstream`:

- `ifstream object`: Este é o construtor básico. Cria um objeto `ifstream` que pode ser associado a um arquivo posteriormente, usando-se o membro `open()` (veja abaixo).
- `ifstream object(char const *name, int mode)`: Este construtor é usado para associar um objeto `ifstream` com um arquivo com nome `name`, usando o modo de entrada `mode`. O modo de entrada padrão é `ios::in`. Veja também a seção 5.4.2.1 para os modos disponíveis.

No seguinte exemplo um objeto `ifstream` é aberto para leitura. O arquivo deve existir:

```
ifstream in("/tmp/scratch");
```

No lugar de associar diretamente um objeto `ifstream` com um arquivo, o objeto pode ser construído primeiro e aberto mais tarde.

- `void ifstream::open(char const *name, int mode)`: Tendo construído um objeto `ifstream` a função membro `open()` é usada para associar o `ifstream` com um arquivo.
- `ifstream::close()`: Para fechar um objeto `ifstream` explicitamente se usa a função membro `close()`. A função põe a flag `ios::fail` do objeto fechado. Um arquivo automaticamente fechado quando o objeto `ifstream` associado deixa de existir.

A seguinte é uma sutileza: Assuma que uma stream foi construída, mas não está ligada a um arquivo. P.ex., o comando `ifstream ostr` foi executado. Quando se examina seu estado com `good()` um valor não zero (i.e., ok) é retornado. O 'bom' estado indica que o objeto stream foi construído propriamente. Não significa que o arquivo também está aberto. Para examinar que uma stream está aberta use `ifstream::is_open()`: Se verdadeiro a stream está aberta. Veja também o exemplo da seção 5.4.2.

Para ilustrar a leitura de um arquivo binário (veja também a seção 5.5.1.1), um valor duplo é lido em forma binária de um arquivo no seguinte exemplo:

```
#include <fstream>
using namespace std;

int main(int argc, char **argv)
{
    ifstream f(argv[1]);
    double d;
```

```

        // lê um duplo em binário.
        f.read(reinterpret_cast<char *>(&d), sizeof(double));
    }

```

5.5.3: Entrada da memória: A classe `istream`

Para se ler informações da memória usando os recursos das streams se usa objetos `istream`. Estes objetos derivam dos objetos `istream`. Os seguintes construtores e membros estão disponíveis:

- `istream istr`: Constrói um objeto `istream` vazio. O objeto pode ser preenchido com informação a ser extraída mais tarde.
- `istream istr(string const &text)`: Constrói um objeto `istream` iniciado com o conteúdo do texto da string.
- `void istream::str(string const &text)`: Esta função membro guardará o conteúdo do texto da string no objeto `istream`, sobre escrevendo o conteúdo anterior.

Um objeto `istream` comumente é usado para converter textos em ASCII em seu equivalente binário, como a função da linguagem C `atoi()`. O seguinte exemplo ilustra o uso da classe `istream`, note especialmente o uso do membro `seekg()`:

```

#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main()
{
    istream istr("123 345"); // guarda algum texto.
    int x;

    istr.seekg(2);           // salta "12"
    istr >> x;               // extrai um inteiro
    cout << x << endl;       // escreve-o em cout
    istr.seekg(0);          // volta ao início
    istr >> x;               // extrai um int
    cout << x << endl;       // escreve-o em cout
    istr.str("666");         // guarda outro texto
    istr >> x;               // extrai um int
}

```

```

        cout << x << endl;           // escreve-o em cout
    }
    /*
        saída deste programa:
    3
    123
    666
    */

```

5.6: Manipuladores

Os objetos `ios` define um conjunto de flags de formato usado para determinar a maneira como os valores são inseridos (veja 5.3.2.1). As flags de formatação podem ser controladas por funções membro (veja seção 5.3.2.2), mas também os manipuladores o fazem. Os manipuladores são inseridos nas streams de saída ou extraídos das streams de entrada, no lugar de serem ativados pelo operador de seleção de membro (`'.'`).

Os manipuladores são funções. Novos manipuladores podem ser construídos também. A construção de manipuladores é vista na seo 9,10.1. Nessa seção os manipuladores disponíveis na biblioteca de E/S da linguagem C++ são discutidos. A maioria dos manipuladores afetam as flags de formato. Veja a seção 5.3.2.1 para detalhes sobre essas flags. A maioria dos manipuladores não suportam parâmetros. As fontes com manipuladores parametrizados precisam incluir: `#include <iomanip>`

- `std::boolalpha`: Este manipulador porá em verdadeiro a flag `ios::boolalpha`.
- `std::dec`: Este manipulador impõe a mostrar e ler números inteiros em formato decimal. Esta é a conversão padrão. A conversão é aplicada a valores inseridos em streams depois de processar os manipuladores. Por exemplo (veja também `std::hex` e `std::oct`, abaixo):

```

cout << 16 << ", " << hex << 16 << ", " << oct << 16;
// produz a saída:      16, 10, 20

```

- `std::endl`: Este manipulador insere um caracter de nova linha no bufer de saída e evacua o bufer depois.
- `std::ends`: Este manipulador insere um caracter de terminação da string no bufer de saída.
- `std::fixed`: Este manipulador porá em verdadeiro a flag `ios::fixed`.
- `std::flush`: Este manipulador evacua o bufer de saída.
- `std::hex`: Este manipulador impõe que os números inteiros sejam mostrados e lidos como

hexadecimais.

- `std::internal`: Este manipulador porá como verdadeiro a flag `ios::internal`.
- `std::left`: Este manipulador alinhará os valores à esquerda em campos amplos.
- `std::noboolalpha`: Este manipulador limpa a flag `ios::boolalpha`.
- `std::noshowpoint`: Este manipulador limpa a flag `ios::showpoint`.
- `std::noshowpos`: Este manipulador limpa a flag `ios::showpos`.
- `std::noshowbase`: Este manipulador porá em falso a flag `ios::showbase`.
- `std::noskipws`: Este manipulador porá em falso a flag `ios::skipws`.
- `std::nounitbuf`: Este manipulador interromperá a evacuação de uma stream de saída depois de cada operação de escritura. Com isto a stream só será evacuada com `flush`, `endl`, `unitbuf` ou quando for fechada.
- `std::nouppercase`: Este manipulador põe em falso a flag `ios::uppercase`.
- `std::oct`: Este manipulador impõe mostrar na tela e ler um número inteiro no formato octal.
- `std::resetiosflags(flags)`: Este manipulador chama `std::reset(flags)` para por em falso as flags indicadas.
- `std::right`: Este manipulador alinha os valores à direita em campos mais amplos que o necessário para o dado.
- `std::scientific`: Este manipulador põe a flag `ios::scientific` em verdadeiro.
- `std::setbase(int b)`: Este manipulador é usado para mostrar valores inteiros usando uma das bases 8, 10 ou 16. Pode ser usado como alternativa a `oct`, `dec` e `hex` em situações onde a base de valores inteiros é parametrizada.
- `std::setfill(int ch)`: Este manipulador define o caracter em situações onde o valor dos números são muito pequenos para preencher o campo usado para estes valores. Como padrão o espaço em branco é usado.

- `std::setiosflags(flags)`: Este manipulador chama `std::setf(flags)` pondo as flags indicadas em verdadeiro.
- `std::setprecision(int width)`: Este manipulador colocará a quantidade de casas decimais nos números fracionários. Em combinação com `std::fixed` pode ser usado para mostrar uma quantidade fixa de dígitos na parte fracionária dos tipos `float` e `double`:

```
cout << fixed << setprecision(3) << 5.0 << endl;
// mostra: 5.000
```

- `std::setw(int width)`: Este manipulador espera como argumento o tamanho do campo seguinte a ser inserido ou extraído. Pode ser como manipulador de inserção, onde define o número máximo de caracteres a serem inseridos em conjuntos de caracteres. Para prevenir extravasamento ao extrair de `cin`, `setv()` também pode ser usada:

```
cin >> setw(sizeof(array)) >> array;
```

Uma boa característica é que uma string longa que apareça em `cin` é subdividida em sub-strings com no máximo `sizeof(array)-1` caracteres e uma cadeia ASCII-Z é automaticamente apêndicula.

Notas:

- `setw()` é válida somente para o campo seguinte. Não atua como p.ex. `hex` que muda em geral o estado da stream de saída para mostrar números.
- Quando `setw(sizeof(someArray))` for usada deve-se ter atenção de que `someArray` é realmente um conjunto e não um apontador a um conjunto: o tamanho de um apontador, sendo p.ex, 4 bytes, em geral não é o tamanho do array para o qual aponta....
- `std::showbase`: Este manipulador põe como verdadeiro a flag `ios::showbase`.
- `std::showpoint`: Este manipulador valida a flag `ios::showpoint`.
- `std::showpos`: Este manipulador valida a flag `ios::showpos`.
- `std::skipws`: Este manipulador valida a flag `ios::skipws`.
- `std::unitbuf`: Este manipulador evacua uma stream de saída depois de cada operação de escrita.
- `std::uppercase`: Este manipulador valida a flag `ios::uppercase`.
- `std::ws`: Este manipulador remove todos os espaços em branco presentes na posição de leitura de

um bufer de entrada.

5.7: A classe *'streambuf'*

A classe streambuf define as seqüências de caracteres de entrada e saída que são processadas pelas streams. Como objeto ios, um objeto streambuf não é diretamente construído, mas implícito a objetos de outras classes que são especializações da classe streambuf.

A classe joga um importante papel na realização de possibilidades disponíveis como extensões aos padrões anteriores ao ANSI/ISO da linguagem C++. Contudo a classe não pode ser utilizada diretamente, seus membros são introduzidos aqui, já que no capítulo atual é o lugar mais lógico para tal. Para tanto, esta seção assume uma familiaridade básica com o conceito de polimorfismo, tópico discutido no Capítulo 14. Os leitores ainda não familiarizados com o conceito de polimorfismo podem, por agora, saltar esta seção sem perder a continuidade.

A razão primária para a existência da classe streambuf é separar as classes stream dos dispositivos sobre os que opera. O racional aqui é utilizar uma camada extra de software entre, de um lado, as classes que nos permitem comunicarmos com o dispositivo e por outro lado, a comunicação entre o software e os dispositivos. Assim, isto introduz uma cadeia de comandos vista regularmente nos projetos de software: A cadeia de comando é considerada um padrão genérico para a construção de software reusável, encontrado também nas, p.ex., pilhas TCP/IP. Um streambuf pode ser considerado outro exemplo do padrão de cadeia de comando: aqui o programa dialoga com os objetos stream, que por seu turno adiantam suas requisições aos objetos streambuf e estes em seu turno se comunicam com os dispositivos. Assim, como veremos logo, estamos em condições de fazer agora no software usuário aquilo que tinha que ser feito, antes, via (dispendioso) um sistema de chamadas.

A classe streambuf não possui construtor público, mas possui diversas funções membro públicas. Além destas funções membro públicas, diversas funções membro são acessíveis somente a classes especializadas. Estes membros protegidos estão listados nesta seção para referência posterior. Na seção 5.7.2 abaixo, é introduzida uma especialização particular da classe streambuf. Note que todos os membros públicos de streambuf discutidos aqui também estão disponíveis em filebuf

Na seção 14.6 o processo de construção de especializações de uma classe é discutido e no Capítulo 20 são mencionadas diversas implicações do uso dos objetos streambuf. Nos exemplos de cópia, redirecionamento, escritura e leitura de streams usando os membros da streambuf no capítulo atual são apresentados (seção 5.8).

A classe streambuf põe à disposição as seguintes funções membro. O tipo streamsize usado

abaixo, para todos os propósitos práticos é um inteiro sem sinal.

Membros públicos para operações de entrada:

- `streamsize streambuf::in_avail()`: Esta função membro retorna um limite inferior do número de caracteres que podem ser lidos imediatamente.
- `int streambuf::sbumpc()`: Esta função membro retorna o caracter seguinte ou EOF. O caracter é removido do objeto `streambuf`. Se não existe entrada disponível `sbumpc()` chamará o membro (protegido) `uflow()` (veja seção 5.7.1 abaixo) para aparecerem novos caracteres disponíveis. EOF só é retornado se não houver mais caracteres disponíveis.
- `int streambuf::sgetc()`: Esta função membro retorna o caracter seguinte ou EOF. O caracter não é removido do objeto `streambuf`.
- `int streambuf::sgetn(char *buffer, streamsize n)`: Esta função membro lê `n` caracteres do bufer de entrada e os guarda no buffer. O número de caracteres lidos é retornado. Esta função chama o membro (protegido) `xsgetn()` (veja seção 5.7.1 abaixo) para obter o número de caracteres requerido.
- `int streambuf::sngetc()`: Esta função membro remove o caracter atual do bufer de entrada e retorna o seguinte caracter disponível ou EOF. O caracter não é removido do objeto `streambuf`.
- `int streambuf::sputback(char c)`: Insere `c` como o caracter seguinte para ler do objeto `streambuf`. Deve-se tomar cuidado ao usar esta função: com frequência existe somente um caracter que pode ser posto de volta.
- `int streambuf::sungetc()`: Retorna o último caracter lido para o bufer de entrada, para ser lido outra vez pela próxima operação de entrada. Deve-se ter cuidado no uso desta função: frequentemente existe somente um caracter que pode ser posto de volta.

Membros públicos para operações de saída:

- `int streambuf::pubsync()`: Sincroniza (i.e., evacua) o bufer, escrevendo todas as informações pendentes disponíveis no bufer do `streambuf` no dispositivo. Normalmente somente usada por classes especializadas.
- `int streambuf::putc(char c)`: Insere `c` no objeto `streambuf`. Se, depois de escrever o caracter, o bufer está cheio a função chama a função membro (protegida) `overflow()` para evacuar o bufer para o dispositivo (veja seção 5.7.1 abaixo)

- `int streambuf::sputn(char const *buffer, streamsize n)`: Esta função membro insere `n` caracteres do bufer no objeto `streambuf`. O número de caracteres inseridos é retornado. Esta função chama o membro (protegido) `xsgputn()` (veja seção 5.7.1 abaixo)

Membros públicos para uma miscelânea de operações

- `pos_type streambuf::pubseekoff(off_type offset, ios::seekdir way, ios::openmode mode = ios::in | ios::out)`: zera o deslocamento do próximo caracter a ser lido ou escrito pondo `offset` relativo ao padrão `ios::seekdir` que indica o endereço da operação de deslocamento. Normalmente somente usada por classes especializadas.
- `pos_type streambuf::pubseekpos(pos_type offset, ios::openmode mode = ios::in | ios::out)`: Zera a posição absoluta do próximo caracter a ser lido ou escrito a `pos`. Normalmente somente usada por classes especializadas.
- `streambuf *streambuf::pubsetbuf(char* buffer, streamsize n)`: Define `buffer` como o bufer a ser usado pelo objeto `streambuf`. Normalmente só usada por classes especializadas.

5.7.1: Membros Protegidos de `streambuf`

Os membros protegidos da classe `streambuf` normalmente não são acessíveis. Contudo são acessíveis a classes especializadas derivadas de `streambuf`. São importantes para compreender e usar a classe `streambuf`. Normalmente existem ambos, dados membros protegidos e funções membros protegidas em uma classe. Como usar dados membros protegidos diretamente viola o princípio de encapsulamento, esses membros não são mencionados aqui. Como as funcionalidades de `streambuf`, disponíveis via suas funções membro, é bastante extensiva, usar diretamente seus dados membros é, provavelmente, dificilmente necessário. Esta seção não lista sequer todas as funções membro da classe `streambuf`. Somente são mencionados aqueles úteis na construção de especializações. A classe `streambuf` mantém um bufer de entrada e um de saída segundo a figura 4. A seguir nos referiremos seguidamente a essa figura.

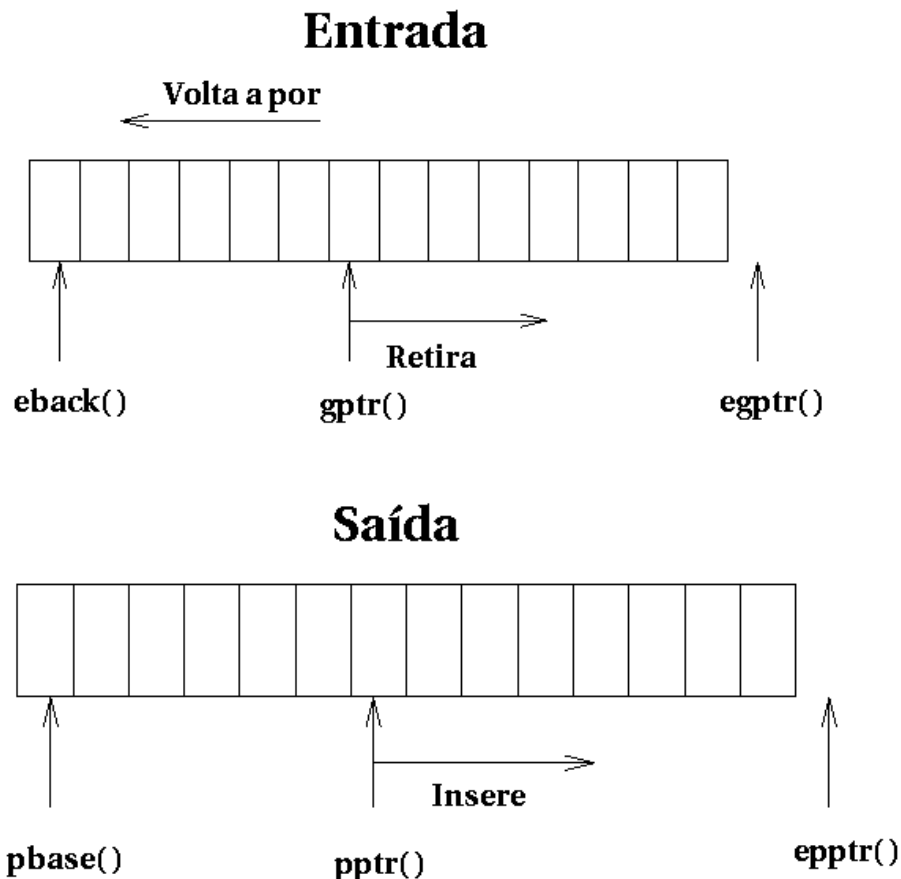


Figura 4 Búfers de Entrada e Saída e ponteiros da classe 'streambuf'

Construtores Protegidos:

- `streambuf::streambuf()`: Construtor (protegido) padrão da classe `streambuf`.

Diversas funções protegidas membro são relativas a operações de entrada. As funções membro marcadas como `virtual` serão redefinidas em classes derivadas, as funções redefinidas serão chamadas pelos objetos `i/ostream` que recebem os endereços de tais objetos de classes derivadas. Veja no Capítulo 14 para detalhes de funções membros virtuais. Eis os membros protegidos:

- `char *streambuf::eback()`: Para os búfers de entrada a classe `streambuf` mantém três apontadores: `eback()` aponta para a área 'fim de putback': os caracteres podem ser postos de volta com segurança nessa posição. Veja também a figura 4. `Eback()` pode ser considerado o início do búfer de entrada.
- `char *streambuf::egptr()`: Para o búfer de entrada a classe `streambuf` mantém três apontadores: `egptr()` aponta para a primeira posição depois do último carácter que pode ser retirado. Veja a

figura 4. Se `gptr()` (veja abaixo) for igual a `egptr()` o bufer necessita ser carregado (está vazio). Isto pode ser feito chamando-se `underflow()`, veja abaixo.

- `void streambuf::gbump(int n)`: Esta função move o apontador de entrada em `n` posições.
- `char *streambuf::gptr()`: Para o bufer de entrada a classe `streambuf` mantém três apontadores: `gptr()` aponta para o seguinte caracter a ser retirado. Veja a figura 4.
- `virtual int streambuf::pbackfail(int c)`: Esta função membro protegida pode ser redefinida por especializações da classe `streambuf` para algo inteligente quando a devolução de um caracter `c` falha. Uma das coisas a considerar aqui é restaurar o antigo valor do apontador de leitura quando o retorno de um caracter falha, porque se chegou ao início do bufer de entrada. Esta função é chamada quando falha a devolução de um caracter.
- `void streambuf::setg(char *beg, char *next, char *beyond)`: Esta função membro protegida inicia o bufer de entrada: põe o apontador 'beg' no início da área de entrada, o 'next' a apontar para o seguinte caracter a ser retirado e o 'beyond' a apontar para além do último caracter do bufer de entrada. Usualmente 'next' é pelo menos `beg+1`, para permitir a operação de devolução. Quando este membro é chamado com argumentos iguais a zero não se usa bufer de entrada (não sem argumentos, mas tendo valores iguais a 0.). Veja também o membro `streambuf::uflow()`, abaixo.
- `virtual streamsize streambuf::showmanyc()`: (Se pronuncia s-show-many-c) Esta função pode ser redefinida por especialização da classe `streambuf`. Retorna, com certeza, um limite inferior no número de caracteres que podem ser lidos do dispositivo antes que `uflow()` ou `underflow()` retorne EOF. Como padrão é retornado 0 (o que significa que pelo menos 0 caracter será retornado antes que uma das funções retorne EOF).
- `virtual int streambuf::uflow()`: para recarregar um bufer de entrada com novos caracteres. O padrão é chamar `underflow()`, veja abaixo, e para incrementar o apontador de leitura `gptr()`. Quando não é requerido bufer de entrada esta função em vez de chamar `underflow()` pode ser saltada para trazer o próximo caracter do dispositivo de leitura.
- `virtual int streambuf::underflow()`: Esta função pode ser redefinida por especialização da classe `streambuf` para ler outro caracter do dispositivo. O padrão é retornar EOF. Quando se usa bufer de entrada, freqüentemente todo o bufer não é refrescado, já que isto impossibilitaria a devolução de caracteres depois de um recarga. Este sistema de só refrescar parte do bufer é chamado 'split buffer'.
- `virtual streamsize streambuf::xsgetn(char *buffer, streamsize n)`: Esta função pode ser redefinida

por especialização da classe `streambuf` para retirar `n` caracteres do dispositivo. O padrão é chamar `sbumpc()` para cada caracter unitário. Como padrão chama (eventualmente) `underflow()` para cada caracter.

Eis as funções membro protegidas de operações de saída. Similarmente as funções de entrada algumas são virtuais: podem ser redefinidas em classes derivadas:

- `virtual int streambuf::overflow(int c)`: Esta função pode ser redefinida por especializações da classe `streambuf` para evacuar os caracteres no bufer de saída para o dispositivo de saída e então reavaliar os ponteiros, já que o bufer estará vazio. Se não se usa bufer de saída `overflow()` é chamada para cada um dos caracteres escrito no objeto `streambuf`. Isto é realizado zerando todos os ponteiros (usando, p.ex., `setp()`, veja abaixo). A implantação padrão retorna EOF, indicando que nenhum caracter pode ser escrito no dispositivo.
- `char *streambuf::pbase()`: Para o bufer de saída a classe `streambuf` mantém três ponteiros: `pbase()` aponta para o início da área do bufer de saída. Veja a figura 4.
- `char *streambuf::epptr()`: Para o bufer de saída a classe `streambuf` mantém três ponteiros: `epptr()` aponta para o primeiro local além do último caracter que pode ser escrito. Veja a figura 4. Se `pptr()` (veja abaixo) igual a `epptr()` o bufer precisa ser evacuado. Isto é feito chamando `overflow()`, veja abaixo.
- `void streambuf::pbump(int n)`: Esta função move o ponteiro de saída em `n` posições.
- `char *streambuf::pptr()`: Para o bufer de saída a classe `streambuf` mantém três ponteiros: `pptr()` aponta para o próximo caracter a ser escrito. Veja a figura 4.
- `void streambuf::setp(char *beg, char *beyond)`: Esta função inicia um bufer de saída: `'beg'` aponta para o início da área de saída e `'beyond'` aponta para a primeira posição depois do último caracter da área de saída. Para indicar que não é requerido bufer de saída os argumentos devem ser 0. Nesse caso a função `overflow()` é chamada para cada caracter para escrever no dispositivo.
- `streamsize streambuf::xsputn(char const *buffer, streamsize n)`: Esta função pode ser redefinida por especializações da classe `streambuf` para escrever `n` caracteres imediatamente no dispositivo. A implantação padrão chama `sputc()` para cada caracter individual, assim, redefinir só se for necessário mais eficiência.

Funções membro protegidas relativas ao gerenciamento do bufer e posicionamento:

- virtual streambuf *streambuf::setbuf(char *buffer, streamsize n): Esta função membro pode ser redefinida por especializações da classe streambuf para instalar um bufer. A implantação padrão não faz nada.

- virtual pos_type streambuf::seekoff(off_type offset, ios::seekdir way,

```
ios::openmode mode = ios::in | ios::out)
```

Esta função membro pode ser redefinida por especializações da classe streambuf para redefinir o valor do ponteiro 'next' de um bufer de entrada ou saída para uma nova posição relativa. A implantação padrão é para indicar falhas, retornando -1. A função é chamada quando, p.ex., tellg() ou tellp() é chamada. Quando a especialização suporta reposicionamento, então a especialização pode definir também esta função para determinar o que fazer com uma requisição de reposicionamento (ou tellp()/tellg()).

- virtual pos_type streambuf::seekpos(pos_type offset,

```
ios::openmode mode = ios::in | ios::out):
```

Esta função membro pode ser redefinida por especializações da classe streambuf para revalorizar o ponteiro 'next' em buffers de entrada ou saída para uma nova posição. A implantação padrão é para indicar falha retornando -1.

- virtual int sync(): Esta função pode ser redefinida por especializações da classe streambuf para evacuar o bufer de saída para o dispositivo ou para reposicionar o dispositivo de entrada para a posição do último caracter consumido. A implantação padrão (não usando bufer) é para retornar 0, indicando sucesso no reposicionamento. A função membro é usada para assegurar que qualquer caracter que ainda esteja no bufer seja escrito no dispositivo ou voltar ao dispositivo os caracteres não consumidos quando o objeto streambuf deixe de existir.

Moral: Quando uma especialização da classe streambuf é projetada, a última coisa a fazer é redefinir underflow() em especializações onde se lê informações de dispositivos e redefinir overflow() em especializações que escrevem informações em dispositivos. Muitos exemplos de especializações da classe streambuf serão dados na sequência.

Os objetos da classe fstream usam um bufer combinado de entrada/saída. Isto resulta do fato de que istream e ostream são virtualmente derivadas de ios, que contém a streambuf. Como vamos ver na seção 14.4.2, isto implica que as classes derivadas de istream e ostream compartilham seus ponteiros streambuf. Para se construir uma classe que possua buffers de entrada e saída separados a streambuf deve definir internamente dois buffers. Quando seekoff() é chamada para leitura, seu parâmetros de modo é posto em ios::in, do contrário em ios::out. Assim a especialização de streambuf sabe se deve acessar o bufer de entrada ou saída. Claro que underflow() e overflow() já sabem sobre qual bufer operar.

5.7.2: A classe 'filebuf'

A classe 'filebuf' é uma especialização da streambuf usada pelas classes 'file'. Além dos membros (públicos) disponíveis na classe streambuf estão definidos os seguintes extra membros (públicos):

- `filebuf::filebuf()`: Como a classe possui um construtor, a diferença da classe streambuf, é possível construir objetos filebuf. Esta função define um objeto filebuf pleno, ainda não conectado a uma stream.
- `bool filebuf::is_open()`: Esta função membro retorna verdadeiro se o objeto filebuf estiver conectado a um arquivo aberto. Veja a função membro `open()` abaixo.
- `filebuf *filebuf::open(char const *name, ios::openmode mode)`: Esta função membro associa o objeto filebuf a um arquivo cujo nome é dado. O arquivo é aberto de acordo com `ios::openmode`.
- `filebuf *filebuf::close()`: Esta função membro fecha a associação entre filebuf e o arquivo. A associação é automaticamente fechada quando o objeto deixa de existir.

Antes de se poder definir objetos filebuf a seguinte diretiva ao preprocessador deve ser dada:

```
#include <fstream>
```

5.8: Tópicos Avançados

5.8.1: Copiando streams

Usualmente os arquivos são copiados lendo de um arquivo fonte caracter por caracter ou linha por linha. O molde básico para processar arquivos é como segue:

- Num laço eterno:
 - lê um caracter
 - se erro de leitura (i.e., `fail()` retorna verdadeiro), `break` sai do laço
 - processa o caracter

É importante notar que a leitura deve preceder o teste, já que só é possível saber se depois de uma leitura do arquivo houve sucesso ou não. Claro que são possíveis variações:

`getline(istream &, string &)` (veja seção 5.5.1) retorna uma `istream & itself`, aqui a leitura e o teste podem ser feitos numa só expressão. Contudo, o modelo acima representa o caso geral. Assim, o seguinte programa pode ser usado para copiar `cin` para `cout`:

```
#include <iostream>

using namespace::std;

int main()
{
    while (true)
    {
        char c;
        cin.get(c);
        if (cin.fail())
            break;
        cout << c;
    }
    return 0;
}
```

Combinando o `get()` com o comando `if` uma construção com `getline()` pode ser usada:

```
if (!cin.get(c))
    break;
```

Note, contudo, que ainda segue a regra básica: 'ler primeiro, testar depois'.

Mas esta forma simples de cópia de um arquivo não é requerida frequentemente. Mais frequente são situações que um arquivo é processado até certo ponto e o resto do arquivo pode ser copiado sem alterações. O seguinte programa ilustra esta situação: A chamada a `ignore()` é usada para saltar a primeira linha (para o objetivo do exemplo assume-se que a primeira linha tem no máximo 80 caracteres), o segundo comando utiliza uma versão sobrecarregada do operador `<<`, na qual um ponteiro `streambuf` é inserido em outra `stream`. Como o membro `rddbuf()` retorna um `streambuf *`, pode ser inserido em `cout`. Assim copia o resto de `cin` a `cout`:

```
#include <iostream>

using namespace std;

int main()
{
```

```

        cin.ignore(80, '\n');    // salta a primeira linha
        cout << cin.rdbuf();    // copia o resto inserindo um streambuf *
    }

```

Note que este método usa um objeto `streambuf`, portanto funciona para qualquer especialização de `streambuf`. Conseqüentemente se a classe `streambuf` é especializada para um dispositivo particular pode ser inserido em outra stream usando o método acima.

5.8.2: Acoplando streams

As `Ostreams` podem ser acopladas a objetos `ios` usando a função membro `tie()`. Como resultado disto toda saída buferizada no objeto `ostream` será evacuada (chamando `flush()`) sempre que uma operação de entrada ou saída é feita sobre o objeto `ios` ao qual a `ostream` está acoplado. Como padrão `cout` está acoplado a `cin` (i.e., `cin.tie(cout)`): Sempre que uma operação é feita sobre `cin`, primeiro `cout` é evacuada. Para romper o acoplamento a função membro `ios::tie(0)` deve ser chamada.

Outro (frequentem útil), mas não padrão) exemplo de acoplamento entre streams é acoplar `cerr` a `cout`: Desta forma a saída estandarte entrega uma mensagem de erro na tela em sincronia com o momento em que foram geradas:

```

#include <iostream>
using namespace std;

int main()
{
    cout << "primeiro (buferizado) linha para cout\n";
    cerr << "primeiro ( não buferizado) linha para cerr\n";

    cerr.tie(&cout);

    cout << "segundo (buferizado) linha para cout\n";
    cerr << "segundo (não buferizado) linha para cerr\n";
}
/*
    Saída gerada:
primeiro (não buferizado) linha para cerr
primeiro (buferizado) linha para cout
segundo (buferizado) linha para cout
segundo (não buferizado) linha para cerr
*/

```

Um meio alternativo para acoplar streams é fazer que usem um objeto `streambuf` comum. Isto

pode ser feito usando a função membro `ios::rdbuf(streambuf *)`. Assim duas streams podem usar seu próprio formato, uma stream para entrada e outra para saída e redirecionando com a biblioteca `iostream` em lugar de usar chamadas ao sistema operacional. Para exemplos veja a próxima seção.

5.8.3: Redirecionando streams

Usando o membro `ios::rdbuf()` pode-se compartilhar seus objetos `streambuf`. Isto significa que a informação escrita numa stream será escrita também na outra, fenômeno normalmente chamado de redireção. A redireção normalmente é feita no nível do sistema operacional e em algumas situações isto ainda é necessário (veja a seção 20.3.1).

Uma situação estandarte onde é desejável a redireção é escrever mensagens de erro diretamente num arquivo no lugar de escrever no arquivo próprio de erros, normalmente indicado pelo seu descritor de arquivo número 2. No sistema operacional Unix usando bash shell isto pode ser realizado assim:

```
program 2>/tmp/error.log
```

Com este comando qualquer mensagem de erro será escrita pelo programa será salva no arquivo `/tmp/error.log` em lugar da tela.

Eis como isto pode ser realizado usando objetos `streambuf`. Assume-se que o programa espera um argumento opcional que define o nome do arquivo para escrever a mensagem de erro; assim o programa chama:

```
program /tmp/error.log
```

Eis o programa que realiza a redireção:

```
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream errlog;                // 1
    streambuf *cerr_buffer = 0;     // 2

    if (argc == 2)
    {
```

```

        errlog.open(argv[1]); // 3
        cerr_buffer = cerr.rdbuf(errlog.rdbuf()); // 4
    }
    else
    {
        cerr << "Faltando o nome do arquivo de log\n";
        return 1;
    }

    cerr << "Diversas mensagens a stderr, msg 1\n";
    cerr << "Diversas mensagens a stderr, msg 2\n";

    cout << "Inspeciona o conteúdo de " <<
        argv[1] << "... [Enter] ";
    cin.get(); // 5

    cerr << "Diversas mensagens a stderr, msg 3\n";

    cerr.rdbuf(cerr_buffer); // 6
    cerr << "Feito\n"; // 7
}
/*
    Saída gerada no arquivo argv[1]

    em cin.get():

    Diversas mensagens a stderr, msg 1
    Diversas mensagens a stderr, msg 2

    no fim do programa:

    Diversas mensagens a stderr, msg 1
    Diversas mensagens a stderr, msg 2
    Diversas mensagens a stderr, msg 3
*/

```

- Nas linhas 1 e 2 as variáveis são definidas: `errlog` é a ofstream onde escrever as mensagens de erro também e `cerr_buffer` é um ponteiro a um streambuf para apontar ao `cerr_buffer` original. Isto é discutido abaixo.
- Na linha 3 a stream de erro alternativa é aberta.
- Na linha 4 a redireção tem lugar: `cerr` escreverá no streambuf definido com `errlog`. O importante é que o bufer original usado por `cerr` é salvo, como explicado abaixo.
- Na linha 5 há uma pausa. Neste ponto duas linhas são escritas no arquivo alternativo de erro,

Temos a oportunidade de dar uma olhada em seu conteúdo: Existem, em verdade, duas linhas escritas no arquivo.

- Na linha 6 a redireção é terminada. É muito importante, já que o objeto `errlog` é destruído no fim de `main()`. Se o bufer de `cerr` restaurado então neste ponto `cerr` poderia se referir a um objeto `streambuf` inexistente, o que poderia produzir efeitos inesperados. É responsabilidade do programador assegurar-se de que um original do `streambuf` está salvo antes do redirecionamento e restaurá-lo quando este termine.
- Finalmente, na linha 7 é escrito `Feito na tela`, já que o redirecionamento terminou.

5.8.4: Lendo e Escrevendo em streams

Para se ler e escrever numa stream um objeto `fstream` deve ser criado. Como com objetos `ifstream` e `ofstream` seu construtor recebe o nome do arquivo a ser aberto: `fstream inout ("arquivo.es", ios::in | ios::out);`

Note o uso das constantes `ios::in` e `ios::out`, indicnado que o arquivo deve ser aberto para leitura e escritura. Indicadores de múltiplos modos é usado, concatenados pelo operador binário `|`. Alternativamente no lugar de `ios::out`, `ios::app` poderia ter sido usado, nesse caso a escritura seria sempre feita no fim do arquivo. A leitura e escritura a um arquivo é algo desajeitado: O que fazer quando o arquivo pode ou não exestir, mas quando existe não pode ser escrito?

Estive me debatendo com este problema algum tempo e atualmente uso este modelo:

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    fstream rw("fname", ios::out | ios::in);
    if (!rw)
    {
        rw.clear();
        rw.open("fname", ios::out | ios::trunc | ios::in);
    }
    if (!rw)
    {
```

```

        cerr << "Abrindo `fname' falhou miseravelmente" << endl;
        return 1;
    }

    cerr << rw.tellp() << endl;

    rw << "Olá Mundo" << endl;
    rw.seekg(0);

    string s;
    getline(rw, s);

    cout << "Read: " << s << endl;
}

```

No exemplo acima o construtor falha quando o arquivo 'fname' não existe ainda. Nesse caso a função `open()` consegue já que o arquivo é criado devido à flag `ios::trunc`. Se o arquivo já existe o construtor terá êxito. Se a flag `ios::ate` poderia ter sido especificada também com a construção inicial de `rw`, a primeira operação de leitura ou escritura teria sido a partir de EOF. Contudo, `ios::ate` não é `ios::app`, e será até que seja possível reposicionar `rw` usando `seekg()` ou `seekp()`.

Sob o sistema operativo DOS, que usa o caracter `\r\n` para separar linhas em arquivos de texto a flag `ios::binary` é requerido para processar arquivos binários para assegurar que combinações do tipo `\r\n` sejam processadas como dois caracteres separados.

Com objetos `fstream` as combinações de flags de arquivos são usadas para assegurar que uma stream seja ou não (re)criada vazia quando aberta. Veja a seção 5.4.2.1 para detalhes.

Uma vez que o arquivo seja aberto em modo leitura e escritura, o operador `<<` pode ser usado para inserir informação ao arquivo, enquanto o operador `>>` pode ser usado para inserir informação no arquivo. Essas operações podem ser feitas em ordem aleatória. O seguinte fragmento lerá um delimitador espaço em branco do arquivo e escreverá, então, uma string no arquivo, justamente depois do ponto onde a string lida terminou, seguida pela leitura de outra string logo após o local onde a string recém escrita terminou:

```

fstream f("filename", ios::in | ios::out | ios::trunc);
string str;

f >> str;           // lê a primeira palavra

f << "Olá Mundo";    // escreve um texto bem conhecido
f >> str;           // e lê outra vez

```

Como os operadores << e >> aparentemente podem ser usados com objetos fstream, seria maravilhoso no lugar de uma série de operadores << e >> usar um só se possível. Depois de tudo, f >> str produz uma fstream &, não é assim?

A resposta é: Não. O compilador converte o objeto fstream num objeto ifstream em combinação com o operador extração e num objeto ofstream em combinação com o operador inserção. Conseqüentemente, um comando como:

```
f >> str << "grandpa" >> str;
```

resulta num erro de compilação como:

```
no match for `operator <<(class istream, char[8])`
```

Já que o compilador não aceita a classe istream, o objeto fstream, aparentemente, é considerado um objeto ifstream em combinação com o operador extração.

Claro, inserções e extrações aleatórias são pesadamente usadas. Geralmente as inserções e extrações têm lugar em posições específicas do arquivo. Nesses casos, a posição onde a inserção ou extração terá lugar é controlada e monitorada pelas funções membro seekg() e tellg() (veja as seções 5.4.1.2 e 5.5.1.2).

As condições de erro (veja 5.3.1) devido, p.ex., ler além do fim do arquivo, encontro do fim de arquivo, posicionar antes do início do arquivo, podem ser limpas usando-se a função membro clear(). Depois de clear() o processamento pode prosseguir. Exemplo:

```
fstream f("filename", ios::in | ios::out | ios::trunc);
string str;

f.seekg(-10); // aqui falha, mas...
f.clear();    // o processamento de f continua

f >> str;     // lê a primeira palavra
```

Em aplicativos de bases de dados a necessidade de leitura e escritura em arquivos é comum, aí os arquivos consistem em gravações de tamanho fixo e a localização e tamanho de cada parte de informação é bem conhecido. Por exemplo, o seguinte programa pode ser usado para acrescentar linhas de texto a um arquivo (possivelmente existente) e retirar certas linhas baseando-se em seu número de ordem dentro do arquivo. Note o uso de índice binário num arquivo para retirar o primeiro byte de uma linha.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```

```

void err(char const *msg)
{
    cout << msg << endl;
    return;
}

void err(char const *msg, long value)
{
    cout << msg << value << endl;
    return;
}

void read(fstream &index, fstream &strings)
{
    int idx;

    if (!(cin >> idx))                // lê o índice
        return err("line number expected");

    index.seekg(idx * sizeof(long)); // vai ao deslocamento do índice

    long offset;

    if
    (
        !index.read                    // lê o deslocamento da linha
        (
            reinterpret_cast<char *>(&offset),
            sizeof(long)
        )
    )
        return err("no offset for line", idx);

    if (!strings.seekg(offset))        // vai ao deslocamento da linha
        return err("can't get string offet ", offset);

    string line;

    if (!getline(strings, line))       // lê a linha
        return err("no line at ", offset);

    cout << "Got line: " << line << endl;    // mostra a linha
}

void write(fstream &index, fstream &strings)

```

```

{
    string line;

    if (!getline(cin, line))                // lê a linha
        return err("line missing");

    strings.seekp(0, ios::end);             // para strings
    index.seekp(0, ios::end);              // para índice

    long offset = strings.tellp();

    if
    (
        !index.write                        // escreve o deslocamento para
o índice
        (
            reinterpret_cast<char *>(&offset),
            sizeof(long)
        )
    )
        err("Writing failed to index: ", offset);

    if (!(strings << line << endl))         // escreve a linha
        err("Writing to `strings' failed");
        // confirma a escritura da linha
    cout << "Write at offset " << offset << " line: " << line <<
endl;
}

int main()
{
    fstream index("index", ios::trunc | ios::in | ios::out);
    fstream strings("strings", ios::trunc | ios::in | ios::out);

    cout << "enter `r <number>' to read line <number> or "
        "w <line>' to write a line\n"
        "or enter 'q' to quit.\n";

    while (true)
    {
        cout << "r <nr>, w <line>, q ? ";    // show prompt

        string cmd;

        cin >> cmd;                          // lê o cmd

        if (cmd == "q")                      // processa o cmd.

```

```

        return 0;

        if (cmd == "r")
            read(index, strings);
        else if (cmd == "w")
            write(index, strings);
        else
            cout << "Unknown command: " << cmd << endl;
    }
}

```

Outro exemplo de leitura e escrita em arquivos. Considere o seguinte programa, que também serve como ilustração de leitura em arquivos com delimitador de string ASCII-Z:

```

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    // l/e no arquivo
    fstream f("hello", ios::in | ios::out | ios::trunc);

    f.write("Olá", 6); // escreve 2 ascii-z
    f.write("Olá", 6);

    f.seekg(0, ios::beg); // vai ao início do arquivo

    char buffer[100]; // ou: char *buffer = new
char[100]
    char c;

    // lê o primeiro `Olá'
    cout << f.get(buffer, sizeof(buffer), 0).tellg() << endl;;
    f >> c; // lê o delimitador ascii-z

    // e lê o segundo `Olá'
    cout << f.get(buffer + 6, sizeof(buffer) - 6, 0).tellg() << endl;

    buffer[5] = ' '; // muda ascii-z para ' '
    cout << buffer << endl; // mostra 2 vezes `Olá'
}
/*
Saída gerada:
5
11
Olá Olá
*/

```


Um meio completamente diferente para ler e escrever numa stream pode ser implantado usando membros de streambuf dos objetos stream. Todas as considerações feitas anteriormente permanecem válidas: Antes de uma operação de leitura seguida de uma operação de escritura, é indispensável, entre elas, uma operação seekg() e antes de uma operação de escritura seguida por uma operação de leitura, entre elas deve vir uma operação seekp(). Quando os objetos streambuf de uma stream são usados, de duas uma, ou uma istream está associada ao objeto streambuf de outro objeto ostream ou, vice-versa, um objeto ostream está associado com o objeto streambuf de outro objeto istream. Eis o mesmo programa anterior, agora usando associação de streams:

```
#include <fstream>
#include <string>
using namespace std;

void err(char const *msg)
{
    cout << msg << endl;
    return;
}

void err(char const *msg, long value)
{
    cout << msg << value << endl;
    return;
}

void read(istream &index, istream &strings)
{
    int idx;

    if (!(cin >> idx))                // lê o índice
        return err("esperado o número da linha");

    index.seekg(idx * sizeof(long));  // vai ao deslocamento do
índice

    long offset;

    if
    (
        !index.read                    // lê o deslocamento da
linha
        (
            reinterpret_cast<char *>(&offset),
            sizeof(long)
        )
    )
```

```

        return err("não há deslocamento para a linha", idx);

        if (!strings.seekg(offset))                // vai ao deslocamento da
linha
            return err("não pode obter o deslocamento da string ",
offset);

        string line;

        if (!getline(strings, line))                // lê a linha
            return err("não há linha em ", offset);

        cout << "Obteve a linha: " << line << endl;    // mostra a
linha
    }

void write(ostream &index, ostream &strings)
{
    string line;

    if (!getline(cin, line))                // lê a linha
        return err("line missing");

    strings.seekp(0, ios::end);                // para a strings
    index.seekp(0, ios::end);                // para o índice

    long offset = strings.tellp();

    if
    (
        !index.write                            // escreve o deslocamento do
índice
        (
            reinterpret_cast<char *>(&offset),
            sizeof(long)
        )
    )
        err("A escritura do índice falhou: ", offset);

    if (!(strings << line << endl))                // escreve a linha
        err("Writing to `strings' failed");

        // confirma a escritura da
linha

    cout << "Escreve no deslocamento " << offset << " line: " << line
<< endl;
}

```

```

int main()
{
    ifstream index_in("índice", ios::trunc | ios::in | ios::out);
    ifstream strings_in("strings", ios::trunc | ios::in | ios::out);
    ostream index_out(index_in.rdbuf());
    ostream strings_out(strings_in.rdbuf());

    cout << "entre `r <número>' para ler o <número> da linha ou "
           "w <linha>' para escrever uma
linha\n"
           "ou entre `q' para sair.\n";

    while (true)
    {
        cout << "r <nr>, w <linha>, q ? ";          // mostra opções

        string cmd;

        cin >> cmd;                                  // lê o cmd

        if (cmd == "q")                              // processa o cmd.
            return 0;

        if (cmd == "r")
            read(index_in, strings_in);
        else if (cmd == "w")
            write(index_out, strings_out);
        else
            cout << "Comando desconhecido: " << cmd << endl;
    }
}

```

Por favor note:

- As streams para se associarem aos objetos streambuf não são objetos ifstream ou ofstream (ou objetos istringstream ou ostringstream, se interessar), mas objetos istream e ostream.
- As streams para se associarem a objetos streambuf de streams existentes não são objetos istream ou ofstream (ou istringstream ou ofstringstream), mas objetos istream ou ostream básicos.
- O objeto streambuf não tem que ser definido num objeto ifstream ou ofstream: é definido fora das streams usando construções como:

```

filebuf fb("índice", ios::in | ios::out | ios::trunc);
istream index_in(&fb);

```

```
ostream index_out(&fb);
```

- Note que um objeto ifstream pode ser construído usando modos de stream normalmente usados para escrever em arquivos. Objetos ofstream podem ser construídos usando modos de stream normalmente usados para ler de arquivos.
- Se uma istream e uma ostream estão associadas a um streambuf comum, então a leitura e escritura os ponteiros (podem) apontar para os mesmos lugares: estão intimamente acoplados.
- A vantagem de usar streambufs separados com um objeto fstream é (claro está) que abre a possibilidade de usar objetos stream com objetos streambuf especializados. Estes objetos streambuf, então, podem ser construídos para interfacear dispositivos particulares. A elaboração disto é deixada como exercício ao leitor.

Capítulo 6: Classes

Neste capítulo as classes são introduzidas formalmente. Duas funções membro especiais, o construtor e o destrutor, são apresentados.

Em degraus construiremos uma classe Pessoa, que pode ser usada num aplicativo de base de dados para armazenar nomes, endereços e números telefônicos de pessoas.

Começemos criando a declaração de uma classe Pessoa imediatamente. A declaração da classe normalmente está no arquivo cabeçalho da classe, p.ex., pessoa.h. Uma declaração de uma classe normalmente não se chama declaração. O nome comum para declaração de uma classe é interface de classe, para se distinguir das definições das funções membro, chamada implantação da classe. Assim, a interface da classe Pessoa é dada a seguir:

```
#include <string>

class Pessoa
{
    std::string d_nome;      // nome da pessoa
    std::string d_endereço;  // campo do endereço
    std::string d_fone;      // número telefônico
    size_t      d_peso;      // o peso em kg.

public:                      // funções de interface
    void setNome(string const &n);
    void setEndereço(string const &a);
    void setFone(string const &p);
    void setPeso(size_t peso);

    string const &nome() const;
    string const &endereço() const;
    string const &fone() const;
    size_t peso() const;
};
```

Pode-se notar que esta terminologia é freqüentemente empregada de modo relaxado. Às vezes a definição de classe é usada para indicar a interface de classe. Enquanto que a definição de classe (digo, a interface) contém as declarações de seus membros, a implantação desses membros é também referida como definição desses membros. Como o conceito de interface de classe e implantação de classe são bem distintos, deve ficar claro do contexto o que é oculto por 'definição'.

Os campos de dados nesta classe são d_nome, d_endereço, d_fone e d_peso. Todos os

campos (exceto `d_peso`) são objetos string. Como aos campos de dados não é dado um modificador de acesso, são privados, o que significa que só podem ser acessados pelas funções da classe Pessoa. Alternativamente a etiqueta 'private:' poderia ter sido usada no início de uma seção privada da definição de classe.

Os dados são manipulados pela função interface que cuida de toda comunicação com o código fora da classe. Seja para por os campos de dados em determinados valores (p.ex., `setnome()`) ou inspecionar os dados (p.ex., `nome`). As funções que somente retornam valores guardados dentro dos objetos, não permitindo que quem chama modifique esses dados internos, são chamadas funções acessoras.

Note uma vez mais as similitudes entre classe e estrutura. A diferença fundamental é que as classes, como padrão, possuem membros privados, enquanto as estruturas possuem membros públicos. Desde que a convenção é que os membros públicos apareçam primeiro, a palavra chave 'private:' é necessária para voltar de membros públicos à situação (padrão) privada.

Algumas notas enquanto ao estilo. Seguindo a Lakos (Lakos, J., 2001)

Large-Scale C++ Software Design (Addison-Wesley). I

sugeri a seguinte implantação de interfaces de classe:

- Todos os membros de dados têm permissões de acesso privado e devem ser postos no início da interface.
- Todos os membros de dados começam por `d_` seguido por um nome sugestivo de seu conteúdo (no Capítulo 10), encontramos também membros de dados iniciando por `s_`.
- Existem membros não privados, mas nos sentimos hesitantes em usar permissões de acesso a membros de dados não privados (veja Capítulo 13).
- Duas classes amplas de funções membro são as funções manipuladoras e acessoras. As manipuladoras permitem aos usuários modificar os dados internos dos objetos. Por convenção começam com `set`. P.ex, `setNome()`.
- Nas acessoras freqüentemente se encontra o prefixo `get`, p.ex., `getNome()`. Contudo, segundo a convenção usada em Ot Graphical User Interface Toolkit (veja <http://www.trolltech.com>), o prefixo `get` caiu. Portanto, no lugar de definir o membro `getEndereço()`, definiremos `endereço()`.

As convenções de estilo geralmente tomam tempo em se desenvolverem. Não há nada

obrigatório respeito a elas, contudo. Sugiro aos leitores compelidos a não usarem que usem as suas próprias. Todos os outros podem adotar as convenções de estilo acima.

6.1: O construtor

Uma classe em C++ pode conter duas categorias de funções membro que estão envolvidas no trabalho interno da classe. Estas categorias de funções membro são, de um lado, as construtoras, de outro, as destrutoras. A principal tarefa das destrutoras é liberar a memória alocada pelo objeto quando este 'sai de escopo'. A alocação de memória é discutida no

e as destrutoras serão então discutidas mais profundamente.

Neste capítulo a ênfase será sobre a forma básica da classe e sobre seus construtores.

O construtor tem por definição o mesmo nome que sua classe. O construtor não especifica um valor de retorno, nem mesmo void. P.ex., para a classe Pessoa o construtor é Pessoa::Pessoa(). O sistema de execução da linguagem C++ assegura que o construtor da classe, se definido, seja chamado quando uma variável da classe, chamada um objeto, seja definida ('criada'). Claro que é possível definir a classe sem construtor nenhum. Nesse caso o programa chamará o construtor padrão quando um objeto correspondente for criado. O que acontece nesse caso depende de como a classe foi definida. A ação do construtor padrão será vista na seção 6.4.1.

Os objetos podem ser definidos localmente ou globalmente. Contudo, em C++ a maioria dos objetos são definidos localmente. Objetos definidos globalmente dificilmente são requeridos.

Quando um objeto é definido localmente (numa função), o construtor é chamado cada vez que a função é chamada. O construtor de objetos é então ativado no ponto onde o objeto é definido (uma sutileza aqui é que uma variável pode ser definida implicitamente como, p.ex., uma variável temporária numa expressão).

Quando um objeto é definido como estático (i.e., é uma variável estática) na função, o construtor é chamado quando a função, onde está a variável estática, é chamada pela primeira vez.

Quando um objeto é definido como global o construtor é chamado mesmo antes do programa começar. Note que neste caso o construtor é chamado antes mesmo que a função main() comece. Esta

característica é ilustrada no seguinte programa:

```
#include <iostream>
using namespace std;

class Demo
{
    public:
        Demo();
};

Demo::Demo()
{
    cout << "Construtor Demo chamado\n";
}

Demo d;

int main()
{}

/*
    Saída gerada:
    Construtor Demo chamado
*/
```

A listagem acima mostra como a classe Demo é definida, consiste de uma única função: o construtor. O construtor realiza uma única ação: imprime uma mensagem. O programa contém um objeto global da classe Demo e main() que tem um corpo vazio. Apesar de tudo o programa produz uma saída.

Algumas importantes características dos construtores são:

- O construtor tem o mesmo nome que sua classe.
- A função primária do construtor é assegurar que todos os membros de dados da classe sejam acessíveis ou pelo menos com valores definidos uma vez que o objeto seja construído. Voltaremos a esta importante tarefa rapidamente.
- O construtor não retorna nenhum valor. Isto é verdade também para a declaração do construtor na definição da classe, como em:

```
class Demo
{
    public:
        Demo();           // não retorna nenhum valor aqui
};
```


e é verdade para a definição da função construtora, como em:

```
Demo::Demo()           // não retorna nenhum valor aqui
{
    // comandos ...
}
```

- A função construtora no exemplo acima não possui argumentos. É chamada construtor padrão. Tal construtor não possui argumentos, contudo, não é requisito per se. Veremos em breve que é possível definir argumentos ao construtor como não defini-los.
- NOTA: Uma vez definido um construtor com argumentos, o construtor padrão já não existe, a menos que o construtor padrão também seja definido explicitamente.

Isto tem importantes consequências, como o construtor padrão é requerido em casos onde seja capaz de construir um objeto com ou sem iniciação explícita de valores. Definindo um construtor com somente um argumento, o construtor padrão implícito desaparece de vista. Como foi chamada a atenção, para torná-lo disponível outra vez deve ser definido explicitamente também.

6.1.1: Um Primeiro Aplicativo

Como está ilustrado no início deste capítulo, a classe Pessoa contém três strings privadas como membros de dados e um inteiro sem sinal `d_peso` como membro de dados. Estes membros de dados podem ser manipulados pelas funções interface.

As classes (podem) operam como segue:

- Quando o objeto é construído seus membros de dados se tornam valores 'sensíveis'. Dessa forma os objetos nunca sofrem de falta de inicialização.
- A adjudicação a um membro de dados (usando uma função `set...()`) consiste na adjudicação do novo valor ao membro de dados correspondente. Esta adjudicação é completamente controlada pelo projetista da classe. Conseqüentemente o objeto é 'responsável' por sua própria integridade.
- A inspeção dos membros de dados usando uma função acessora simplesmente retorna o valor do membro de dados requerido. Outra vez, isto não terá como resultado modificações descontroladas de objetos de dados.

As funções `set...()` podem ser construídas assim:

```
#include "pessoa.h"                                // dado anteriormente

// funções interface set...()
void Pessoa::setNome(string const &nome)
{
    d_nome = nome;
}

void Pessoa::setEndereço(string const &endereço)
{
    d_endereço = endereço;
}

void Pessoa::setfone(string const &fone)
{
    d_fone = fone;
}

void Pessoa::setPeso(size_t peso)
{
    d_peso = peso;
}
```

Em seguida define-se as funções acessoras. Note a ocorrência da palavra chave `const` seguindo a lista de parâmetros destas funções: Estas funções membro são chamadas funções membro constantes, indicando que não modificarão seus objetos de dados quando chamadas. Ainda mais, note que os tipos retornados pelas funções membro são membros de dados `string` também são do tipo `string const &`: o `const` aqui indica que quem chama a função membro não pode alterar o valor retornado. Quem chama uma função acessora pode copiar a variável retornada e esse valor pode então ser modificado livremente. Funções membro constantes serão discutidas em grande detalhe na seção 6.2. O valor retornado pela função membro `peso()` é um inteiro sem sinal pleno, já que é uma simples cópia do valor guardado no membro `peso` de `Pessoa`:

```
#include "pessoa.h"                                // dado anteriormente

// funções ...() acessoras
string const &Pessoa::nome() const
{
    return d_nome;
}

string const &Pessoa::endereço() const
{
    return d_endereço;
}
```

```

string const &Pessoa::fone() const
{
    return d_fone;
}

size_t Pessoa::peso() const
{
    return d_peso;
}

```

A definição da classe Pessoa dada acima pode ainda ser usada. As funções set()... e acessoras simplesmente implanta as funções membro declaradas na definição da classe.

O exemplo seguinte mostra o uso da classe Pessoa. Um objeto é iniciado e passa a uma função printpessoa(), que imprime os dados da pessoa. Note também o uso do operador referência & na lista de argumentos da função printpessoa(). Dessa forma somente uma referência a um objeto Pessoa é passada no lugar de todo o objeto. O fato de que printpessoa() não modifica seu argumento é evidente do fato que o parâmetro é declarado const.

Alternativamente a função printpessoa() poderia ter sido definida como função membro pública da classe Pessoa em vez de uma função plena sem objetos.

```

#include <iostream>
#include "pessoa.h" // dada anteriormente

void printpessoa(Pessoa const &p)
{
    cout << "Nome      : " << p.nome()      << endl <<
         "Endereço  : " << p.endereço()    << endl <<
         "Fone      : " << p.fone()        << endl <<
         "Peso   : " << p.peso()          << endl;
}

int main()
{
    Pessoa p;

    p.setNome("Linus Torvalds");
    p.setEndereço("E-mail: Torvalds@cs.helsinki.fi");
    p.setFone("- desconhecido - ");
    p.setPeso(75); // kg.

    printpessoa(p);
    return 0;
}

```

```

/*
    Saída Produzida:

Nome      : Linus Torvalds
Endereço  : E-mail: Torvalds@cs.helsinki.fi
Fone      : - desconhecido -
Peso      : 75

*/

```

6.1.2: Construtores: com e sem argumentos

Nas declarações acima da classe Pessoa o construtor não possui argumentos. A linguagem C++ permite construtores com ou sem argumentos. Os argumentos são fornecidos quando o objeto é criado.

Para a classe Pessoa um construtor com três argumentos e um inteiro sem sinal pode ser prático: Estes argumentos representariam, respectivamente, o nome da pessoa, o endereço, o número do telefone e o peso. Tal construtor é:

```

Pessoa::Pessoa(string const &nome, string const &endereço,
               string const &fone, size_t peso)
{
    d_nome = nome;
    d_endereço = endereço;
    d_fone = fone;
    d_peso = peso;
}

```

O construtor precisa ser declarado na interface de classe:

```

class Pessoa
{
public:
    Pessoa(string const &nome, string const &endereço,
           string const &fone, size_t peso);

    // o resto da interface de classe
};

```

Agora que este construtor foi declarado, o construtor padrão precisa ser declarado explicitamente também, se ainda quisermos poder construir um objeto Pessoa pleno sem qualquer especificação inicial de valores para seus membros de dados.

Como a linguagem C++ permite a sobrecarga das funções, uma tal declaração pode coexistir

com um construtor sem argumentos. A classe Pessoa teria então dois construtores e a parte relevante da interface de classe ficaria:

```
class Pessoa
{
    public:
        Pessoa();
        Pessoa(string const &nome, string const &endereço,
               string const &fone, size_t peso);

        // o resto da interface da classe
};
```

Neste caso o construtor de Pessoa() não tem muito a fazer, já que não tem que iniciar as strings membros de dados do objeto Pessoa: Como esses membros de dados strings são objetos, já são iniciados vazios por padrão. Porém, há também o membro de dados size_t, sendo um tipo de dado básico não é iniciado (os tipos básicos não são iniciados automaticamente). Portanto, a menos que d_peso seja explicitamente iniciado, será:

- Um valor aleatório para objetos Pessoa locais,
- 0 para objetos Pessoa globais e estáticos

O valor 0 não seria tão mau, mas normalmente não desejamos um valor aleatório em nossos membros de dados não iniciados a um valor razoável automaticamente. Eis a implantação do construtor padrão:

```
Pessoa::Pessoa()
{
    d_peso = 0;
}
```

O uso de um construtor com e sem argumentos (i.e., o padrão) é ilustrado no seguinte fragmento de código. O objeto é iniciado em sua definição usando o construtor com argumentos, com o objeto b o construtor padrão é usado:

```
int main()
{
    Pessoa a("Karel", "Rietveldlaan 37", "542 6044", 70);
    Pessoa b;

    return 0;
}
```

Neste exemplo, os objetos Pessoa a e b são criados quando main() parte: são objetos locais, que vivem enquanto a função main() está ativa.

Se os objetos Pessoa precisam ser construídos usando-se outros argumentos, outro construtor é requerido. É possível definir valores padrão para os parâmetros. Esses valores devem ser dados na interface de classe, p.ex.:

```
class Pessoa
{
    public:
        Pessoa();
        Pessoa(string const &nome,
                string const &endereço = "--desconhecido--",
                string const &fone     = "--desconhecido--",
                size_t peso = 0);

        // o resto da interface de classe
};
```

Com frequência os construtores são implantados muito semelhantes. Devido ao fato de que com frequência os parâmetros do construtor são definidos por conveniência: Um construtor requerendo o peso e não o número telefônico não pode ser definido com os argumentos padrão, já que somente o último parâmetro é requerido e não os quatro. Isto não pode ser resolvido usando os parâmetros padrão, somente definindo outro construtor, sem especificar o telefone.

Em algumas linguagens (p.ex., Java) permitem aos construtores chamar construtores, isto é conceitualmente estranho. Estranho porque acaba com o conceito de construtor. Um construtor é um meio para construir um objeto, não a si mesmo, já que ainda não construiu nada

Na linguagem C++ a maneira de proceder é a seguinte: Todos os construtores devem iniciar seus membros de dados de referência, ou o compilador (com direito) reclamará. Esta é uma das razões fundamentais que não se pode chamar um construtor durante uma construção. A seguir temos duas opções:

- Se o corpo do processo de construção é extenso, mas (parametrizável) igual ao corpo de outro construtor, fatorise! Faça um membro privado init (talvez parametrizado) chamado pelos construtores. Cada construtor iniciará qualquer referência a membros de dados que sua classe possa ter.
- Se os construtores atuam radicalmente diferentes, então não resta nada mais que fazer construtores diferentes.

6.1.2.1: A ordem da construção

A probabilidade de passar argumentos aos construtores nos permite monitorar a construção dos objetos durante a execução do programa. Este fato pode-se apreciar na seguinte listagem, usando uma classe Teste. O programa abaixo mostra a classe Teste, um objeto Teste global e dois locais: numa função func() e na função main(). A ordem da construção é como esperado: primeiro o objeto global, o primeiro objeto local de main(), o objeto local da função func() e finalmente o segundo objeto de main():

```
#include <iostream>
#include <string>
using namespace std;

class Teste
{
public:
    Teste(string const &nome);    // construtor com um argumento
};

Teste::Teste(string const &nome)
{
    cout << "Objeto de Teste " << nome << " criado" << endl;
}

Teste globaltest("global");

void func()
{
    Teste functest("func");
}

int main()
{
    Teste first("primeiro de main");
    func();
    Teste second("segundo de main");
    return 0;
}

/*
Saída Gerada:
Teste objeto global criado
Teste objeto primeiro de main criado
Teste objeto func criado
Teste objeto segundo de main criado
*/
```

6.2: Funções membro constantes e objetos constantes

A palavra chave 'const' é freqüentemente usada trás a lista de parâmetros das funções membro. A palavra chave indica que a função membro não altera os membros de dados de seus objetos, mas somente os inspeciona. Estas funções membro são chamadas funções membro constantes. Usando o exemplo da classe Pessoa, vemos que as funções acessoras foram declaradas constantes:

```
class Pessoa
{
    public:
        std::string const &nome() const;
        std::string const &endereço() const;
        std::string const &fone() const;
};
```

Este fragmento ilustra o fato de que a palavra chave const aparece trás a lista de argumentos das funções. Note que nesta situação a regra prática dada na seção 3.1.3 está bem aplicada: aquilo que aparece antes da palavra chave const não pode ser alterado e não altera (seus próprios) dados.

A especificação const tem que ser repetida na declaração das funções membro:

```
string const &Pessoa::nome() const
{
    return d_nome;
}
```

Uma função membro que é declarada e definida como constante não pode alterar qualquer campo de dado da classe. Em outras palavras, um comando como:

```
d_nome = 0;
```

na função constante acima nome() resultaria num erro de compilação.

As funções membro constantes existem porque a linguagem C++ permite serem criados objetos constantes ou (usado mais a miúdo) referência a objetos constantes serem passadas a funções. Somente funções membro que não modificam tais objetos, i.e., as funções membro constantes, podem ser chamadas. As únicas exceções a esta regra são os construtores e os destrutores: estes são chamados 'automaticamente'. A possibilidade de chamar construtores ou destrutores é comparável à definição de uma variável `int const max = 10`. Em situações como essas não uma adjudicação, mas uma iniciação tem lugar na criação. Analogamente o construtor pode iniciar seus objetos quando uma variável constante é criada os campos de dados são iniciados pelo construtor.

O seguinte exemplo mostra a definição de um objeto constante da classe Pessoa. Quando o objeto é criado o campo de dados é iniciado pelo construtor:


```
Pessoa const eu("Karel", "karel@icce.rug.nl", "542 6044");
```

Seguindo esta definição seria ilegal tentar redefinir o nome, endereço ou número telefônico do objeto eu: num comando como:

```
eu.setNome("Lerak");
```

Não seria aceito pelo compilador. Uma vez mais, olhe a posição da palavra chave const na definição da variável: const seguindo Pessoa e precedendo eu associada à esquerda: o objeto Pessoa em geral tem que permanecer inalterado. Daí que se múltiplos objetos foram definidos aqui, todos são objetos constantes de Pessoa, como em:

```
Pessoa const          // todos os objetos constantes de Pessoa
    kk("Karel", "karel@icce.rug.nl", "542 6044"),
    fbb("Frank", "f.b.brokken@rc.rug.nl", "363 9281");
```

As funções membro que não modificam seu objeto devem ser definidas funções membro constantes. Isto permite o subsequente uso dessas funções com objetos constantes ou referências constante. Como regra prática deve-se sempre dar atributo constante às funções membro, a menos que modifiquem o objeto de dados.

Anteriormente, na seção 2.5.11, o conceito de sobrecarga de funções foi introduzido. Lá diz que as funções membro podem ser sobrecarregadas somente em relação a seus atributos constantes. Nesses casos o compilador usará a função membro que tenha objetos com qualificação constante mais semelhantes:

- Quando o objeto é constante, somente funções membro constantes podem ser usadas.
- Quando o objeto não é constante, se usará funções membro não constante. a menos de que uma função membro constante esteja disponível. Nesse caso a função membro constante será usada.

A seguir é dado um exemplo de seleção de funções membro (não) constante:

```
#include <iostream>
using namespace std;

class X
{
    public:
        X();
        void member();
        void member() const;
};

X::X()
```

```

{}
void X::member()
{
    cout << "membro não constante\n";
}
void X::member() const
{
    cout << "membro constante\n";
}

int main()
{
    X const constObject;
    X      nonConstObject;

    constObject.member();
    nonConstObject.member();
}
/*
    Saída Gerada:

    membro constante
    membro não constante
*/

```

A sobrecarga das funções membro em seus atributos constantes ocorre comumente no contexto do operador de sobrecarga. Veja o Capítulo 9, em particular a seção 9.1 para detalhes.

6.2.1: Objetos anônimos

Existem situações onde são usados objetos porque oferecem uma certa funcionalidade. Só existem pela funcionalidade que oferecem e nada nos próprios objetos nunca muda. Estas situações lembram a situação bem conhecida na linguagem de programação **C** onde um ponteiro de uma função é passado a outra função, para permitir a configuração, em tempo de execução do comportamento da última função.

Por exemplo, a classe `Print` pode oferecer uma facilidade para imprimir uma 'string', prefixando-a com um prefixo configurável e afixando um afixo configurável a ela. Tal classe *poderia* ser dada pelo seguinte protótipo:

```

class Print
{
public:
    printout(std::string const &prefix, std::string const &text,
            std::string const &affix) const;
};

```

Uma interface como esta permitiria-nos fazer coisas como:

```
Print print;
for (unsigned idx = 0; idx < argc; idx++)
    print.printout("arg: ", argv[idx], "\n");
```

Isto funcionaria bem, mas pode ser bastante melhorado se pudéssemos passar os argumentos invariáveis de `printout` aos construtores de `Print`: Desta forma poderíamos não só simplificar o protótipo de `printout` (só um argumento seria necessário ser passado no lugar de três, permitindo-nos fazer as chamadas mais rápidas a `printout`) mas também poderíamos capturar o código acima numa função que espere um objeto `Print`:

```
void printText(Print const &print, int argc, char *argv[])
{
    for (unsigned idx = 0; idx < argc; idx++)
        print.printout(argv[idx]);
}
```

Agora temos um pedaço de código bastante genérico, por fim, no que concerne a `Print`. Se pudéssemos fornecer à interface de `Print` os seguintes construtores seríamos capazes de configurar nossa 'stream' de saída também:

```
Print(char const *prefix, char const *affix);
Print(ostream &out, char const *prefix, char const *affix);
```

Agora `printText` pode ser usado como segue:

```
Print p1("arg: ", "\n");           // prints to cout
Print p2(cerr, "err: --", "--\n");  // prints to cerr

printText(p1, argc, argv);          // prints to cout
printText(p2, argc, argv);          // prints to cerr
```

Porém, quando olhamos de perto este exemplo, poderia ser limpo para que ambos `p1` e `p2` sejam usados somente dentro da função `printText`. Ainda mais, como podemos ver do protótipo de `printText`, ela não modifica os dados internos do objeto `Print` que usa.

Em situações como estas não é necessário definir objetos antes de serem usados. Em vez disso poderiam ser usados *objetos anônimos*. É indicado usar objetos anônimos quando:

- O parâmetro de uma função define uma referência `const` a um objeto;
- O objeto *só* é preciso dentro da chamada à função.

Os `objects` anônimos são definidos chamando-se um construtor sem um nome para o objeto construído. No exemplo acima os objetos anônimos podem ser usados como segue:

```
printText(Print("arg: ", "\n"), argc, argv);           // prints to cout
```

```
printText(Print(cerr, "err: --", "--\n"), argc, argv); // prints to cerr
```

Nesta situação os objetos `Print` são construídos e imediatamente passados como os primeiros argumentos das funções `printText`, onde são acessíveis como parâmetro da função `print`. Enquanto a função `printText` é executada podem ser usados, mas uma vez que a função terminou, os objetos `Print` já não são acessíveis.

Os objetos anônimos cessam de existir quando a função para a qual foram criados terminou. Respeito a isto diferem das variáveis locais ordinárias cujo tempo de vida acaba junto com o da função onde foram definidos.

6.2.1.1: Subtítulos com objetos anônimos

Como foi discutido, os objetos anônimos podem ser usados para iniciar parâmetros de funções que são referências a objetos `const`. Estes objetos são criados justo antes de que uma função seja chamada, e destruídos uma vez que a função termine. Este uso de objetos anônimos para iniciar parâmetros de funções são freqüentemente vistos, mas a gramática `C++` nos permite usar objetos anônimos em outras situações também. Considere o seguinte pedaço de código:

```
int main()
{
    // initial statements
    Print("hello", "world");
    // later statements
}
```

Neste exemplo o objeto anônimos `Print` é construído e imediatamente destruído depois de sua construção. Assim, segundo os 'comandos iniciais' nosso objeto `Print` é construído, então é destruído, seguindo-se a execução dos 'últimos comandos'. Isto é notável já que mostra que o padrão das regras de tempo de vida não se aplica aos objetos anônimos. Seus tempos de vida estão limitados pelo *comando*, antes que pelo *fim do bloco* onde estão definidos.

Claro que nos admiramos porque um objeto anônimo pleno pode ser considerado útil. Pensamos pelo menos numa situação. Assuma que queremos por *marcadores* em nosso código que produza alguma saída quando a execução do programa chegue em certo ponto. Um construtor do objeto pode ser implantado de tal maneira que faça uma marca, permitindo-nos assim marcar o código pela definição de objetos anônimos, antes que com objetos nomeados.

Contudo, a gramática da `C++` contém outra característica notável. Considere o seguinte exemplo:

```
int main(int argc, char *argv[])
{
    Print p(" ", " "); // 1
```

```

    printText(Print(p), argc, argv);    // 2
}

```

Neste exemplo um objeto `p` não-anônimo é construído no comando 1, cujo objeto é então usado no comando 2 para *iniciar* um objeto anônimo que, em seu turno, é usado para iniciar a referência ao parâmetro `printText` `const`. Este uso de um objeto existente iniciar outro objeto é prática comum, e é baseada na existência de um, assim chamado, *construtor de cópia*. Um construtor de cópia cria um objeto (comoit é um construtor), usando as características de um objeto existente para iniciar os dados do novo objeto. Os construtores de cópia são discutidos em profundidade no capítulo [7](#), mas presentemente somente o conceito de um construtor de cópia é usado.

No último exemplo um construtor de cópia foi usado para iniciar um objeto anônimo, que foi então usado para iniciar um parâmetro de uma função. Contudo, quando tentamos aplicar o mesmo truque (i.e., usar um objeto existente para iniciar um objeto anônimo) num comando pleno, o compilador gera um erro: o objeto `p` não pode ser redefinido (no comando 3, abaixo):

```

int main(int argc, char *argv[])
{
    Print p("", "");                // 1
    printText(Print(p), argc, argv); // 2
    Print(p);                        // 3 error!
}

```

Então, usar um objeto existente para iniciar um objeto anônimo que é usado como argumento de uma função está bem, mas um objeto existente não pode ser usado para iniciar um objeto anônimo num comando pleno?

A resposta a esta aparente contradição está na mensagem de erro do compilador. No comando 3 o compilador afirma algo como:

```

error: redeclaration of 'Print p'

```

que resolve o problema, comprovando que num comando composto os objetos e variáveis também podem ser definidos. Dentro de um comando composto, um *nome de tipo* seguido pelo nome de uma variável é a forma gramatical de definição de uma variável. Pode-se usar *parênteses* para quebrar as prioridades, mas se não há prioridades que quebrar, não têm efeito e são simplesmente ignorados pelo compilador. No comando 3 os parênteses permite livrarmo-nos do espaço em branco requerido entre um nome de tipo e o nome da variável, mas para o compilador escrevemos

```

Print (p);

```

que é, já que os parênteses são supérfluos, igual a

```

Print p;

```

produzindo, assim, a re-declaração de `p`.

Como um exemplo tardio: quando definimos uma variável usando um tipo básico (e.g., `double`) usando parênteses supérfluos o compilador removerá caladamente esses parênteses para nós:

```
double (((a)));          // feio, mas ok.
```

Para resumir nossas descobertas acerca de variáveis anônimas:

- Objetos anônimos são ótimos para iniciar referências a parâmetros `const`.
- A mesma sintaxe, contudo, também pode ser usada em comandos isolados, onde são interpretados como definições de variáveis se nossa intenção for iniciar um objeto anônimo usando um objeto existente.
- Como isto pode causar confusão, talvez seja melhor restringir o uso de objetos anônimos à primeira (e principal) forma: iniciando os parâmetros da função.

6.3: A palavra chave `inline`

Vamos dar uma outra olhada na implantação da função `Pessoa::nome()`:

```
string const &Pessoa::nome() const
{
    return d_nome;
}
```

Esta função é usada para retirar o campo `nome` de um objeto da classe `Pessoa`. Num fragmento de código como:

```
Pessoa
    frank("Frank", "Oostumerweg 17", "403 2223");

cout << frank.name();
```

as seguintes ações têm lugar:

- A função `Pessoa::nome()` é chamada.
- Esta função retorna o nome do objeto `frank` como referência.
- O nome referenciado é posto em `cout`.

Especialmente a primeira parte destas ações resulta na perda de algum tempo, já que uma chamada extra à função é necessária para retirar o valor do campo `nome`. Às vezes é desejável um

procedimento mais rápido, nesses casos o campo nome fica disponível imediatamente, sem mesmo chamar uma função nome(). Isto pode ser feito usando-se funções 'inline'.

6.3.1: Funções 'inline' dentro das interfaces de classe

As funções inline podem ser implantadas dentro da interface de classe. Para a classe Pessoa resulta na seguinte implantação de nome():

```
class Pessoa
{
    public:
        string const &nome() const
        {
            return d_nome;
        }
};
```

Note que o código inline da função nome() agora ocorre literalmente 'inline' na interface da classe Pessoa. A palavra chave const ocorre depois da declaração da função e antes do bloco de código.

Portanto, as funções inline que aparecem na interface de classe são completamente definidas com a interface de classe.

Isto tem o seguinte efeito: Sempre que nome() seja chamada num comando do programa, o compilador gera o código do corpo da função no lugar da chamada. A própria função nunca será chamada. Conseqüentemente, é evitada a chamada mas o corpo da função aparecerá tantas vezes no programa final quantas forem as chamadas à função inline.

Esta construção onde o próprio código da função é inserido antes que uma chamada a ela é chamada de função inline. Isto estará provavelmente bem se a função é pequena e necessita ser executada rápido. Já não será tão desejável se o código for extenso. O compilador também sabe disto e considera o uso de funções inline melhor como um comando: Se o compilador considerar a função muito longa não garantirá a requisição, mas em vez disso, tratará a função como normal.

6.3.2: Funções 'inline' fora das interfaces de classe

A segunda maneira de implantar uma função inline deixa a interface de classe intacta, mas a menciona a palavra chave inline na definição da função. A interface e a implantação neste caso será como

segue:

```
class Pessoa
{
    public:
        string const &nome() const;
};

inline string const &Pessoa::nome() const
{
    return d_nome;
}
```

Mais uma vez o compilador vai inserir o código da função nome() no lugar de gerar uma chamada.

A função inline deve estar no mesmo arquivo que a interface de classe e não pode ser compilada para ser guardada, p.ex., numa biblioteca. A razão para isto é que o compilador e mais ainda o lincador deve ser capaz de inserir o código da função num texto fonte em compilação. O código mantido numa biblioteca está inacessível ao compilador. Conseqüentemente, funções inline são sempre definidas junto à interface de classe.

6.3.3: Quando usar funções `inline`

Quando se deve ou não usar funções inline? Existem algumas regras práticas que se deve seguir:

- Em geral não se deve usar funções inline. Voilà, simples, não?
- A definição de funções inline deve ser considerada uma vez que um programa pronto e testado se revele muito lento e mostre engarrafamentos respeito a algumas funções. Será necessário um analisador que execute o programa e determine onde é gasta a maior parte do tempo para então realizar tais otimizações.
- Pode-se usar funções inline quando as funções membro forem com um comando muito simples (tal como o comando de retorno na função Pessoa::nome()).
- Ao definirmos uma função como inline sua implantação será inserida no código sempre que a função for usada. Como consequência quando a implantação da função inline muda, todo o código que use a função inline terá que ser recompilado. Na prática significa que todas as funções serão recompiladas incluindo (direta ou indiretamente) o arquivo cabeçalho da classe onde a

função inline está definida.

- Somente é útil implantar uma função inline quando o tempo de chamada da função é longo comparado com o do código da função. Um exemplo que dificilmente terá algum efeito sobre a velocidade do programa é:

```
void Pessoa::printnome() const
{
    cout << d_nome << endl;
}
```

Esta função que é, para efeito do exemplo, apresentada como membro da classe Pessoa, contém só um comando. Contudo, o comando toma um tempo relativamente longo na execução. Em geral funções que realizam operações de entrada e saída tomam muito tempo. O efeito da conversão desta função `printnome()` a inline leva a um ganho insignificante em tempo de execução.

Todas as funções inline têm uma desvantagem: O código é inserido durante a compilação e necessita ser conhecido aí. Assim, como dito anteriormente, uma função inline não pode nunca estar numa biblioteca de tempo de execução. Na prática significa que uma função inline se localiza perto da interface de classe, usualmente no mesmo arquivo cabeçalho. O resultado é um cabeçalho não só com a declaração da classe mas também com parte de sua implantação, isto borra a distinção entre interface e implantação.

Finalmente, note uma vez mais, a palavra chave `inline` não é realmente um comando ao compilador. Antes é uma requisição ao compilador que pode ou não aceitar.

6.4: Objetos dentro de objetos: composição

Com frequência os objetos são usados como membros de dados nas definições de classe. Isto é chamado composição.

Por exemplo, a classe Pessoa guarda informações sobre o nome, endereço e número telefônico. Estas informações são armazenadas em strings de dados membro, que são eles próprios: composição.

A composição não é algo extraordinário ou específico da linguagem C++: Na linguagem C uma estrutura ou campo união é comumente usada em outras composições de tipos.

A iniciação de objetos compostos merece alguma atenção especial: Os tópicos das seções

vindouras.

6.4.1: Composição e objetos constantes: iniciadores de membros compostos

A composição de objetos tem importantes conseqüências para o construtor de funções do objeto 'composto' (embebido). A menos de instruções explícitas do contrário, o compilador gera o código de chamada do construtor padrão de todas as classes compostas no construtor da classe que está compondo.

Com freqüência é desejável iniciar um objeto composto com um construtor específico da classe em composição. Isto está ilustrado abaixo para a classe Pessoa. No fragmento assume-se que um construtor para Pessoa pode ser definido com quatro argumentos: o nome, o endereço, o número telefônico e o peso:

```
Pessoa::Pessoa(char const *nome, char const *endereço,
               char const *fone, size_t peso)
:
    d_nome(nome),
    d_endereço(endereço),
    d_fone(fone),
    d_peso(peso)
{ }
```

Seguindo a lista de argumentos do construtor Pessoa::Pessoa(), os construtores das strings de dados membro são chamados especificamente, p.ex., nome(mn). A iniciação tem lugar antes do bloco de código de Pessoa::Pessoa() (agora vazio) seja executado. Esta construção, onde a iniciação dos membros tem lugar antes de que o bloco de código seja executado é chamada iniciação de membro. A iniciação de membro pode ser explicitada na lista de iniciação dos membros, esta pode aparecer depois da lista de parâmetros, entre dois pontos (anunciando o começo da lista de iniciação dos membros) e a chave aberta do bloco de código do construtor.

A iniciação dos membros sempre acontece quando os objetos são compostos na classe: Se não são mencionados construtores na lista de iniciação dos membros os construtores padrão dos objetos são chamados. Note que isto só é verdade para objetos. Os membros de dados de tipos primitivos não são automaticamente iniciados.

A iniciação dos membros, contudo, pode ser usada também para membros de dados com tipos primitivos, como inteiros e duplos. O exemplo acima mostra a iniciação dos membros de dados do parâmetro peso. Para uma ilustração maior, usamos os mesmos identificadores aqui: com a iniciação dos

membros não há ambigüidade e o primeiro (esquerda) identificador em peso (peso) é sempre interpretado como um membro de dados a ser iniciado, pois, o identificador entre parênteses é interpretado como o parâmetro.

Quando uma classe tem múltiplos membros de dados compostos, todos os membros podem ser iniciados usando-se uma 'lista de membros iniciadora': Esta lista consiste nos construtores de todos os objetos compostos, separados por vírgulas. A ordem na qual os objetos são iniciados é aquela em que os objetos são definidos na interface de classe. Se a ordem de iniciação no construtor difere daquela na interface da classe, então o compilador reclama e reordena a iniciação para estar de acordo com a ordem da interface de classe.

Os iniciadores de membros devem ser usados tão a miúdo quanto possível: devem ser usados sem cerimônias e o não uso deles pode resultar num código ineficiente: com os objetos sempre em última instância o construtor padrão é chamado. Assim, no exemplo seguinte primeiro as strings membro são iniciadas vazias, para depois esses valores serem redefinidos com seus valores iniciais pretendidos. Claro que a iniciação imediata com os valores pretendidos seria mais eficiente.

```
Pessoa::Pessoa(char const *nome, char const *endereço,
               char const *fone, size_t peso)
{
    d_nome = nome;
    d_endereço = endereço;
    d_fone = fone;
    d_peso = peso;
}
```

Este método é não só ineficiente, mas ainda mais: não funciona quando o objeto é declarado constante. Um campo de dados como aniversário é um bom candidato a ser declarado constante, já que o dia de nascimento das pessoas não muda muito.

Isto significa que quando a definição de uma Pessoa é alterada, como contém uma string membro constante aniversário, a implantação do construtor Pessoa::Pessoa() onde também está o aniversário e tem que ser iniciada, um iniciador de membro precisa ser usado para aniversário. Uma adjudicação direta a aniversário seria ilegal, já que aniversário é constante. O seguinte exemplo ilustra a iniciação de um membro de dados constante:

```
Pessoa::Pessoa(char const *nome, char const *endereço,
               char const *fone, char const *aniversário,
               size_t peso)
:
    d_nome(nome),
    d_endereço(endereço),
    d_fone(fone),
```

```
d_aniversário(aniversário),    // assume: string const d_aniversário
d_peso(peso)
{}
```

Conclusão, a regra prática é a seguinte: Quando uma composição de objetos é usada, o método para o iniciador de membros é preferível que seja de iniciação explícita dos objetos compostos. Isto não só resulta em mais eficiência do código, mas também permite aos objetos compostos serem declarados constantes.

6.4.2: Objetos compostos e de referência: iniciadores de membros de referência

Além de usar iniciadores de objetos membros compostos (sejam constantes ou não), existe outra situação em que iniciadores de membros precisam ser usados. Considere a seguinte situação:

Um programa usa um objeto da classe Configfile, definido em main() para acessar informações num arquivo de configuração. O arquivo de configuração contém parâmetros do programa que devem ter seus valores copiados do arquivo de configuração, melhor dito, postos como argumentos para a linha de comando.

Assume-se que outro objeto usado na função main() é da classe Processo, fazendo 'todo o trabalho'. Que possibilidades temos de dizer ao objeto da classe Processo que um objeto da classe Configfile existe?

- Os objetos podem ter sido declarados como globais. Isto é uma possibilidade, mas não muito boa, já que todas as vantagens dos objetos locais se perdem.
- O objeto Configfile pode ser passado ao objeto de Processo na hora da construção. Passagem de um objeto pelo valor (bluntly) não parece ser uma boa idéia, já que o objeto deve ser copiado no parâmetro da classe Configfile e então um membro de dados da classe Processo pode ser usado para fazer o objeto de Configfile acessível à classe Processo. Isto envolve ainda outra tarefa de cópia de um objeto, como na seguinte situação:

```
Processo::Processo(Configfile conf)    // cópia de quem chamou
{
    d_conf = conf;                      // copia ao membro conf
}
```

o As instruções de cópia podem ser evitada se usarmos ponteiros aos objetos de Configfile, como em:

```
Processo::Processo(Configfile *conf)  // ponteiro ao objeto externo
```

```

{
    d_conf = conf;                // d_conf é um Configfile *
}

```

Esta construção está boa, mas nos força a usar o operador de seleção de campo ' \rightarrow ', antes que o operador '.', que é (em disputa) inconveniente: conceitualmente se tende em pensar o objeto de Configfile como um objeto e não como um apontador a um objeto. Na linguagem C, provavelmente, seria o método preferido, mas em C++ podemos fazer melhor.

- Melhor que usar um valor ou apontador a um parâmetro, o parâmetro de Configfile poderia ser definido como um parâmetro de referência ao construtor de Processo. Em seguida definimos como um membro de dados um parâmetro de referência a Configfile na classe Processo. Usando a variável de referência efetivamente se usa um ponteiro como uma variável.

Contudo, a seguinte construção não resultará na iniciação do membro de dados de Configfile &d_conf:

```

Processo::Processo(Configfile &conf)
{
    d_conf = conf;            // errado: não adjudica
}

```

A instrução `d_conf = conf` falha porque o compilador não a vê como uma inicialização, mas considera uma adjudicação a um objeto Configfile (i.e., `conf`) a outro (`d_conf`). Assim o faz porque é a interpretação normal: Uma adjudicação a uma varável de referência é uma adjudicação à variável à qual a variável de referência referencia. Mas, a qual variável deve `d_conf` referenciar? A nenhuma variável, já que não iniciamos `d_conf`. Depois de tudo, o único propósito do comando `d_conf = conf` era iniciar `d_conf`...

Assim, como proceder para iniciar `d_conf`? Nesta situação, uma vez mais, usamos a sintaxe de iniciação de membro. O seguinte exemplo mostra o modo correto de iniciar `d_conf`:

```

Processo::Processo(Configfile &conf)
:
    d_conf(conf)            // iniciando o membro de referência
{}

```

Note que esta sintaxe pode ser usada em todos os casos onde são usados membros de dados de referência, uma construção como:

```

Processo::Processo(int &ir)
:
    d_ir(ir)
{}

```

Poderia ser chamada para tanto.

6.5: Organização do Arquivo Cabeçalho

Na seção 2.5.9 foram discutidos os requerimentos de um arquivo cabeçalho quando um programa C++ usa também funções C.

Quando se usa classes existem mais requerimentos para a organização dos arquivos cabeçalhos. Aqui os discutiremos.

Primeiro os arquivos fonte. Com exceção de funções ocasionais sem classe, os arquivos fonte devem conter o código das funções membro das classes. Para os arquivos fonte há basicamente dois métodos:

- Todos os arquivos cabeçalhos para uma função membro são incluídos em cada arquivo fonte individual.
- Todos os arquivos cabeçalho requeridos para todas as funções membro são incluídos no arquivo cabeçalho da classe e cada arquivo fonte dessa classe inclui somente o arquivo cabeçalho da classe.

A primeira alternativa tem a vantagem de economia para o compilador: Somente necessita ler os cabeçalhos necessários ao arquivo fonte em particular. Tem a desvantagem de que o programador precisa incluir múltiplos arquivos cabeçalho uma e outra vez e pensar sobre quais arquivos devem ser incluídos onde.

A segunda alternativa tem a vantagem de economizar trabalho ao programador: O arquivo cabeçalho da classe acumula todos os outros, assim tende a ser mais e mais útil. Tem a desvantagem que o compilador tem que ler arquivos que muitas vezes não são usados no arquivo fonte em compilação.

Como os computadores são cada vez mais rápidos, a segunda alternativa é preferível. Assim, como ponto de partida sugerimos que os arquivos fonte de uma classe particular MinhaClasse sejam organizados como no seguinte exemplo:

```
#include <minhaclasse.h>

int MinhaClasse::aMemberFunction()
{ }
```

Existe só uma diretiva de inclusão. Note que a diretiva se refere ao arquivo cabeçalho num

diretório mencionado na variável de ambiente INCLUDE-file. Os arquivos cabeçalho locais (que usam #include “minhaclasse.h) também podem ser usados, mas isto tenderia a complicar um pouco a organização do arquivo cabeçalho da classe.

Se houver colisões com os nomes dos arquivos cabeçalho é posto num subdiretório, se houver, de um dos diretórios da variável de ambiente INCLUDE (p.ex., /usr/local/include/meucabaçalho/).

Se uma classe MinhaClasse for desenvolvida lá, cria um subdiretório (ou um link a um subdiretório) meuscabeçlhos de um dos diretórios padrão para conter todos os arquivos cabeçalho de todas as classes desenvolvidas como parte do projeto. As diretivas de inclusão serão, então, similares a #include <meucabeçalho/minhaclasse.h> e as colisões de nome com outros arquivos cabeçalho são evitadas.

A própria organização do arquivo cabeçalho requer alguma atenção. Considere o seguinte exemplo, no qual duas classes Arquivo e String são usadas.

Assume-se que a classe Arquivo tem um membro gets(String &destino), enquanto a classe String tem uma função membro getLine(Arquivo &Arquivo). O cabeçalho (parcial) da classe String é:

```
#ifndef _String_h_
#define _String_h_

#include <projeto/file.h>    // para saber sobre um Arquivo

class String
{
    public:
        void getLine(File &file);
};
#endif
```

Contudo, uma iniciação semelhante é necessária para a classe Arquivo:

```
#ifndef _File_h_
#define _File_h_

#include <projeto/string.h>    // para saber sobre uma String

class File
{
    public:
        void gets(String &string);
};
```

```
};  
#endif
```

Que problema nos criamos. O compilador intenta compilar o arquivo fonte da função `Arquivo::gets()` procedendo assim:

- O cabeçalho `projeto/arquivo.h` é aberto para leitura;
- `_Arquivo_h_` é definido;
- O cabeçalho `projeto/string.h` é aberto para leitura;
- `_String_h_` é definido;
- O cabeçalho `projeto/arquivo.h` é (outra vez) aberto para leitura;
- Aparentemente `_Arquivo_h_` já está definido, portanto, o resto de `projeto/arquivo.h` é saltado;
- A interface da classe `String` é passada pelo parser;
- Na interface de classe uma referência a um objeto de `Arquivo` é encontrada;
- Como a classe `Arquivo` ainda não passou pelo parser `Arquivo` é um tipo não definido e o compilador sai com erro.

A solução para este problema é usar uma referência à classe antes da interface de classe e incluir o arquivo cabeçalho correspondente depois da interface de classe. Assim:

```
#ifndef _String_h_  
#define _String_h_  
  
class Arquivo;                // referência avançada  
  
class String  
{  
public:  
    void getLine(Arquivo &arquivo);  
};  
  
#include <projeto/arquivo.h>  // para saber sobre um Arquivo  
#endif
```

Uma iniciação semelhante é necessária para a classe `Arquivo`:


```

#ifndef _Arquivo_h_
#define _Arquivo_h_

class String;                // referência avançada

class Arquivo
{
    public:
        void gets(String &string);
};

#include <project/string.h>    // para saber sobre uma String

#endif

```

Isto funciona bem em qualquer situação onde as referências ou ponteiros a outra classe estão envolvidos e com funções membro (não inline) que retornam tipos da classe como valores ou parâmetros.

Note que esta iniciação não funciona com composições, nem com funções membro inline. Assumamos que a classe Arquivo tenha um membro de dados composto da classe String. Nesse caso a interface de classe da classe Arquivo precisa incluir o arquivo cabeçalho da classe String antes da interface de classe, porque, de outra forma, o compilador não pode saber o tamanho do objeto Arquivo, já que não conhece o tamanho do objeto String sendo que a interface da classe Arquivo já está completa.

No caso onde as classes possuem objetos compostos (ou derivados de outras classes veja o capítulo 13) o arquivo cabeçalho das classes com objetos compostos deve ser lida antes que a própria interface de classe. Nesse caso a classe Arquivo deve ser definida como segue:

```

#ifndef _Arquivo_h_
#define _Arquivo_h_

#include <projeto/string.h>    // para saber sobre uma String

class Arquivo
{
    String d_line;            // composição !

    public:
        void gets(String &string);
};

#endif

```

Note que a classe String não pode ter um objeto Arquivo como um membro composto: tal situação resultaria numa indefinição de classe durante a compilação as fontes dessas classes.

Todos os arquivos cabeçalho (que aparecem abaixo da interface de classe) são requeridos porque são usados pelos arquivos fonte da classe.

O método nos permite introduzir ainda outro refinamento:

- Os arquivos cabeçalho que definem uma interface de classe deve declarar o que deve ser declarado antes da definição da interface de classe. Assim, as classes mencionadas na interface de classe devem ser especificadas usando declarações antes, a menos que
- É a classe base da classe atual (veja Capítulo 13);
- É um tipo de classe com membros de dados compostos;
- É usada com funções membro inline.

Em particular: arquivos cabeçalho adicionais não são requeridos para:

- Tipos de funções de classe que retornam valores;
- Tipos de funções de classe que retornam valores de parâmetros.

Arquivos cabeçalho de classes cujos objetos ou são compostos ou herdados ou usados em funções inline, precisam ser conhecidos pelo compilador antes que a interface da classe inicie. A informação do arquivo cabeçalho é protegida pela construção `#ifndef...#endif`, introduzida na seção 2.5.9.

- Os programas cujas fontes usam a classe necessitam só incluir o arquivo cabeçalho. Lakos (2001) refina este processo ainda mais. Veja seu livro *Large-Scale C++ Software Design* para maiores detalhes. Estes arquivos cabeçalho devem estar disponíveis em lugares bem conhecidos, tal como um diretório ou subdiretório do caminho de inclusão.
- A implantação das funções membro pode ser no arquivo cabeçalho da classe e usualmente em outros arquivos cabeçalho (como `#include <string>`) também. O próprio arquivo cabeçalho da classe bem como esses arquivos cabeçalho adicionais devem ser incluídos num arquivo cabeçalho adicional interno (com extensão `.in` ('internal header')) é sugerida).

O arquivo `.in` deve ser definido no mesmo diretório que os arquivos fonte da classe e tem as seguintes características:

- Não necessitam da proteção `#ifndef...#endif`, já que o arquivo cabeçalho nunca é incluído por outro arquivo cabeçalho.

- O arquivo cabeçalho padrão .h que define a interface de classe está aí incluído.
- Os arquivos cabeçalho de todas as classes usadas com referência avançada no arquivo cabeçalho padrão .h são incluídos.
- Finalmente, todos os outros arquivos cabeçalho requeridos nos arquivos fonte da classe são incluídos.

Como exemplo da organização de um arquivo cabeçalho é:

- Primeira parte, p.ex., /usr/local/include/arquivo.h:

```
#ifndef _Arquivo_h_
#define _Arquivo_h_

#include <fstream>          // para compor com 'ifstream'

class Buffer;              // referência avançada

class Arquivo              // interface de classe
{
    ifstream d_instream;

    public:
        void gets(Buffer &buffer);
};
#endif
```

- Segunda parte, p.ex., ~/meuprojeto/arquivo/arquivo.ih, onde todas as fontes da classe Arquivo são guardadas:

```
#include <meuscabecalhos/arquivo.h> // faz a classe Arquivo conhecida

#include <buffer.h>                  // faz Buffer conhecida por Arquivo
#include <string>                    // usado por membros da classe
#include <sys/stat.h>                // Arquivo.
```

6.5.1: Usando espaços nomeados nos arquivos cabeçalho

Quando entidades de espaços nomeados (namespaces) são usadas em arquivos cabeçalho, em geral, não se deve usar diretivas 'using' nesses arquivos cabeçalho se são de uso geral como arquivos cabeçalho com declarações de classes ou outras entidades de uma biblioteca. Quando a diretiva 'using' é posta num arquivo cabeçalho então os usuários desse arquivo são forçados a aceitar e usar as declarações em todo o código que inclua o arquivo cabeçalho particular.

Por exemplo, se num espaço nomeado especial um objeto `Insertter cout` é declarado, então `especial::cout` é, por certo, um objeto diferente de `std::cout`. Porém, se a classe `Fluxo` é construída, e seu construtor espera uma referência a `especial::Insertter`, então a classe deve ser construída como segue:

```
class especial::Insertter;

class Fluxo
{
public:
    Fluxo(especial::Insertter &ins);
};
```

Mas se a pessoa que projetou a classe `Fluxo` está de humor preguiçoso e se aborrece em escrever continuamente o prefixo `especial::` antes de cada entidade desse espaço nomeado, então deve usar a construção:

```
using namespace especial;

class Insertter;

class Fluxo
{
public:
    Fluxo(Insertter &ins);
};
```

Isto funciona bem até o ponto onde alguém queira incluir `fluxo.h` em outro arquivo fonte: devido à diretiva `'using'` esta outra pessoa também usando a diretiva `'using namespace especial'` por implicação, o que poderá produzir efeitos indesejados ou inesperados:

```
#include <fluxo.h>
#include <iostream>

using std::cout;

int main()
{
    cout << "começando" << endl;    // não compila
}
```

O compilador se vê confrontado com duas interpretações de `cout`: primeiro devido à diretiva `'using'` no arquivo cabeçalho `fluxo.h` ele considera `cout` um `especial::Extractor`, logo, devido à diretiva `'using'` no programa usuário, considera `cout` uma `std::ostream`. Como os compiladores fazem, quando confrontados a uma ambigüidade, geram uma nota de erro e vão embora.

Como regra prática: arquivos cabeçalho para uso geral não devem conter diretivas `'using'`.

Esta regra não vale para cabeçalhos para inclusão só em fontes de uma classe: Aqui o programador está livre para usar tantas diretivas 'using' quantas quizer, já que estas diretivas nunca chegam a outras fontes.

6.6: A palavra chave *'mutable'*

Anteriormente, na seção 6.2, os conceitos de funções membro constantes e objetos constantes foram introduzidos.

A linguagem C++, contudo, permite a construção de objetos que são, de certa forma, nem constantes nem não-constantes. Os membros de dados definidos com a palavra chave 'mutable' (mutável), podem ser modificados por funções membro constantes.

Um exemplo desta situação, onde mutável vem a ser útil, é onde um objeto constante necessita registrar o número de vezes que foi usado. Um exemplo onde um objeto constante da classe Mutável pode ser:

```
#include <string>
#include <iostream>
#include <memory>

class Mutável
{
    std::string d_nome;
    mutable int d_conta;           // usa a palavra chave mutable

public:
    Mutável(std::string const &nome)
    :
        d_nome(nome),
        d_conta(0)
    {}

    void called() const
    {
        std::cout << "Chamando " << d_nome <<
            " (passo " << ++d_count << ")\n";
    }
};

int main()
{
    Mutable const x("Objeto constante mutável");
}
```

```

    for (int idx = 0; idx < 4; idx++)
        x.called(); // modifica dados do objeto constante
}

/*
Saída Gerada:

Chamando Objeto constante mutável (passo 1)
Chamando Objeto constante mutável (passo 2)
Chamando Objeto constante mutável (passo 3)
Chamando Objeto constante mutável (passo 4)
*/

```

A palavra chave 'mutable' também pode ser útil na implantação de classes, p.ex., contagem de referência. Considere a classe que implanta a contagem de referência de cadeias de caracteres em textos. O objeto que realiza a contagem de referência é constante, mas a classe pode definir um construtor cópia. Como os objetos são constantes, como pode o construtor cópia ser capaz de implantar a contagem de referência? Aqui a palavra chave 'mutable' pode ser usada, já que pode incrementar e decrementar, mesmo objetos constantes.

A vantagem de ter uma palavra chave mutável é, finalmente, que o programador decide quais membros de dados podem ser modificados e quais não. Mas pode ser uma desvantagem: tendo a palavra chave à mão nos evita fazer suposições rígidas sobre a estabilidade de objetos constantes. Dependendo do contexto, isto pode ou não ser um problema. Na prática, mutável termina sendo útil só para controle interno: Acessoras que retornam valores de membros de dados mutáveis podem gerar um quebra-cabeça a clientes que as usam com objetos constantes. Nessas situações a natureza do valor retornado deve estar bem documentada. Como regra: não usar mutável a menos que seja por uma razão muito clara para divergir desta regra.

Capítulo 7: Classes e alocação de memória

Em contraste com o conjunto de funções que manipulam a alocação de memória na linguagem C (i.e., 'malloc()' etc.), os operadores 'new' e 'delete' são especificamente para usar-se com os recursos da linguagem C++. As diferenças importantes de 'new' em relação a 'malloc()' são:

- A função malloc() não 'conhece' para que a memória alocada será usada. P.ex., quando q memória é alocada para inteiros, o programador deve dar a expressão correta usando uma multiplicação por sizeof(int). Em contraste, 'new' requer o uso de um tipo; a expressão sizeof é manipulada implicitamente pelo compilador.
- A única maneira de iniciar a memória alocada por malloc() é usar calloc(), que aloca memória e a põe num valor determinado. Em contraste, 'new' chama o construtor de um objeto alocado e define ações iniciais. Este construtor é passado como argumento.
- Todas as funções C de alocação precisam de inspeção de retorno NULL. Em contraste, o operador 'new' fornece facilidades chamadas new-handler (seção 7.2.2) que pode ser usado em lugar de explicitamente testar o valor 0 de retorno.

Uma correlação comparável existe entre free() e delete: delete assegura que quando o objeto seja desalojado, um destrutor correspondente seja chamado.

A chamada automática de construtores e destrutores quando os objetos são criados e destruídos, tem muitas conseqüências que discutiremos neste capítulo. Muitos problemas encontrados durante o desenvolvimento de programas em linguagem C são causados por alocação incorreta de memória e fugas em memória: a memória não é alocada, não é liberada, não é iniciada, os limites são sobre-escritos, etc.. A linguagem C++ resolve estes problemas não magicamente, mas fornecendo certas ferramentas úteis.

Infelizmente as funções muito freqüentemente usadas str...(), como strdup() são todas baseadas em malloc() e, portanto, é preferível não usa-las mais em programas C++. Em lugar um novo conjunto de funções correspondentes, baseadas no operador 'new' são preferíveis. Por outro lado, desde que a classe string está disponível, há menos necessidade de usar essas funções em C++ que em C. Nos casos onde as operações com char * são preferíveis ou necessárias, funções comparáveis baseadas em 'new' podem ser desenvolvidas. P.ex., para a função strdup() uma função comparável char *strdupnew(char const *str) pode ser como segue:

```
char *strdupnew(char const *str)
{
```

```

        return str ? strcpy(new char [strlen(str) + 1], str) : 0;
    }

```

Neste capítulo os seguintes tópicos serão vistos:

- O operador adjudicador (e sobrecarga de operadores em geral);
- O ponteiro 'this';
- O construtor 'copy'.

7.1: Os operadores 'new' e 'delete'

A linguagem C++ define dois operadores para alocar e desalojar memória. Esses operadores são: 'new' e 'delete'.

O exemplo básico do uso desses operadores é dado abaixo. Um apontador a uma variável inteira é usado para apontar a memória alocada por 'new'. Esta memória é em seguida liberada por 'delete'.

```

int *ip;

ip = new int;
delete ip;

```

Note que 'new' e 'delete' e portanto não requerem parênteses, como as funções malloc() e free(). O operador 'delete' retorna void, o operador 'new' retorna um apontador para o tipo de memória pedido em seu argumento (p.ex, um apontador a um inteiro, como no exemplo acima). Note que o operador 'new' usa um tipo como operando, que tem o benefício da memória alocada possuir o tipo do objeto a ser alocado e, assim, fica imediatamente disponível. Ainda mais, este procedimento é inteiramente seguro enquanto ao tipo, já que 'new' retorna um ponteiro ao tipo de seu operando e o tipo do apontador deve coincidir com a variável que recebe o ponteiro.

O operador 'new' pode alocar tipos primitivos e objetos. Quando um tipo não-classe é alocado (tipo primitivo ou estrutura sem construtor), não há garantia da memória alocada estar iniciada com 0. Pode-se, opcionalmente, iniciá-la com uma expressão:

```

int *v1 = new int;           // sem garantia de iniciação em 0
int *v1 = new int();         // iniciada em 0
int *v2 = new int(3);        // iniciada em 3
int *v3 = new int(3 * *v2);  // iniciada em 9

```

Quando são alocados objetos com tipos de classe o construtor tem que ser mencionado e a

memória alocada será iniciada de acordo ao construtor usado. Por exemplo, para alocar um objeto string a seguinte instrução pode ser dada:

```
string *s = new string();
```

Aqui o construtor padrão foi usado e s apontará para a recém alocada string vazia. Se existem formas do construtor sobrecarregadas também se pode usar, p.ex.:

```
string *s = new string("Olá Mundo");
```

O que trará em s o endereço de uma string contendo o texto 'Olá Mundo'.

A alocação pode falhar. O que ocorre então veremos na seção 7.2.2.

7.1.1: Alocação de conjuntos

O operador 'new[]' é usado para alocar conjuntos. A notação genérica 'new[]' é uma abreviação usada nas Anotações C++. O número de elementos a serem alocados é especificado como uma expressão entre os colchetes prefixada pelo tipo dos valores ou classe dos objetos alocados:

```
int *intarr = new int[20];    // aloca 20 inteiros
```

Note bem que o operador 'new' é um operador diferente de 'new[]'. Na seção 9.9 veremos redefinindo o operador 'new'.

Os conjuntos alocados pelo operador 'new[]' são chamados conjuntos dinâmicos. São construídos durante a execução do programa e seu tempo de vida pode exceder o tempo de vida da função onde são criados. Os conjuntos dinamicamente alocados podem durar durante toda a execução do programa.

Quando se usa 'new[]' para alocar um conjunto de valores primitivos ou de objetos, 'new[]' precisa ser especificado com um tipo e uma expressão (sem sinal) dentro do colchete. O tipo e a expressão são usados pelo compilador para determinar o tamanho requerido do bloco de memória. Na alocação de conjuntos todos os elementos são dispostos consecutivamente na memória. A indexação do conjunto pode ser usada para acessar os elementos individualmente: intarr[0] será o primeiro valor inteiro, intarr[1] o seguinte e assim por diante até o último elemento: intarr[19]. Com tipos não-classe (tipos primitivos, estruturas sem construtor e apontadores) não há garantia do bloco de memória estar iniciado com 0.

Para alocar conjuntos de objetos a notação 'new[]' também é usada. Por exemplo, para alocar um conjunto de 20 strings a seguinte construção é usada:

```
string *strarr = new string[20];    // aloca 20 strings
```

Note que quando se alocam objetos, os construtores são automaticamente usados. Assim, 'new int[20]' resulta num bloco de 20 valores inteiros não iniciados, 'new string[20]' resulta num bloco de 20 objetos string iniciados. com conjunto de objetos o construtor padrão é usado. Infelizmente não é possível usar um conatrutor com argumentos na alocação de conjunto de objetos. Contudo, é possível sobrecarregar o operador 'new[]' fazendo-o aceitar parâmetros que podem ser usados para uma alocação não padronizada com iniciação de conjunto de objetos. A sobrecarga do operador 'new[]' é vista na seção 9.9.

Similar à linguagem C e sem recorrer ao operador 'new[]', os conjuntos de tamanho variável também podem ser construídos como conjuntos locais com funções. Tais conjuntos não são dinâmicos, mas locais e seu tempo de vida é restrito ao tempo de vida do bloco onde foram definidos.

Uma vez alocados todos os conjuntos são de tamanho fixo. Não há nenhum meio simples de aumentar ou diminuir um conjunto: não existe um operador 'renew'. Na seção 7.1.3 é dado um exemplo de como aumentar um conjunto.

7.1.2: Eliminação de conjuntos

Um conjunto dinamicamente alocado pode ser eliminado usando o operador 'delete[]'. O operador 'delete[]' espera um apontador ao bloco de memória previamente alocado usando o operador 'new[]'.

Quando um objeto é eliminado é chamado seu destrutor automaticamente (veja a seção 7.2), à similitude da chamada ao construtor quando o objeto é criado. É um trabalho do destrutor, como discutido a fundo mais tarde neste capítulo, para todo tipo de operação de limpeza é requerido o destrutor próprio do objeto.

O operador 'delete[]' (colchete vazio) espera como argumento um apontador a um conjunto de objetos. Este operador chamará o destrutor dos objetos individuais e eliminará o bloco de memória alocado. Assim, a maneira própria de eliminar um conjunto de objetos é:

```
Object *op = new Object[10];  
delete[] op;
```

Realizar esse 'delete[]' tem só um efeito adicional se o bloco de memória a ser liberado consiste de objetos. Com ponteiros ou valores de tipos primitivos normalmente não se efetua nenhuma ação especial. Acompanhando a 'int *it=new int[10]' a instrução 'delete' retorna a memória ocupada pelos 10 valores inteiros ao conjunto comum de memória livre. Nada especial acontece.

Note que um conjunto de ponteiros a objetos não é tratado como um conjunto de objetos por 'delete[]'. O conjunto de ponteiros a objetos não contém objetos, assim, os objetos não são propriamente eliminados por 'delete[]', como um conjunto que contém objetos, que são propriamente eliminados por 'delete[]'. Na seção 7.2 diversos exemplos do uso de 'delete' serão dados.

O operador 'delete' é um operador diferente do 'delete[]'. Na seção 9.9 redefinindo 'delete[]' ele é discutido. A regra prática é: se foi usado 'new[]' também use 'delete[]'.

7.1.3: Aumentando conjuntos

Uma vez alocados, todos os conjuntos são de tamanho fixo. Não existe um meio simples de se aumentar ou diminuir conjuntos: não existe o operador 'renew'. Nesta seção é dado um exemplo de como aumentar um conjunto. Aumentar um conjunto só é possível com conjuntos dinâmicos. Quando um conjunto precisa ser aumentado o seguinte procedimento deve ser empregado:

- Aloque um novo bloco de memória de maior tamanho;
- Copie o conteúdo do conjunto ao novo conjunto;
- Elimine o conjunto anterior (veja seção 7.1.2);
- Faça o ponteiro antigo apontar para o novo conjunto alocado.

O seguinte exemplo enfoca o aumento de um conjunto de objetos string:

```
#include <string>
using namespace std;

string *enlarge(string *old, unsigned oldsize, unsigned newsize)
{
    string *tmp = new string[newsize]; // aloca um conjunto maior

    for (unsigned idx = 0; idx < oldsize; ++idx)
        tmp[idx] = old[idx];           // copia old para tmp

    delete[] old;                       // usando [] são objetos

    return tmp;                         // retorna o novo conjunto
}

int main()
{
```

```

        string *arr = new string[4];           // inicialmente: conjunto de
4 strings

        arr = enlarge(arr, 4, 6);             // arr aumentado para 6
elementos.
    }

```

7.2: O Destrutor

Assim como as classes devem declarar um construtor, devem declarar um destrutor. Esta função é a oposta da função construtora no sentido de que é chamada para fazer com que o objeto deixe de existir. Para objetos locais e variáveis não estáticas, o destrutor é chamado quando o bloco onde o objeto foi definido é abandonado: Os destrutores de objetos definidos em blocos aninhados em funções usualmente são chamados justo antes da função retornar (terminar). Para variáveis estáticas ou globais o destrutor é chamado antes do programa terminar.

Contudo, quando um programa é interrompido usando-se uma chamada a `exit()`, os destrutores chamados são só os dos objetos globais existentes nesse momento. Os destrutores dos objetos definidos localmente em funções não são chamados quando forçamos um programa terminar com `exit()`.

A definição de um destrutor deve obedecer às seguintes regras:

- O destrutor tem o mesmo nome da classe mas seu nome é prefixado com um til.
- O destrutor não tem argumento nem retorna valor algum.

O destrutor para a classe `Pessoa` é declarado como segue:

```

class Pessoa
{
    public:
        Pessoa();           // construtor
        ~Pessoa();          // destrutor
};

```

A posição do construtor(es) e destrutor na definição de classe é ditada pela convenção: Primeiro o construtor é declarado, então o destrutor e só então os outros membros são declarados.

A tarefa principal de um destrutor é assegurar que a memória alocada pelo objeto (p.ex, pelo construtor) seja propriamente eliminada quando o objeto saia do escopo. Considere a seguinte definição da classe `Pessoa`:

```

        class Pessoa
    {
        char *d_nome;
        char *d_endereço;
        char *d_fone;

    public:
        Pessoa()
        {}
        Pessoa(char const *nome, char const *endereço,
                char const *fone);
        ~Pessoa();

        char const *nome() const;
        char const *endereço() const;
        char const *fone() const;
    };
/*
    pessoa.ih contains:

    #include "pessoa.h"
    char const *strdupnew(char const *org);
*/

```

A tarefa do construtor é iniciar os campos de dados do objeto. Pex., o construtor é definido como segue:

```

        #include "pessoa.ih"

        Pessoa::Pessoa(char const *nome, char const *endereço, char const
*fone)
        :
            d_nome(strdupnew(nome)),
            d_endereço(strdupnew(endereço)),
            d_fone(strdupnew(fone))
        {}

```

Nesta classe o destrutor é necessário para evitar que a memória alocada para os campos nome, endereço, e fone fique inacessível quando um objeto deixe de existir, produzindo, assim, fuga de memória. O destrutor de um objeto é chamado automaticamente:

- Quando um objeto sai do escopo;
- Quando um objeto alocado dinamicamente é eliminado;
- Quando um conjunto de objetos alocado dinamicamente é eliminado o operador usando 'delete[]'

(veja a seção 7.1.2).

Como é tarefa do destrutor liberar a memória dinamicamente alocada e usada pelo objeto, a tarefa do destrutor de Pessoa é liberar a memória apontada pelos três ponteiros dos membros de dados. A implantação do destrutor, então, será:

```
#include "pessoa.ih"

Pessoa::~Pessoa()
{
    delete d_nome;
    delete d_endereço;
    delete d_fone;
}
```

No exemplo seguinte um objeto Pessoa é criado e seus campos de dados são impressos. Depois a função showPessoa() para, resultando na liberação da memória. Note que no exemplo o objeto da classe Pessoa é criado e destruído dinamicamente pelos operadores, respectivamente, 'new' e 'delete'.

```
#include "pessoa.h"
#include <iostream>

void showPessoa()
{
    Pessoa karel("Karel", "Marskramerstraat", "038 420 1971");
    Pessoa *frank = new Pessoa("Frank", "Oostumerweg", "050 403
2223");

    cout << karel.nome()      << ", " <<
        karel.endereço()    << ", " <<
        karel.fone()        << endl <<
        frank->nome()        << ", " <<
        frank->endereço()    << ", " <<
        frank->fone()        << endl;

    delete frank;
}
```

A memória ocupada pelo objeto 'karel' é automaticamente liberada quando showPessoa() termina: O compilador C++ assegura a chamada ao destrutor. Note contudo, que o objeto apontado por 'frank' é manipulado de outra forma. Por isso, antes de main() terminar a memória ocupada pelo objeto apontado por 'frank' deve ser explicitamente eliminado; daí a instrução 'delete frank'. O operador assegura a chamada ao destrutor e, portanto, a eliminação dos três 'char * string' do objeto.

7.2.1: 'New' e 'delete' e apontadores a objetos

Os operadores 'new' e 'delete' são usados quando um objeto de dada classe é alocado. Como vimos, uma das vantagens dos operadores 'new' e 'delete' sobre funções como malloc() e free() é que 'new' e 'delete' chamam os correspondentes construtores e destrutores. Isto está ilustrado no seguinte exemplo:

```
Pessoa *pp = new Pessoa(); // ponteiro ao objeto Pessoa

delete pp;                // agora destruído
```

A alocação de um novo objeto Pessoa apontado por pp é um processo de dois passos. Primeiro, a memória para o objeto é alocada. Segundo, o construtor é chamado, inicia o objeto. No exemplo acima o construtor é uma versão sem argumento; mas é possível o uso de um construtor com argumentos:

```
frank = new Pessoa("Frank", "Oostumerweg", "050 403 2223");
delete frank;
```

Note que analogamente à construção de um objeto, a destruição também é um processo de dois passos: Primeiro, o destrutor da classe é chamado para eliminar a memória alocada e o objeto usado; então a memória usada pelo objeto é liberada.

Conjuntos de objetos dinamicamente alocados também pode ser manipulados por 'new' e 'delete'. Neste caso o tamanho do conjunto é dado dentro do colchete [] quando o conjunto é criado:

```
Pessoa *pessoaarray = new Pessoa [10];
```

O compilador gerará o código para chamar o construtor padrão para cada objeto criado. Como vimos na seção 7.1.2, o operador 'delete[]' tem que ser usado aqui para eliminar o conjunto de maneira apropriada:

```
delete[] pessoaarray;
```

A presença de [] assegura que o destrutor é chamado para cada objeto do conjunto.

Que passa se usarmos 'delete' no lugar de 'delete[]'? Considere a seguinte situação na qual o destrutor ~Pessoa() é modificado e nos diga que foi chamado. Numa função main() um conjunto de dois objetos Pessoa são alocados por 'new[]', para serem eliminados por 'delete[]'. Em seguida as mesmas ações são repetidas, se bem que o operador 'delete' é chamado sem []:

```
#include <iostream>
#include "pessoa.h"
using namespace std;

Pessoa::~~Pessoa()
```

```

    {
        cout << "Destrutor de Pessoa chamado" << endl;
    }

int main()
{
    Pessoa *a  = new Pessoa[2];

    cout << "Destruição com []" << endl;
    delete [] a;

    a = new Pessoa[2];

    cout << "Destruição sem []" << endl;
    delete a;

    return 0;
}
/*
    Saída Gerada:
Destruição com []
Destrutor de Pessoa chamado
Destrutor de Pessoa chamado
Destruição sem []
Destrutor de Pessoa chamado
*/

```

Olhando para a saída gerada vemos que o destrutor de objetos Pessoa individuais é chamado duas vezes se a sintaxe 'delete[]' é seguida e uma vez se [] é omitido.

Se não foi definido um destrutor ele não é chamado. Esta pode parecer uma afirmação trivial, mas tem severas implicações: os objetos alocados em memória resultam em vazamento de memória quando não se define um destrutor. Considere o seguinte programa:

```

#include <iostream>
#include "pessoa.h"
using namespace std;

Pessoa::~Pessoa()
{
    cout << "Destrutor de Pessoa chamado" << endl;
}

int main()
{
    Pessoa **a = new Pessoa* [2];

```



```

    a[0] = new Pessoa[2];
    a[1] = new Pessoa[2];

    delete [] a;

    return 0;
}

```

Este programa não produz saída nenhuma. Por que? A variável 'a' é definida como ponteiro a um ponteiro. Para esta situação não há um destrutor definido. Conseqüentemente o colchete [] é ignorado.

Como o [] é ignorado, somente o conjunto é eliminado, porque aqui 'delete[] a' elimina a memória apontada por 'a'. Isto é tudo.

Claro que não queremos isto, mas eliminar os objetos Pessoa apontados pelos elementos de 'a' também. Neste caso temos duas opções:

- Percorrer explicitamente todos os elementos do conjunto 'a' eliminando-os por turno. Isto chamará o destrutor para um ponteiro dos objetos Pessoa que destruirá todos os elementos se o operador [] for usado, como em:

```

#include <iostream>
#include "pessoa.h"

Pessoa::~Pessoa()
{
    cout << "Destrutor de Pessoa chamado" << endl;
}

int main()
{
    Pessoa **a = new Pessoa* [2];

    a[0] = new Pessoa[2];
    a[1] = new Pessoa[2];

    for (int index = 0; index < 2; index++)
        delete [] a[index];

    delete[] a;
}
/*
Saída Gerada:

```

```

Destrutor de Pessoa chamado
Destrutor de Pessoa chamado
Destrutor de Pessoa chamado
Destrutor de Pessoa chamado
*/

```

- Definir uma classe de cobertura contendo um ponteiro aos objetos Pessoa e alocar um ponteiro a esta classe, antes que um ponteiro ao ponteiro aos objetos Pessoa. O tópico de uma classe conter outra, composição, foi discutido na seção 6.4. Eis um exemplo mostrando a eliminação de ponteiros a memória usando tal classe de cobertura:

```

#include <iostream>
using namespace std;

class Informer
{
public:
    ~Informer()
    {
        cout << "destrutor chamado\n";
    }
};

class Wrapper //classe de cobertura
{
    Informer *d_i;

public:
    Wrapper()
    :
        d_i(new Informer())
    {}
    ~Wrapper()
    {
        delete d_i;
    }
};

int main()
{
    delete [] new Informer *[4];        // vazamento de memória:
destrutor não                            // foi chamado

    cout << "=====\n";

    delete [] new Wrapper[4];           // ok: 4 x destrutor chamado

```

```

}
/*
    Saída Gerada:
    =====
    destrutor chamado
    destrutor chamado
    destrutor chamado
    destrutor chamado
*/

```

7.2.2: A função `set_new_handler()`

O sistema de execução da linguagem C++ assegura quando a alocação de memória falhe uma função de erro seja ativada. Como padrão esta função lança uma (`bad_alloc`) exceção() (veja seção 8.10), terminando o programa. Conseqüentemente no caso padrão nunca é necessário examinar o retorno do operador 'new'. Este comportamento padrão pode ser modificado de várias maneiras. Uma dessas maneiras é redefinir a função que manipula a falha de alocação de memória. De todas formas, qualquer função definida pelo usuário deve cumprir os seguintes pré-requisitos:

- não possuir argumentos e
- não retornar nenhum valor.

A função de erro redefinida deve, p.ex., imprimir uma mensagem e terminar o programa. A função de erro escrita pelo usuário se torna parte do sistema de alocação através da função `set_new_handler()`.

A implantação de uma função de erro está ilustrada abaixo (Esta implantação aplica os requerimentos Gnu C/C++. Não se recomenda o uso do exemplo, pois retardará enormemente o computador devido ao uso resultante da área de swap do Unix):

```

#include <iostream>
using namespace std;

void outOfMemory()
{
    cout << "Memória esgotada. Programa terminado." << endl;
    exit(1);
}

int main()
{
    long allocated = 0;

```

```

set_new_handler(outOfMemory);          // instala a função de erro

while (true)                            // consome toda a memória
{
    new int [100000];
    allocated += 100000 * sizeof(int);
    cout << "Alocados " << allocated << " bytes\n";
}
}

```

Depois de instalar a função de erro ela é automaticamente invocada quando a alocação de memória falha e o programa termina. Note que a alocação de memória pode falhar na chamada indireta de código também, p.ex., na construção ou uso de streams ou quando da duplicação de strings por funções de baixo nível.

Note que não se deve assumir que as funções padrão da linguagem C de alocação de memória, tais como `strdup()`, `malloc()`, `realloc()`, etc. farão partir o novo manipulador quando a alocação de memória falhar. Isto significa que uma vez instalado um novo manipulador tais funções não devem ser usadas automaticamente de modo desprotegido em programas C++. Um exemplo usando 'new' para duplicar uma string foi dado numa re-escritura da função `strdup()` (veja seção 7.3).

7.3: O operador adjudicação

As variáveis do tipo estruturas ou classes podem ser diretamente adjudicadas em C++ da mesma forma que em C. A ação padrão dessa adjudicação para tipos não membros de dados de classes é uma cópia direta byte a byte de um membro de dados a outro. Agora considere as consequências desta ação padrão numa função como a seguinte:

```

void printpessoa(Pessoa const &p)
{
    Pessoa tmp;

    tmp = p;
    cout << "Nome:      " << tmp.nome()      << endl <<
         "Endereço:   " << tmp.endereço()    << endl <<
         "fone:      " << tmp.fone()        << endl;
}

```

Seguiremos a execução desta função passo a passo.

- A função `printpessoa()` espera uma referência a `Pessoa` como seu parâmetro 'p'. Adiante nada extraordinário está acontecendo.

- A função define um objeto local tmp. Isto significa que o construtor padrão de Pessoa é chamado, que -se definido propriamente- zera os ponteiros dos campos nome, endereço e fone do objeto tmp.
- Em seguida o objeto referenciado por 'p' é copiado a tmp. Como padrão isto significa que sizeof(Pessoa) bytes de 'p' são copiados a tmp.

Atingimos uma situação potencialmente perigosa. Note que os valores em 'p' são ponteiros à memória alocada. Seguindo a adjudicação essa memória está endereçada por dois objetos: 'p' e 'tmp'.

- A situação de potencial se desenvolveu em agudamente perigosa quando a função printpessoa() termine, o objeto 'tmp' será destruído. O destrutor da classe Pessoa libera a memória apontada pelos campos nome, endereço e fone: infelizmente essa memória também é apontada por 'p'....

A adjudicação incorreta é ilustrada na Figura 5.

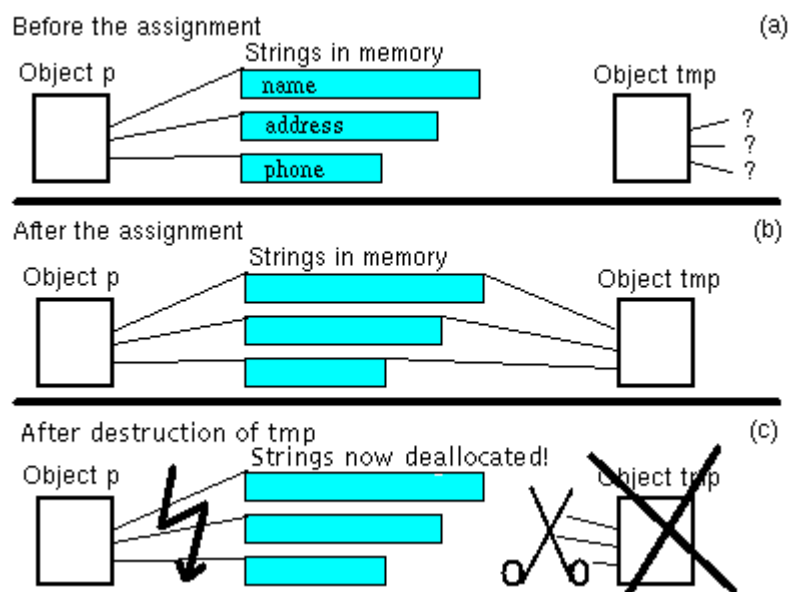


Figura 5 (Private data and public interface functions of the class Person, using byte-by-byte assignment) about here (file: memory/badassign) -----

Depois de executar `printpessoa()`, o objeto que era referenciado por 'p' contém agora apontadores a uma memória vazia.

Esta situação é indubitavelmente indesejável para uma função como a acima. A memória liberada será igualmente ocupada durante alocações subsequentes: os membros apontadores de 'p' se tornaram indubitavelmente selvagens, já que não apontam à memória alocada. Em geral se pode concluir que:

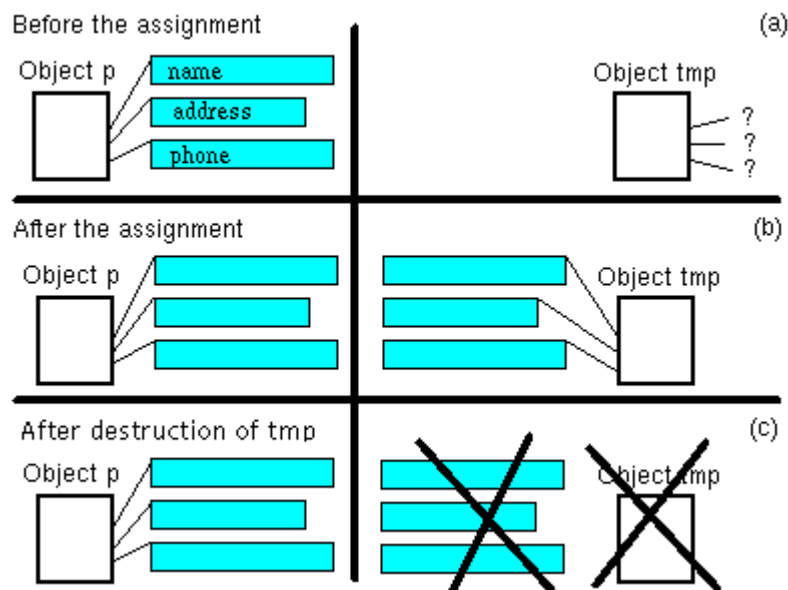
Toda classe que contém ponteiros como membros é uma candidata potencial de problemas.

Afortunadamente é possível prever esses problemas, como veremos na próxima seção.

7.3.1: Sobrecarga do operador adjudicação

Obviamente a maneira correta de adjudicar um objeto Pessoa a outro não é copiando o conteúdo do objeto byte a byte. A melhor maneira é fazer outro objeto equivalente; um com sua memória própria de alocação, mas que contém as mesmas strings.

A maneira correta de duplicar um objeto Pessoa é ilustrada na Figura 6.



-----Insert Figure
 6(Private data and public interface functions of the class Person,
 using the 'correct' assignment.)about here (file:
 memory/rightass)-----

Existem diversas maneiras de duplicar um objeto Pessoa. Uma maneira seria definir uma função membro para manipular adjudicações de objetos da classe Pessoa. O propósito dessa função membro seria criar uma cópia de um objeto, mas uma com suas próprias strings nome, endereço e fone. Tal função poderia ser:

```
void Pessoa::assign(Pessoa const &outra)
{
    // elimina a memória previamente usada
    delete d_nome;
    delete d_endereço;
    delete d_fone;

    // copia os dados de outra Pessoa
    d_nome = strdupnew(outra.d_nome);
    d_endereço = strdupnew(outra.d_endereço);
    d_fone = strdupnew(outra.d_fone);
}
```

Usando esta ferramenta pudemos re-escrever a função printpessoa():

```
void printpessoa(Pessoa const &p)
{
    Pessoa tmp;

    // faz uma cópia de p em tmp, mas com sua própria alocação de
memória
    tmp.assign(p);

    cout << "Nome:      " << tmp.nome() << endl <<
        "Endereço:   " << tmp.endereço() << endl <<
        "Fone:      " << tmp.fone() << endl;

    // agora não importa se tmp for destruído..
```

}

Em si a solução é válida, porém é uma solução sintomática. Esta solução requer que o programador use uma função membro específica no lugar do operador igual (=). O problema básico, contudo, persiste se não se segue estritamente a regra. A experiência mostra que errare humanum est: uma solução que não force ações especiais é preferível.

O problema do operador adjudicação é resolvido usando-se sobrecarga do operador: As possibilidades sintáticas da linguagem C++ oferece a possibilidade de se redefinir as ações de um operador num contexto dado. A sobrecarga de operadores foi mencionada anteriormente, quando os operadores << e >> foram redefinidos para seu uso em streams como cin, cout e cerr (veja seção 3.1.2).

A sobrecarga do operador adjudicação talvez seja a forma mais comum de sobrecarga de um operador. Mas, uma palavra de precaução é apropriada: O fato de que a linguagem C++ permita a sobrecarga de operadores não significa que esta característica tenha que se usar sempre. Algumas regras são:

- Pode-se usar a sobrecarga de um operador quando a ação de um operador pode provocar efeitos colaterais negativos. Um exemplo é o acima, o efeito do operador adjudicação (=) no contexto da classe Pessoa.
- A sobrecarga de um operador pode ser usada em situações onde o uso do operador é comum e quando não se introduz ambigüidade no significado do operador ao redefini-lo. Um exemplo de redefinição do operador + pode ser a feita para se usá-lo numa classe com números complexos. O significado do operador + entre dois complexos é claro e sem ambigüidades.
- Em todos os outros casos é preferível definir uma função membro no lugar de redefinir um operador.

Utilizando estas regras a sobrecarga de operadores é minimizada o que ajuda a conservar a legibilidade das fontes. Um operador faz simplesmente aquilo para o que foi projetado. Por isso, em nossa visão, os operadores inserção (<<) e extração (>>) no contexto das streams são infortunados: As operações com streams nada têm a ver com operações bit a bit.

7.3.1.1: O 'operator=()' membro

Para se levar a cabo a sobrecarga de um operador no contexto de uma classe, a classe é simplesmente expandida com uma função membro (usualmente pública) significando o operador particular. A função membro é então definida.

Por exemplo, para sobrecarregar o operador adjudicação (=), uma função `operator=()` precisa ser definida. Note que o nome da função é composto de duas partes: A palavra chave 'operator' seguida pelo próprio operador. Quando aumentamos uma interface de classe com uma função membro '`operator=()`', então o operador é redefinido para a classe, o que evita o uso do operador padrão. Anteriormente (na seção 7.3.1) a função `assign()` foi oferecida para resolver os problemas de memória resultantes do uso do operador 'assign' padrão. Contudo, no lugar de usar uma função membro ordinária é muito mais comum em C++ definir um operador dedicado a estes casos especiais. Assim, o membro anterior `assign()` é redefinido como segue (note que o membro '`operator=()`' apresentado abaixo é uma primeira versão, bem menos sofisticada da sobrecarga do operador adjudicação. Ele será melhorado um pouco):

```
class Pessoa
{
    public:                                // extensão da classe Pessoa
                                           // são assumidos membros
anteriores.
    void operator=(Pessoa const &outro);
};
```

e sua implantação poderia ser:

```
void Pessoa::operator=(Pessoa const &outro)
{
    delete d_nome;                        // elimina dados anteriores
    delete d_endereço;
    delete d_fone;

    d_nome = strdupnew(outro.d_nome);      // duplica os dados de outro
    d_endereço = strdupnew(outro.d_endereço);
    d_fone = strdupnew(outro.d_fone);
}
```

As ações da função membro são similares àsquelas da função `assign()` anteriormente proposta, mas seu nome assegura que esta função também é ativada quando o operador = seja usado. Existem duas maneiras de se chamar os operadores sobrecarregados:

```
Pessoa pers("Frank", "Oostumerweg", "403 2223");
Pessoa copia;

copia = pers;                            // primeira possibilidade
copia.operator=(pers);                    // segunda possibilidade
```

Atualmente a segunda possibilidade, chamada explícita '`operator=()`' não é muito usada. Contudo o fragmento de código ilustra duas formas de se chamar a função membro operador sobrecarregado.

7.4: O ponteiro `this`

Como vimos, uma função membro de dada classe é sempre chamada no contexto de algum objeto da classe. Existe sempre um 'substrato' implícito para a função atuar. A linguagem C++ define a palavra chave, `this`, para endereçar o substrato (note que `this` não está disponível entre as funções membro estáticas, não discutidas ainda)

Esta palavra chave é um ponteiro variável, que sempre contém o endereço do objeto em questão. O ponteiro `this` é declarado implicitamente em cada função membro (em lugar de público, protegido ou privado). Por isso é como se cada função membro da classe Pessoa tivesse a seguinte declaração:

```
extern Pessoa *const this;
```

Uma função membro como `nome()`, que retorna o campo `nome` de uma Pessoa, pode ser implantada de duas formas: Com ou sem o ponteiro `this`:

```
char const *Pessoa::nome()    // uso implícito de `this'
{
    return d_nome;
}

char const *Pessoa::nome()    // uso explícito de `this'
{
    return this->d_nome;
}
```

O ponteiro `this` não é frequentemente usado explicitamente. Contudo existem situações onde este ponteiro é requerido (veja Capítulo 15).

7.4.1: Evitando a auto-destruição com o uso de `this`

Como vimos, o operador `=` pode ser redefinido para a classe Pessoa de formas que os objetos da classe podem ser adjudicados, resultando na cópia de dois objetos idênticos.

Enquanto as duas variáveis sejam diferentes, a versão previamente apresentada da função `operator=()` atuará propriamente: a memória do objeto adjudicado será liberada, depois do que será alocada para conter strings. Contudo, quando um objeto é adjudicado a si mesmo (que chamamos de auto-adjudicação), um problema ocorre: A string alocada do objeto que recebe primeiro é eliminada, resultando na eliminação da memória do objeto à direita o que chamamos auto-destruição. Eis um

exemplo dessa situação:

```
void fubar(Pessoa const &p)
{
    p = p;           // auto-adjudicação!
}
```

Nesta exemplo é perfeitamente claro que algo inecessário, possivelmente errado, está ocorrendo. Mas auto-adjudicação pode se dar em forma velada:

```
Pessoa um;
Pessoa dois;
Pessoa *pp = &um;

*pp = dois;
um = *pp;
```

O problema da auto-adjudicação pode ser resolvido usando-se o ponteiro 'this'. No operador de adjudicação sobrecarregado simplesmente testando se o objeto da direita é idêntico ao da esquerda: se for nenhuma ação se realizará. A definição do operator=() se torna:

```
void Pessoa::operator=(Pessoa const &outro)
{
    // só realiza ação se o endereço do objeto atual
    // (this) NÃO é igual ao endereço do objeto outro

    if (this != &outro)
    {
        delete d_nome;
        delete d_endereço;
        delete d_fone;

        d_nome = strdupnew(outro.d_nome);
        d_endereço = strdupnew(outro.d_endereço);
        d_fone = strdupnew(outro.d_fone);
    }
}
```

Esta é a segunda versão da função adjudicação sobrecarregada. Mais uma, ainda melhor fica para ser discutida.

Como sutileza, note que o uso do endereço com o operador '&' na instrução:

```
if (this != &outro)
```

A variável 'this' é um apontador ao objeto corrente, enquanto 'outro' é a referência; que é um 'apelido' do objeto Pessoa atual. O endereço do outro objeto é, portanto, '&outro', enquanto o endereço do

objeto atual é 'this'.

7.4.2: Associatividade de operadores e 'this'

De acordo com a sintaxe C++ o operador adjudicação se associa da direita para a esquerda, i.e., em instruções como:

```
a = b = c;
```

A expressão `b = c` é avaliada primeiro e o resultado é adjudicado a 'a'.

A implantação do operador adjudicação não permite tais construções, já que uma adjudicação usando a função membro não retorna nada (void). Assim podemos concluir que a implantação vista resolve o problema de alocação, mas adjudicações concatenadas ainda não são permitidas.

O problema pode ser ilustrado como segue. Quando re-escrevemos a expressão `a = b = c` na forma que explicitamente menciona a função membro adjudicação sobrecarregada, chegamos a:

```
a.operator=(b.operator=(c));
```

Esta forma está sintaticamente errada, já que a sub-expressão `b.operator=(c)` não retorna nada. Contudo a classe Pessoa não contém função membro com o protótipo `operator= (void)`.

Este problema também pode ser remediado usando o ponteiro 'this'. A função adjudicação sobrecarregada espera como argumento seu uma referência a um objeto Pessoa. Também pode retornar uma referência a tal objeto. Esta referência pode então ser usada como argumento numa adjudicação concatenada.

É habitual fazer a adjudicação sobrecarregada retornar uma referência ao objeto corrente (i.e., '*this'). A versão (final) do operador adjudicação sobrecarregado para a classe Pessoa então fica:

```
Pessoa &Pessoa::operator=(Pessoa const &outro)
{
    if (this != &outro)
    {
        delete d_endereço;
        delete d_nome;
        delete d_fone;

        d_endereço = strdupnew(outro.d_endereço);
        d_nome = strdupnew(outro.d_nome);
        d_fone = strdupnew(outro.d_fone);
    }
}
```

```

    }
    // retorna o objeto corrente. O compilador assegurará
    // que uma referência seja retornada
    return *this;
}

```

7.5: A construção de cópia: iniciação versus adjudicação

Nas seções seguintes veremos mais de perto o uso do operador =. Considere, uma vez mais, a classe Pessoa. A classe tem as seguintes características:

- A classe contém diversos ponteiros, possivelmente apontando para memória alocada. Como discutido, tal classe necessita construtor e destrutor. Uma ação típica do construtor seria zerar os membros apontadores. Uma ação típica do destrutor seria liberar a memória alocada.
- Pela mesma razão a classe requer um operador de adjudicação sobrecarregado.
- A classe tem, além de um construtor padrão, um construtor que espera os objetos membros de Pessoa nome, endereço e número telefônico.
- Por agora a única função interface retorna os objetos de Pessoa nome, endereço e número telefônico.

Agora considere o seguinte fragmento de código. As referências aos comandos são discutidas em seguida ao exemplo:

```

    Pessoa karel("Karel", "Marskramerstraat", "038 420 1971"); // veja
(1)
    Pessoa karel2; // veja
(2)
    Pessoa karel3 = karel; // veja
(3)

int main()
{
    karel2 = karel3; // veja (4)
    return 0;
}

```

- Comando 1: Mostra a iniciação. O objeto karel é iniciado com textos apropriados. A construção de karel portanto usa o construtor que espera três argumentos 'const *'.
- Assuma um construtor com só um parâmetro 'char const *', p.ex.:

```
Pessoa::Pessoa(char const *n);
```

- Pode-se notar que a iniciação 'Pessoa frank("Frank")' é idêntica a:

```
Pessoa frank = "Frank";
```

- Este pedaço de código usa o operador =, não há adjudicação: melhor dito é uma iniciação e portanto é feita durante a construção pelo construtor da classe Pessoa.
- Comando 2: Aqui um segundo objeto Pessoa é criado. novamente um construtor é chamado. Como não estão presentes argumentos especiais, o construtor padrão é usado.
- Comando 3: Outra vez um novo objeto karel3 é criado. Um construtor é chamado outra vez. O novo objeto também é iniciado. Desta vez com uma cópia do objeto karel.

- Esta forma de iniciação ainda não foi discutida. Podemos reescrever este comando como:

```
Pessoa karel3(karel);
```

- Fica sugerido que um construtor é chamado, tendo uma referência a um objeto Pessoa como argumento. Tais construtores são perfeitamente comuns na linguagem C++ e chamados construtor de cópia.
- Comando 4: Aqui um objeto é adjudicado a outro. Não há criação de objeto neste comando. Esta é uma adjudicação simplesmente, usando o operador de adjudicação sobrecarregado.
- A regra simples que emana destes exemplos é que sempre que um objeto é criado é necessário o construtor. Todos os construtores têm as seguintes características:
- Os construtores não retornam nenhum valor.
- Os construtores são definidos por funções com o mesmo nome da classe a que pertencem.
- O construtor a ser usado no momento é deduzido a partir da lista de argumentos. O operador adjudicação pode ser usado, no caso do construtor possuir só um parâmetro (e também quando os parâmetros restantes têm valores padrão de seus argumentos).

Portanto concluímos que dado o comando (3) acima a classe Pessoa deve ser aumentada com um construtor cópia:

```
class Pessoa
{
    public:
```

```

        Pessoa(Pessoa const &outro);
};

```

A implantação do construtor cópia de Pessoa é:

```

Pessoa::Pessoa(Pessoa const &outro)
{
    d_nome      = strdupnew(outro.d_nome);
    d_endereço  = strdupnew(outro.d_endereço);
    d_fone      = strdupnew(outro.d_fone);
}

```

As ações de um construtor cópia são comparáveis àquelas do operador adjudicação sobrecarregado: um objeto é duplicado, assim contém os mesmos dados alojados. O construtor cópia, contudo, é mais simples nos seguintes aspectos:

- Um construtor de cópia não necessita apagar a memória previamente alocada: Já que o objeto em questão acaba de ser criado.
- Um construtor de cópias não necessita nunca examinar se ocorreu uma auto-duplicação. Nenhuma variável pode ser iniciada por si mesma.

Além dos usos óbvios, mencionados acima, do construtor de cópias, ele possui outras tarefas importantes. Todas essas tarefas estão relacionadas ao fato de que o construtor de cópias sempre é chamado quando um objeto é iniciado usando outro objeto da classe. O construtor de cópias é chamado mesmo que o objeto seja vazio ou uma variável temporária.

- Quando uma função toma um objeto como argumento, noutro lugar de, p.ex., um ponteiro de referência, o construtor de cópias é chamado para passar uma cópia do objeto como argumento. Este argumento, que usualmente é passado através da pilha (stack), é portanto, um novo objeto. É criado e iniciado com os dados do argumento passado.

Isto está ilustrado no seguinte fragmento de código:

```

void nomeDe(Pessoa p)          // sem ponteiro, sem referência
{
    // mas a própria Pessoa
    cout << p.nome() << endl;
}

int main()
{
    Pessoa frank("Frank");

    nomeDe(frank);
}

```

```

        return 0;
    }

```

Nesse fragmento de código, `frank`, não é passado como argumento, mas é criada como variável temporária (na pilha) usando o construtor de cópias. Esta cópia temporária conhecida dentro de `nomeDe()` como `p`. Note que `nomeDe()` poderia ter um parâmetro, então se usaria mais pilha e uma chamada ao construtor de cópias seria economizada.

- O construtor de cópias também é chamado implicitamente quando uma função retorna um objeto:

```

Pessoa pessoa()
{
    string nome;
    string endereço;
    string fone;

    cin >> nome >> endereço >> fone;

    Pessoa p(nome.c_str(), endereço.c_str(), fone.c_str());

    return p;           // retorna uma cópia de 'p'.
}

```

Aqui um objeto vazio da classe `Pessoa` é iniciado usando o construtor de cópias, como valor retornado pela função. A variável local `p` deixa de existir quando `pessoa()` termina.

Para demonstrar que o construtor de cópias não é chamado em todas situações, considere o seguinte. Podemos reescrever a função acima `pssoa()` da seguinte forma:

```

Pessoa pessoa()
{
    string nome;
    string endereço;
    string fone;

    cin >> nome >> endereço >> fone;

    return Pessoa(nome.c_str(), endereço.c_str(), fone.c_str());
}

```

Este fragmento de código é perfeitamente válido e ilustra o uso de um objeto anônimo. Os objetos anônimos são objetos constantes: Seus dados membros não podem mudar. O uso de um objeto anônimo na exemplo acima ilustra o fato que os valores retornados por objetos são objetos constantes, mesmo que a palavra chave `const` não esteja explícita.

Como outro exemplo, mais uma vez assumindo a disponibilidade do construtor Pessoa(char const *nome), considere:

```
Pessoa nomePessoa()  
{  
    string nome;  
  
    cin >> nome;  
    return nome.c_str();  
}
```

Aqui o valor de retorno nome.c_str() não coincide com o tipo retornado Pessoa, há um construtor disponível para construir um objeto Pessoa de char const *. Desde que tal construtor exista, o valor (anônimo) retornado pode ser construído promovendo char const * a um tipo Pessoa usando um construtor apropriado.

Ao contrário da situação que encontramos com o construtor padrão, o construtor de cópias padrão continua disponível uma vez que um construtor (qualquer construtor) seja definido explicitamente. O construtor de cópias pode ser definido, mas se não, o construtor de cópias padrão continua disponível quando outro construtor é definido.

7.5.1: Similaridades entre o construtor de cópias e o operador=()

As similaridades entre o construtor de cópias e o operador de adjudicação sobrecarregado são investigadas nesta seção. Apresentamos aqui duas funções primitivas que ocorrem com frequência em nossos códigos e que pensamos são muito úteis. Note as seguintes características do construtor de cópias, operadores de adjudicação sobrecarregado e destrutores:

- Ocorrem cópias de dados (privados) (1) no construtor de cópias e (2) na função de adjudicação sobrecarregada.
- A destruição de memória alocada ocorre (1) na função de adjudicação sobrecarregada e (2) no destrutor.

As duas ações acima (duplicação e destruição) podem ser implantadas em duas funções privadas, digamos copy() e destroy(), que são usadas que são usadas no operador de adjudicação sobrecarregado, construtor de cópias e destrutor. Quando aplicamos este método à classe Pessoa podemos implantar da seguinte maneira:

- Primeiro, a definição de classe é expandida com duas funções copy() e destroy(). O propósito

dessas funções são copiar dados a outro objeto ou apagar a memória do objeto corrente incondicionalmente. Portanto essas funções implantam funcionalidades 'primitivas':

```
// definição de classe, só funções relevantes são mostradas aqui
class Pessoa
{
    char *d_nome;
    char *d_endereço;
    char *d_fone;

    public:
        Pessoa(Pessoa const &outro);
        ~Pessoa();
        Pessoa &operator=(Pessoa const &outro);
    private:
        void copy(Pessoa const &outro);      // novos membros
        void destroy(void);

};
```

- Em seguida as funções copy() e destroy() são construídas:

```
void Pessoa::copy(Pessoa const &outro)
{
    d_nome = strdupnew(outro.d_nome);          // cópia incondicional
    d_endereço = strdupnew(outro.d_endereço);
    d_fone = strdupnew(outro.d_fone);
}

void Pessoa::destroy()
{
    delete d_nome;                             // eliminação incondicional
    delete d_endereço;
    delete d_fone;
}
```

o Finalmente as funções públicas onde a memória do objeto 'outro' é copiado ou é apagado são reescritas:

```
Pessoa::Pessoa (Pessoa const &outro)      // construtor de cópias
{
    copy(outro);
}

Pessoa::~~Pessoa()                        // destrutor
{
    destroy();
}
```

```

// adjudicação sobrecarregada
Pessoa const &Pessoa::operator=(Pessoa const &outro)
{
    if (this != &outro)
    {
        destroy();
        copy(outro);
    }
    return *this;
}

```

Aquilo que gostamos deste modelo é que o destrutor, o construtor de cópias e a função de adjudicação sobrecarregada agora são completamente padrão: São independentes de qualquer classe particular e sua implantação pode, portanto, ser usada em qualquer classe. Qualquer dependência de classe está reduzida à implantação dos membros privados `copy()` e `destroy()`.

Note que a função membro `copy()` é responsável pela cópia de campo de dados de outro objeto ao corrente. Mostramos a situação quando uma classe só possui ponteiros aos membros de dados. Na maioria dos casos as classes não possuem dados que sejam ponteiros. Estes membros precisam ser copiados com o construtor de cópias. Isto pode ser realizado simplesmente pelo corpo do construtor exceto pela iniciação dos membros que precisam ser iniciados pelo método iniciador introduzido na seção 6.4.2. Contudo, nesta classe, o operador de adjudicação sobrecarregado não pode ser implantado completamente tão pouco, já que não se pode dar outros valores aos membros de referência, uma vez iniciados. Um objeto com membros de dados de referência está inseparavelmente ligado ao objeto referenciado uma vez construído.

7.5.2: Evitando o uso de certos membros

Como vimos na seção anterior, pode-se encontrar a situação onde uma função membro não pode realizar sua tarefa de modo satisfatório. Em particular um operador de adjudicação sobrecarregado não pode fazer seu trabalho completamente se a classe contém membros de referência a dados. Nesta situação e em outras comparáveis o programador deve evitar o uso (acidental) de certas funções membro. Isto pode ser feito do seguinte modo:

- Mova todas as funções que não devem ser chamadas para a região privada da interface de classe. Isto evitará efetivamente o uso desses membros pelos usuários da classe. Movendo o operador de adjudicação para a região privada os objetos da classe não poderão ser adjudicados um ao outro mais. Aqui o compilador detetará o uso de membro privado fora da classe e notificará um erro de compilação.

- A solução acima ainda permitirá ao construtor da classe usar a função membro dentro da classe. Se isto também for indesejável, tais funções não devem ser implementadas. O compilador não é capaz de evitar o uso (acidental) desses membros proibidos, mas o gerador de ligações (linker) não será capaz de resolver as associações de referências externas.
- Nem sempre é uma boa idéia omitir funções membros que não podem ser chamadas da interface de classe. Em particular o operador de adjudicação sobrecarregado tem uma implantação padrão que será usada se não for nomeada uma versão sobrecarregada na interface de classe. Assim, com o operador de adjudicação sobrecarregado a comportamento mencionado deve ser seguido. Movendo certos construtores para a região privada da interface de classe também é uma boa técnica para evitar o uso deles pelo 'público em geral'.

7.6: Conclusões

Duas extensões importantes à classe foram discutidas neste capítulo: O operador de adjudicação sobrecarregado e o construtor de cópias. Como vimos, classes com ponteiros a membros de dados, endereçando memória local, são fontes potenciais de evasão de memória. As duas extensões introduzidas neste capítulo representam os meios padrão de evitar estas perdas de memória.

A conclusão simples é, portanto: Classes cujos objetos alocam memória usada por eles mesmos deve-se implantar um destrutor, um operador de adjudicação sobrecarregado e um construtor de cópias também.

Capítulo 8: Exceções

A linguagem C suporta diversos meios através dos quais um programa pode reagir a situações que quebram o fluxo normal do programa:

- A função pode noticiar a anormalidade e mostrar uma mensagem. Esta é talvez amais desastrosa reação que um programa possa mostrar.
- A função que detetou a anormalidade pode decidir a parar sua tarefa, retornando um código de erro a quem chamou. Este é um grande exemplo de pós-por decisões: Agora a função que chamou encara um grande problema. Evidentemente a função que chamou atuará à semelhança, passando o código do erro à função acima.
- A função pode decidir que as coisas estão saindo de controle e chamar `exit()` para terminar o programa completamente. Uma brava maneira de enfrentar um problema....
- A função pode usar uma combinação das funções `setjmp()` e `longjmp()` para forçar uma saída não local. Este mecanismo implanta uma espécie de salto goto, permitindo ao programa continue a um nível exterior, saltando níveis intermediários que teriam que ser visitados se a série de retornos das funções aninhadas tivessem sido usadas.

Na linguagem C++ todas as formas acima para manipular situações de quebra do fluxo continuam válidas. Contudo, das alternativas mencionadas, `setjmp()` e `longjmp()` não são vistas freqüentemente em programas C++ (ou mesmo em C), devido que o fluxo do programa é completamente destruído.

A linguagem C++ oferece as exceções como uma alternativa preferível a `setjmp()` e `longjmp()`. As exceções permitem aos programas C++ configurar um retorno controlado não localizado, sem as desvantagens de `setjmp()` e `longjmp()`.

As exceções representam um meio próprio de se livrar de situações que não podem ser manipuladas com facilidade por uma função, mas que não é suficientemente desastrosa para fazer o programa terminar completamente. Também, as exceções, oferecem uma camada de controle entre um pequeno retorno e um `crú exit()`.

Neste capítulo as exceções e sua sintaxe serão introduzidas. Primeiro um exemplo da diferença que o impacto das exceções têm sobre o programa e aquele de `setjmp()` e `longjmp()`.

8.1: Usando as exceções: elementos de sua sintaxe

Os seguintes elementos sintáticos são usados com as exceções:

- 'try': O bloco 'try' envolve instruções onde é possível a geração de exceções (o jargão fala em lançamento das exceções):

```
try
{
    // instruções onde as exceções podem ser lançadas
}
```

- 'throw': seguida por uma expressão de certo tipo lança o valor da expressão como uma exceção. A instrução 'throw' deve ser executada com o bloco 'try' em alguma parte: diretamente ou de uma função chamada direta ou indiretamente desde o bloco 'try'. Exemplo:

```
throw "Este gera uma char * exception";
```

- 'catch': Imediatamente em seguida ao bloco 'try', o bloco 'catch' recebe o lançamento das exceções. Exemplo de um bloco 'catch' que recebe char * exception:

```
catch (char *message)
{
    // instrução onde o char * exception lançado é manipulado
}
```

8.2: Um exemplo usando exceções

Nas próximas duas seções o mesmo programa básico será usado. O programa usa duas classes, Exterior e Interior. Um objeto Exterior é criado em main() e seu membro exterior::fun() é chamado. Então em Exterior::fun() um objeto Interior é construído. Depois de construído o objeto Interior, seu membro Interior::fun() é chamado.

A função Exterior::fun() termina e o destrutor do objeto Interior é chamado. O programa termina e o destrutor do objeto Exterior é chamado. Eis o programa básico:

```
#include <iostream>
using namespace std;

class Interior
{
public:
```

```

        Interior();
        ~Interior();
        void fun();
};

class Exterior
{
    public:
        Exterior();
        ~Exterior();
        void fun();
};

Interior::Interior()
{
    cout << "Construtor de Interior\n";
}

Interior::~~Interior()
{
    cout << "Destrutor de Interior\n";
}

void Interior::fun()
{
    cout << "fun de Interior\n";
}

Exterior::Exterior()
{
    cout << "Construtor de Exterior\n";
}

Exterior::~~Exterior()
{
    cout << "Destrutor de Exterior\n";
}

void Exterior::fun()
{
    Interior in;

    cout << "fun Exterior\n";
    in.fun();
}

int main()

```

```

{
    Exterior out;

    out.fun();
}

/*
    Saída Gerada:
    Construtor de Exterior
    Construtor de Interior
    fun de Exterior
    fun de Interior
    Destrutor de Interior
    Destrutor de Exterior
*/

```

Depois de compilar e executar a saída do programa é inteiramente a esperada e mostra exatamente o que queremos: Os destrutores são chamados em sua ordem correta, revertendo a seqüência das chamadas dos construtores.

Agora enfoquemos a atenção em duas variantes onde simulamos um evento desastroso não fatal em `Interior::fun()` que é supostamente manipulada em algum lugar no fim de `main()`. Consideraremos duas variantes. A primeira tentará manipular esta situação usando `setjmp()` e `longjmp()`; a segunda tentará manipular a situação com o mecanismo C++ de exceções.

8.2.1: Anacronismos: `'setjmp()'` e `'longjmp()'`

Para usar `setjmp()` e `longjmp()` o programa básico da seção 8.2 é ligeiramente modificado para conter a variável `'jmpbuf'`. A função `Interior::fun()` agora chama `longjmp()`, simulando um evento desastroso que será manipulado no fim de `main()`. Em `main()` vemos o código padrão que define o local alvo do salto longo usando a função `setjmp()`. O valor de retorno zero indica a iniciação da variável `jmp_buf` sobre a qual é chamada `Exterior::fun()`. Esta situação representa o 'fluxo normal'.

Para completar a simulação, o retorno do programa só é zero se for capaz de retornar de `Exterior::fun()` normalmente. Contudo, como sabemos, isto não acontecerá: `Interior::fun()` chama `longjmp()`, retornando ao `setjmp()` que (nesse momento) não retornará zero. Daqui, depois de chamar `Interior::fun()` de `Exterior::fun()` o programa passa por alto do comando 'if' na função `main()` e termina retornando o valor 1. Agora tente acompanhar esses passos estudando o seguinte programa fonte, modificação do programa básico dado na seção 8.2:


```

        #include <iostream>
#include <setjmp.h>
#include <cstdlib>

using namespace std;

class Interior
{
    public:
        Interior();
        ~Interior();
        void fun();
};

class Exterior
{
    public:
        Exterior();
        ~Exterior();
        void fun();
};

jmp_buf jmpBuf;

Interior::Interior()
{
    cout << "Construtor de Interior\n";
}

void Interior::fun()
{
    cout << "fun() de Interior\n";
    longjmp(jmpBuf, 0);
}

Interior::~~Interior()
{
    cout << "Destrutor de Interior\n";
}

Exterior::Exterior()
{
    cout << "Construtor de Exterior\n";
}

Exterior::~~Exterior()
{

```

```

        cout << "Destrutor de Exterior\n";
    }

void Exterior::fun()
{
    Interior in;

    cout << "fun() de Exterior\n";
    in.fun();
}

int main()
{
    Exterior out;

    if (!setjmp(jmpBuf))
    {
        out.fun();
        return 0;
    }
    return 1;
}

/*
    Saída Gerada:
    Construtor de Exterior
    Construtor de Interior
    fun() de Exterior
    fun() de Interior
    Destrutor de Exterior
*/

```

A saída produzida por este programa mostra claramente que o destrutor da classe Interior não foi executado. Isto é resultado direto da característica não local da chamada a `longjmp()`: o processamento continua imediatamente da chamada a `longjmp()` com a função membro `Interior::fun()` e a função `setjmp()` em `main()`. Ali retorna zero, portanto o programa com o retorno de 1. O que é importante aqui é que a chamada ao destrutor `Interior::~~Interior()`, esperando para ser executado no fim de `Exterior::fun()`, nunca chegou.

Este exemplo mostra que destrutores de objetos pode ser saltados facilmente quando usamos `longjmp()` e `setjmp()`, estas funções devem ser evitadas completamente em programas C++.

8.2.2: Exceções: A alternativa preferida

Na linguagem C++ as exceções são as melhores alternativas a `setjmp()` e `longjmp()`. Nesta seção apresentamos um exemplo que usa exceções. Outra vez, o programa deriva do programa básico dado na seção 8.2:

```
#include <iostream>
using namespace std;

class Interior
{
public:
    Interior();
    ~Interior();
    void fun();
};

class Exterior
{
public:
    Exterior();
    ~Exterior();
    void fun();
};

Inner::Inner()
{
    cout << "Inner constructor\n";
}

Inner::~~Inner()
{
    cout << "Inner destructor\n";
}

void Interior::fun()
{
    cout << "fun de Interior\n";
    throw 1;
    cout << "Este comando não é executado\n";
}

Exterior::Exterior()
{
    cout << "Construtor de Exterior\n";
}
```

```

Exterior::~Exterior()
{
    cout << "Destrutor de Exterior\n";
}

void Exterior::fun()
{
    Interior in;

    cout << "fun de Exterior\n";
    in.fun();
}

int main()
{
    Exterior out;

    try
    {
        out.fun();
    }
    catch (...)
    {}
}
/*
    Saída Gerada:
Construtor de Exterior
Construtor de Interior
fun de Exterior
fun de Interior
Destrutor de Interior
Destrutor de Exterior
*/

```

Neste programa a exceção é lançada ('throw') onde um `longjmp()` era usada no programa da seção 8.2.1. A construção comparável à chamada a `setjmp()` no programa é representado aqui pelos blocos 'try' e 'catch'. O bloco 'try' envolve comandos (incluindo chamadas a funções) em cujas exceções está 'throw', o bloco 'catch' pode conter instruções a serem executadas logo depois do lançamento de exceção.

Assim, em comparação com o exemplo na seção 8.2.1, a função `Interior::fun()` termina, se bem que com uma exceção no lugar de chamar `longjmp()`. A exceção é apanhada em `main()` e o programa termina. Quando inspecionamos a saída do programa notamos que o destrutor do objeto Interior, criado

em `Exterior::fun()` agora foi corretamente chamada. Notamos também que a execução da função `Interior::fun()` realmente terminou na instrução `'throw'`: a inserção do texto em `'cout'`, justo depois da instrução `'throw'`, não teve lugar.

Esperamos que isto tenha aumentado seu apetite por exceções, já que mostrou que:

- As exceções fornecem um meio de quebrar o controle normal do fluxo tendo que usar os retornos em cascata e sem necessidade de terminar o programa.
- As exceções não rompem a ativação dos destrutores e, portanto, são preferíveis no lugar de `setjmp()` e `longjmp()`.

8.3: Lançando exceções

As exceções são geradas no comando `'throw'`. A palavra chave `'throw'` é seguida de uma expressão que resulta num valor de certo tipo. Por exemplo:

```
throw "Olá Mundo";           // lança um char *
throw 18;                     // lança um inteiro
throw string("olá");          // lança uma string
```

Objetos definidos localmente em funções são automaticamente destruídos uma vez lançada uma exceção nessas funções e saindo dessas funções. Contudo, se o objeto é lançado, o receptor da exceção recebe uma cópia do objeto lançado. Esta cópia é construída justo antes do objeto ser destruído.

O próximo exemplo ilustra este ponto. Na função `Objeto::fun()` um objeto local `toThrow` é criado, que é lançado como uma exceção. A exceção é apanhada fora de `Objeto::fun()`, em `main()`. Neste ponto o objeto lançado já não existe mais, vejamos primeiro o texto fonte:

```
#include <iostream>
#include <string>
using namespace std;

class Objeto
{
    string d_nome;

public:
    Objeto(string nome)
    :
        d_nome(nome)
```

```

    {
        cout << "Construtor do Objeto " << d_nome << "\n";
    }
Objeto(Objeto const &outro)
:
    d_nome(outro.d_nome + " (copia)")
{
    cout << "Construtor de Cópia para " << d_nome << "\n";
}
~Objeto()
{
    cout << "Destrutor do Objeto " << d_nome << "\n";
}
void fun()
{
    Object toThrow("'objeto local'");

    cout << "Objeto fun() of " << d_nome << "\n";
    throw toThrow;
}
void ola()
{
    cout << "Olá por " << d_nome << "\n";
}
};

int main()
{
    Objeto out("'objeto main'");

    try
    {
        out.fun();
    }
    catch (Object o)
    {
        cout << "Recebeu a exceção\n";
        o.ola();
    }
}
/*

```

Saída Gerada:
 Construtor do Objeto 'objeto main'
 Construtor do Objeto 'objeto local'
 Objeto fun() de 'objeto main'
 Construtor de Cópia para 'objeto local' (copia)
 Destrutor do Objeto 'objeto local'

```

Construtor de Cópia para 'objeto local' (copia) (copia)
Recebeu a exceção
Olá por 'objeto local' (copia) (copia)
Destrutor do Objeto 'objeto local' (copia) (copia)
Destrutor do Objeto 'objeto local' (copia)
Destrutor do Objeto 'objeto main'
*/

```

A classe Objeto define diversos construtores simples e membros. O construtor de cópias é especial porque junta o texto “ (copia)” ao nome recebido, para permitir-nos monitorar a construção e destruição dos objetos de perto. A função membro Objeto::fun() gera a exceção e a lança com o objeto localmente definido. Antes do lançamento da exceção as seguintes saídas são geradas pelo programa:

```

Construtor do Objeto 'objeto main'
Construtor do Objeto 'objeto local'
Objeto fun() de 'objeto main'

```

Aqui a exceção é gerada, resultando na linha seguinte da saída:

Construtor de Cópia para 'objeto local' (copia)

A cláusula de lançamento recebe o objeto local e trata-o como um valor de argumento: cria uma cópia do objeto local. Em seguida a exceção é processada: o objeto local é destruído e 'catch' apanha um objeto, outra vez, como argumento. Outra cópia é criada. Assim vemos as linhas:

```

Destrutor do Objeto 'objeto local'
Construtor de Cópia para 'objeto local' (copia) (copia)

```

Agora estamos dentro de 'catch' que mostra a mensagem:

Recebeu a exceção

Seguida da chamada do membro ola() do objeto recebido. Este membro também mostra-nos que recebemos uma cópia da cópia do objeto local da função membro Objeto::fun():

```

Olá por 'objeto local' (copia) (copia)

```

Finalmente o programa termina e os objetos ainda vivos são, agora, destruídos na ordem inversa de sua criação:

```

Destrutor do Objeto 'objeto local' (copia) (copia)
Destrutor do Objeto 'objeto local' (copia)
Destrutor do Objeto 'objeto main'

```

Se 'catch' tivesse sido implantado para receber uma referência a um objeto que se fizesse usando 'catch (Objeto &o)' então se evitaria repetidas chamadas ao construtor de cópia. Nesse caso a saída do programa seria:

```

Construtor do Objeto 'objeto main'

```

```
Construtor do Objeto 'objeto local'
Objeto fun() de 'objeto main'
Construtor de Cópia para 'objeto local' (copia)
Destrutor do Objeto 'objeto local'
Recebeu a exceção
Olá por 'objeto local' (copia)
Destrutor do Objeto 'objeto local' (copia)
Destrutor do Objeto 'objeto main'
```

Isto mostra-nos que uma simples cópia do objeto local foi usada.

Claro que é má idéia lançar um ponteiro a um objeto local: o ponteiro é lançado, mas o objeto ao qual aponta morre uma vez lançada a exceção e o receptor terá em suas mãos um ponteiro selvagem. Más notícias....

Resumindo:

- Os objetos são lançados como cópia;
- Não se deve lançar ponteiros a objetos locais;
- Contudo ser possível lançar ponteiros ou referências a objetos gerados dinamicamente. Neste caso é indispensável tomar cuidado, pois o objeto gerado é devidamente eliminado quando a exceção gerada for apanhada, para prevenir fuga de memória.

As exceções são lançadas quando uma função não pode continuar sua tarefa normal, apesar de que o programa ainda é capaz de continuar. Imagine um programa que representa um calculador interativo. O programa continuamente pede expressões, que são avaliadas. Neste caso o analista (parser) de expressões pode encontrar um erro sintático; e a avaliação da expressão pode resultar em expressões impossíveis de avaliar, p.ex., porque a expressão resulta numa divisão por zero. Também o calculador permite o uso de variáveis: cheio de razões para lançar uma exceção, mas sem nenhuma razão irresistível para terminar o programa. No programa o seguinte código deve ser usado, lançando uma exceção:

```
if (!parse(expressionBuffer))           // falha no parser
    throw "Erro Sintático na expressão";

if (!lookup(variableName))              // variável não encontrada
    throw "Variável não definida";

if (divisionByZero())                   // incapaz de dividir
    throw "Divisão por zero não está definida";
```

O lugar destes lançamentos é imaterial: devem estar profundamente aninhados no programa ou num nível mais superficial. Ainda mais, as funções devem ser usadas para gerar a expressão que será

então lançada. Uma função:

```
char const *formatMessage(char const *fmt, ...);
```

Pode permitir-nos lançar mensagens mais específicas, como:

```
if (!lookup(variableName))  
    throw formatMessage("Variável '%s' não definida", variableName);
```

8.3.1: O comando 'throw' vazio

Podem ocorrer situações onde requer inspecionar um lançamento de exceção. Então, dependendo da natureza da exceção recebida, o programa pode continuar sua operação normal ou um evento sério teve lugar, requerendo uma reação mais drástica do programa. Numa situação servidor-cliente pode entrar requisições ao servidor numa fila. Toda requisição da fila é normalmente respondida pelo servidor, dizendo ao cliente que a requisição foi completada com sucesso ou que alguma sorte de erro ocorreu. Nesse momento o servidor pode ter morrido e o cliente deve ser capaz de descobrir essa calamidade, não ficando indefinidamente na espera da resposta do servidor.

Nesta situação é chamado imediatamente um manipulador de exceções. A exceção lançada é primeiro examinada em nível médio. Se for possível será processada ali. Se não for possível processá-la em nível médio, é passada inalterada a um nível mais superficial, onde, falando realmente, as exceções são processadas.

Pondo um comando 'throw' vazio no manipulador de exceções, as exceções recebidas são passadas ao próximo nível que poderia estar habilitado a processar aquele tipo de exceção.

Na situação de nosso servidor-cliente uma função:

```
initialExceptionHandler(char *exceção)
```

Estaria projetada para fazê-lo. A mensagem recebida é inspecionada. Se é uma mensagem simples, é processada, senão é passada a um nível mais externo. A implantação da `initialExceptionHandler()` mostra o comando 'throw' vazio:

```
void initialExceptionHandler(char *exceção)  
{  
    if (!plainMessage(exceção))  
        throw;  
  
    handleTheMessage(exceção);  
}
```

Como veremos abaixo (seção 8.5), o comando 'throw' vazio passa ao receptor da exceção, o

bloco 'catch'. Por isso, uma função como `initialExceptionHandler()` pode ser usada para uma variedade de exceções lançadas, enquanto o argumento usado na função `initialExceptionHandler()` seja compatível com o receptor da exceção.

Isto intriga? Então siga o exemplo seguinte, que salta claramente pelos tópicos cobertos no Capítulo 14. O exemplo seguinte pode ser saltado, digamos, sem perda da continuidade.

Podemos afirmar que uma manipulação básica da classe pode ser construída das exceções específicas de que derivam. Suponha que temos uma classe `Exceção`, contendo uma função membro que nos diz (um pouco milagroso!) a severidade da exceção lançada. Poderia ser `Mensagem`, `Alerta`, `Engano`, `Erro`, `Fatal`. Ainda mais, dependendo da severidade, uma exceção lançada pode conter mais ou menos informação, de alguma forma processada por uma função `processo()`. Além disso, todas as exceções possuem um texto pleno produzido pela função, p.ex., `toString()`, que diz-nos um pouco mais sobre a natureza da exceção gerada.

Usando o polimorfismo, `processo()` pode ser feita para se comportar diferenciadamente, dependendo da natureza da exceção lançada, quando chamada através de um ponteiro ou referência de `Exceção`.

Neste caso, um programa pode lançar qualquer dos tipos de exceção. Assumamos que `Mensagem` e `Atenção` podem ser processadas pela nossa função inicial `initialExceptionHandler()`. Então seu código seria:

```
void initialExceptionHandler(Exceção const *e)
{
    cout << e->toString() << endl;    // mostra a informação do texto
pleno

    if
    (
        e->severity() != ExceptionWarning
        &&
        e->severity() != ExceptionMessage
    )
        throw;                        // Passa a outro tipo de Exceção

        e->processo();                  // Processa uma mensagem ou uma
atenção
        delete e;
}
```

Devido ao polimorfismo (veja Capítulo 14), `e->processo()` ou processará uma `Mensagem` ou uma `Atenção`. As exceções lançadas são geradas como segue:

```
throw new Mensagem(<argumentos>);  
throw new Atenção(<argumentos>);  
throw new Engano(<argumentos>);  
throw new Erro(<argumentos>);  
throw new Fatal(<argumentos>);
```

Todas essas exceções são processáveis por `initialExceptionHandler()` que decide passar a exceção para cima ou terminar se processamento ela mesma. A classe polimórfica Exceção será desenvolvida mais tarde na seção 14.7.

8.4: O bloco 'try'

O bloco 'try' envolve comandos onde as exceções podem ser lançadas. Como vimos, a instrução 'throw' pode se localizar em qualquer parte, não necessariamente diretamente no bloco 'try'. Pode, p.ex., estar numa função, chamando dali o bloco 'try'.

A palavra chave 'try' é seguida por chaves atuando como um comando padrão C++: múltiplos comandos e definições podem ser postos aqui.

É possível (e muito comum) criar-se níveis de onde as exceções são lançadas. Por exemplo, o código de `main()` envolvido por um bloco 'try', são chamadas funções contendo blocos 'try', formando o nível seguinte onde as exceções podem ser geradas. Como vimos (na seção 8.3.1), as exceções lançadas em níveis internos de blocos 'try' podem ou não serem processadas nesse nível. Pondo-se um 'throw' vazio num manipulador de exceções, as exceções lançadas podem passar ao próximo nível (acima).

Se uma exceção é lançada fora de qualquer bloco 'try', então o modo padrão de manipular uma exceção é usado (sem recepção), que normalmente aborta o programa. Prove compilar e executar o seguinte programa e veja o que acontece:

```
int main()  
{  
    throw "Olá";  
}
```

8.5: Apanhando exceções

O bloco 'catch' contém o código que é executado quando uma exceção é lançada. Como as expressões são lançadas o bloco 'catch' deve saber que tipos de exceções pode manipular. Por isso, a palavra chave 'catch' é seguida de uma lista de parâmetros de pelo menos um parâmetro que é o tipo de exceção manipulado pelo bloco 'catch'. Assim, um manipulador para `char const *exceções` terá a seguinte

forma:

```
catch (char const *mensagem)
{
    // código para manipular a mensagem
}
```

Atrás (seção 8.3) vimos que tal mensagem não necessita ser lançada como uma string estática. É também possível, para uma função retornar uma string que é então lançada como uma exceção. Se tal função cria uma string que é lançada como uma exceção dinamicamente, o manipulador de exceções tem que remover a memória alocada para evitar perdas de memória.

Deve-se por muita atenção à natureza do parâmetro do manipulador de exceções, para assegurar que as exceções geradas dinamicamente sejam eliminadas uma vez que o manipulador as processou. Claro que quando uma exceção é passada a um manipulador de exceções um nível acima, a exceção recebida não deve ser eliminada pelo nível inferior.

Diferentes espécies de exceções podem ser lançadas: char *s, inteiros, ponteiros ou referências a objetos, etc.: todos esses diferentes tipos podem ser usados no lançamento e recepção de exceções. Assim, vários tipos de exceções podem sair de um bloco 'try'. Para poder apanhar todas as expressões que emergem de um bloco 'try' múltiplos manipuladores de exceções (i.e., blocos 'catch') podem seguir o bloco 'try'.

A ordem dos manipuladores de exceções é importante. Quando uma exceção é lançada, o primeiro manipulador de exceções é usado e os demais são ignorados. Assim, só um manipulador de exceções que seguinte ao bloco 'try' será executado. Consequentemente os manipuladores de exceção devem ser postos dos que têm os parâmetros mais específicos aos que possuem parâmetros mais genéricos. Por exemplo, se os manipuladores de exceções são definidos para char *s e void *s (i.e., qualquer ponteiro antigo) então o manipulador de exceções para o tipo formatado deve ser posto antes que o manipulador de exceções para o último tipo:

```
try
{
    // lança todas formas de ponteiros
}
catch (char const *mensagem)
{
    // processa ponteiros char lançados
}
catch (void *whatever)
{
    // processa todos os outros ponteiros lançados
}
```

```
}
```

Como alternativa de construção de diferentes tipos de manipuladores de exceções para diferentes tipos de exceções, é claro, é o projeto de uma classe específica onde os objetos contenham informações sobre a exceção. Este método foi mencionado antes, na seção 8.3.1. Este modelo requer só um manipulador, já que sabemos que não temos que lançar outro tipo de exceção:

```
try
{
    // código lançador só de Exceções ponteiros
}
catch (Exception *e)
{
    e->processa();
    delete e;
}
```

O comando 'delete e' no código acima indica que o objeto Exceção foi criado dinamicamente.

Quando o código de um manipulador de exceções localizado além do bloco 'try' foi processado a execução do programa continua mais adiante do último manipulador de exceções que segue esse bloco 'try' (a menos que o manipulador use 'return', 'throw' ou 'exit()' para sair da função prematuramente). Por isso distinguimos os seguintes casos:

- Se não foi lançada nenhuma exceção com o bloco 'try' nenhum manipulador de exceções é ativado e a execução continua do último comando do bloco 'try' para o primeiro comando além do último bloco 'catch'.
- Se foi lançada uma exceção no bloco 'try', mas nem o nível atual nem outro nível contém um manipulador de exceções apropriado, é chamado o manipulador padrão do programa, usualmente, abortando o programa.
- Se uma exceção é lançada do bloco 'try' e um manipulador de exceções apropriado está disponível, então o código desse manipulador de exceções é executado. Depois da execução do manipulador de exceções a execução do programa continua com o primeiro comando além do último bloco 'catch'.

Todos os comandos no bloco 'try' posteriores a um comando 'throw' executado serão ignorados. Contudo, os destrutores de objetos definidos localmente no bloco 'try' são chamados e antes de qualquer execução do código do manipulador de exceções.

A computação ou construção de uma exceção é realizada usando vários graus de sofisticação. Por exemplo, é possível usar o operador 'new', para usar funções membro estáticas de uma classe; retornar um ponteiro a um objeto; ou usar objetos de classes derivadas de uma classe, possivelmente envolvendo polimorfismo.

8.5.1: O Receptor Padrão

Nos casos onde se lancem diferentes tipos de exceções só um conjunto limitado de manipuladores pode ser requerido em certo nível do programa. As exceções dos tipos desse conjunto limitado são processadas, todas as demais são passadas a um nível exterior de manipulação de exceções.

Uma forma intermediária de manipulador de exceções pode ser implantada usando o manipulador de exceções padrão, que deve (devido à natureza hierárquica dos receptores de exceções, discutida na seção 8.5) se localizar além de todos os demais manipuladores mais específicos. Neste caso o nível corrente de manipulação de exceções deve fazer algum processamento padrão, mas, usando um 'throw' vazio (veja seção 8.3.1), passar a exceção lançada a um nível externo. Eis um exemplo que mostra o uso de um manipulador de exceções padrão:

```
#include <iostream>
using namespace std;

int main()
{
    try
    {
        try
        {
            throw 12.25;    // sem manipulador específico para double
        }
        catch (char const *mensagem)
        {
            cout << "Nível Interno: recebeu char const *\n";
        }
        catch (int value)
        {
            cout << "Nível Interno: recebeu int\n";
        }
        catch (...)
        {
            cout << "Nível Interno: manipulador genérico de
exceções\n";
            throw;
        }
    }
}
```

```

        }
    }
    catch(double d)
    {
        cout << "Nível Externo ainda conhece o double: " << d <<
endl;
    }
}
/*
    Saída Gerada:
Nível Interior: manipulador genérico de exceções
Nível Externo ainda conhece o double: 12.25
*/

```

Da saída gerada podemos concluir que um 'throw' vazio lança a exceção recebida ao próximo (externo) nível de receptores de exceções, conservando o tipo e o valor da exceção: assim, a manipulação de exceções básica ou genérica pode ser feita num nível interno, manipulação específica, baseada no tipo da expressão lançada, pode, então, continuar num nível externo.

8.6: Declarando Receptores de Exceções

Funções definidas em outra parte podem ser linkadas ao código que usa essas funções. Tais funções são normalmente declaradas nos arquivos cabeçalho, como funções independentes ou membros de uma classe.

Claro que essas funções podem lançar exceções. As declarações de tais funções contém uma lista de funções que lançam exceções ou uma lista de especificação de exceções, onde os tipos das exceções que podem ser lançadas são especificados. Por exemplo, uma função que pode lançar exceções 'char*' e 'int' vem declarado como:

```
void exceptionThrower() throw(char *, int);
```

Se especificado, uma lista da função aparece imediatamente além do cabeçalho da função (e além de uma possível especificação 'const') e nada nessa lista de lançamento deve estar vazio, ela tem a seguinte forma genérica:

```
throw([tipo1 [, tipo2, tipo3, ...]])
```

Se uma função não lança exceções pode-se usar uma lista de lançamento de exceções vazia para uma função que não lance exceções. P.ex.:

```
void noExceptions() throw ();
```

Em todos os casos o cabeçalho da função usado na definição da função deve coincidir com aquele usado em sua declaração, p.ex., incluindo possíveis listas de lançamento de exceções vazia.

Uma função para a que uma lista de lançamento é especificada não pode lançar outro tipo de exceções. Ocorre erro de execução se a função tenta lançar outro tipo de exceção além daqueles mencionados na lista de lançamento.

Por exemplo, considere as declarações e definições no seguinte programa:

```
#include <iostream>
using namespace std;

void charPintThrower() throw(char const *, int);    // declarações

class Thrower
{
public:
    void intThrower(int) const throw(int);
};

void Thrower::intThrower(int x) const throw(int)    // definições
{
    if (x)
        throw x;
}

void charPintThrower() throw(char const *, int)
{
    int x;

    cerr << "Entre um inteiro: ";
    cin >> x;

    Thrower().intThrower(x);
    throw "Este texto é lançado se entrou 0";
}

void runTimeError() throw(int)
{
    throw 12.5;
}

int main()
{
```



```

try
{
    charPintThrower();
}
catch (char const *mensagem)
{
    cerr << "Texto da Exceção: " << mensagem << endl;
}
catch (int value)
{
    cerr << "Exceção em Inteiro: " << value << endl;
}
try
{
    cerr << "Para erro de execução\n";
    runTimeError();
}
catch(...)
{
    cerr << "não alcançado\n";
}
}

```

Na função `charPintThower()` o comando de lançamento claramente lança um `'char const *'`. Contudo, como `intThrower()` pode lançar uma exceção `'int'`, a lista de lançamento de `charPintThrower()` também necessita conter `'int'`.

Se a lista de lançamento de exceções da função não é usada, a função pode lançar exceções (de qualquer tipo) ou não. Sem uma lista de lançamento da função a responsabilidade de fornecer os manipuladores corretos está nas mãos de quem projetou o programa.

8.7: *Iostreams e Exceções*

A biblioteca C++ de E/S era usada bem antes de se dispor das exceções em C++. Portanto, normalmente as classes da biblioteca `iostream` não lançam exceções. Contudo, é possível modificar o comportamento usando a função membro `ios::exceptions()`. Esta função tem duas versões sobrecarregadas:

- `iosstate exceptions()`: Este membro retorna o estado das flags através das quais stream lançará as exceções;
- `void exceptions(iosstate state)`: Este membro lançará uma exceção quando `'state'` for observado.

No contexto da biblioteca de E/S as exceções são objetos da classe `ios::failure`, derivada de `ios::exception`. Um objeto `failure` pode ser construído com uma `'string const &message'` que pode ser retirada usando o membro `'virtual char const *what()'`.

As exceções são usadas em situações excepcionais. Por isso pensamos que é questionável lançar um objeto `stream` numa exceção em situações estandartes como é o caso de um EOF. Usando exceções para manipular erros de entrada, poderia ser defensável, p.ex., quando um erro de entrada não ocorreu, mas se trata de um arquivo corrupto. no entanto aborta-se o programa quando com uma mensagem apropriada usualmente seria resolvida a questão. Eis um exemplo mostrando o uso de exceções num programa interativo que espera números:

```
#include <iostream>
using namespace::std;

int main()
{
    cin.exceptions(ios::failbit);

    while (true)
    {
        try
        {
            cout << "Entre um número: ";

            int value;

            cin >> value;
            cout << "Você entrou " << value << endl;
        }
        catch (ios::failure const &problem)
        {
            cout << problem.what() << endl;
            cin.clear();
            string s;
            getline(cin, s);
        }
    }
}
```

8.8: Exceções em construtores e destrutores

Só objetos construídos são eventualmente destruídos. Apesar de que isto pode soar como uma banalidade há uma sutileza aqui. Se a construção de um objeto falha, o destrutor de objetos não será

chamado, uma vez que o objeto está fora do escopo. Isto pode acontecer se uma exceção não recebida for gerada pelo construtor. Se a exceção for lançada depois da alocação ao objeto alguma memória, então seu destrutor (como não é chamado) não estará habilitado a eliminar o bloco de memória alocado. Isto resultará em perda de memória.

Os seguintes exemplos ilustram esta situação em forma de protótipo. O construtor da classe `Incompleta` primeiro mostra a mensagem e então lança uma exceção. Seu destrutor também mostra uma mensagem:

```
class Incompleta
{
public:
    Incompleta()
    {
        cerr << "Aloca alguma memória\n";
        throw 0;
    }
    ~Incompleta()
    {
        cerr << "Destroi a memória alocada\n";
    }
};
```

Em seguida `main()` cria um objeto dentro de um bloco `'try'`. Uma exceção gerada é apanhada na sequência:

```
int main()
{
    try
    {
        cerr << "Criando objeto `Incompleta'\n";
        Incompleta();
        cerr << "Objeto construído\n";
    }
    catch(...)
    {
        cerr << "Exceção apanhada\n";
    }
}
```

Quando este programa é executado produz a seguinte saída:

```
Criando objeto `Incompleta'
Alocada alguma memória
Exceção apanhada
```

Assim, se o construtor de Incompleta tenha alocado alguma memória, o programa sofrerá de fuga de memória. Para evitar este fato temos a contra-medida:

- As exceções não podem sair do construtor. Se parte do código do construtor pode gerar exceções, então esta parte deve estar envolta por um bloco 'try', a recepção da exceção deve ser no construtor. Existem, talvez, boas razões para lançar exceções fora do construtor, já que é um modo direto de informar o código que usa o construtor que o objeto não ficou disponível. Mas antes que a exceção deixe o construtor deve-se dar uma chance de liberar a memória já alocada. O seguinte esqueleto de um construtor mostra como isto pode ser realizado. Note que qualquer exceção que teria sido gerada é re-lançada, permitindo ao código externo inspecioná-la:

```
Incompleta::Incompleta()
{
    try
    {
        d_memória = new Type;
        código_que_pode_lançar_exceções();
    }
    catch (...)
    {
        delete d_memória;
        throw;
    }
};
```

- Ao iniciar membros podem ser geradas exceções. Nesses casos um bloco 'try' no corpo do construtor não pode apanhar tais exceções. Quando uma classe usa ponteiros como membros e gera exceções depois que esses ponteiros membros foram iniciados, pode-se evitar fuga de memória. Isto se faz usando ponteiros espertos, p.ex., objetos auto_ptr, vistos na seção 17.3. Já que para objetos auto_ptr seu destrutor é chamado mesmo depois de sua construção terminada e a construção de seus objetos componentes falhe. Neste caso a regra: uma vez construído um objeto seu destrutor é chamado quando o objeto sai do escopo ainda é aplicada.

A seção 17.3.6 cobre o uso de objetos auto_ptr para evitar fugas de memória quando são lançadas exceções fora do construtor, mesmo se a exceção é gerada na iniciação de um membro.

A linguagem C++ suporta um meio ainda mais genérico para evitar exceções depois de sair de funções (ou construtores): Funções blocos 'try'. Estas funções são discutidas na próxima seção.

Os destrutores têm problemas quando geram exceções. As exceções geradas ao sair dos destrutores, por certo, produzem fuga de memória, já que nem toda a memória alocada pode ter sido liberada quando a exceção é gerada. Outras formas de manipulação incompleta podem ser encontradas.

Por exemplo, uma classe `base_de_dados` pode armazenar modificações da base de dados em memória, deixando a atualização do arquivo da base de dados ao seu construtor. Se o destrutor gera uma exceção antes que o arquivo seja atualizado, então não haverá atualização. Mas existe outra consequência mais sutil, de exceções ao abandonar destrutores.

A situação que estamos por discutir pode ser comparada à de um carpinteiro que constroi um armário com uma gaveta. O armário está pronto e um cliente o compra, descobrindo que o armário pode ser usado como esperado. Satisfeito com o armário, o cliente pede ao carpinteiro para construir outro armário, esta vez com duas gavetas. Quando o segundo armário está pronto, o cliente o leva para casa e fica absolutamente assombrado quando o segundo armário imediata e completamente colapsa após seu primeiro uso.

História misteriosa? Considere o seguinte programa:

```
int main()
{
    try
    {
        cerr << "Criando Armário1\n";
        Armário1();
        cerr << "Além do objeto Armário1\n";
    }
    catch (...)
    {
        cerr << "Armário1 se comporta como esperado\n";
    }
    try
    {
        cerr << "Criando Armário2\n";
        Armário2();
        cerr << "Além do objeto Armário2\n";
    }
    catch (...)
    {
        cerr << "Armário2 se comporta como esperado\n";
    }
}
```

A execução deste programa produz a seguinte saída:

```
Criando Armário1
Gaveta 1 usada
Armário1 se comporta como esperado
Criando Armário2
Gaveta 2 usada
```

```
Gaveta 1 usada
Abortado
```

O abortado final indica que o programa foi abortado no lugar de mostrar uma mensagem como 'Armário2 se comporta como esperado'. Agora vejamos as três classes envolvidas. A classe Gaveta não tem características particulares, exceto que seu destrutor lança uma exceção:

```
class Gaveta
{
    size_t d_nr;
public:
    Gaveta(size_t nr)
        :
            d_nr(nr)
    {}
    ~Gaveta()
    {
        cerr << "Gaveta " << d_nr << " usada\n";
        throw 0;
    }
};
```

A classe Armário1 não tem características especiais. Só tem um objeto composto Gaveta:

```
class Armário1
{
    Gaveta esquerda;
public:
    Armário1()
        :
            esquerda(1)
    {}
};
```

A classe Armário2 é construída com semelhança, mas tem dois objetos compostos Gaveta:

```
class Armário2
{
    Gaveta esquerda;
    Gaveta direita;
public:
    Armário2()
        :
            esquerda(1),
            direita(2)
    {}
};
```

Quando o destrutor de Armário1 é chamado, o destrutor de Gaveta é eventualmente chamado para destruir seu objeto composto. Este destrutor lança uma exceção, que é apanhada além do primeiro bloco 'try' no programa. Este comportamento como o esperado. Contudo, o problema ocorre quando o destrutor de Armário2 é chamado. Dos dois objetos compostos, o destrutor do segundo objeto Gaveta é chamado antes. Este destrutor lança uma exceção, que é apanhada no programa além do segundo bloco 'try'. Apesar de que o fluxo de controle deixou o contexto do destrutor de Armário2, esse objeto não ficou completamente destruído ainda, já que o destrutor de seu outro objeto Gaveta (esquerda) ainda tem que ser chamado. Normalmente isto não seria grande problema: uma vez que a exceção saíra do destrutor de Armário2 foi lançada, qualquer ação seria simplesmente ignorada, se bem que (como os dois objetos Gaveta foram construídos bem) o construtor do esquerdo ainda pode ser chamado. Assim acontece aqui também. Contudo, o destrutor do objeto esquerdo também lança uma exceção. Como já deixamos o contexto do segundo bloco 'try', o controle de fluxo programado fica completamente misturado e o programa não tem outra opção que abortar. Isto é feito chamando terminate(), que em seu turno chama abort(). Aqui temos o colapso do armário com duas gavetas, mesmo que o armário com uma gaveta funcione perfeitamente.

O programa aborta uma vez que há múltiplos objetos compostos cujos destrutores lançam exceções que deixam os destrutores. Nesta situação um dos objetos compostos pode lançar uma exceção no momento em que o controle de fluxo do programa já deixou o contexto próprio. Isto causa o aborto do programa.

Esta situação pode ser evitada facilmente se asseguramos que as exceções jamais deixem os destrutores. No exemplo do armário, o destrutor de Gaveta lança uma exceção que deixa o destrutor. Isto não devia acontecer: A exceção devia ser apanhada pelo próprio destrutor. Quando as exceções nunca são lançadas para fora dos destrutores, nunca seremos capazes de apanhar exceções geradas por destrutores num nível exterior. Isto não é uma consequência séria já que vemos os destrutores como realizando serviços a membros diretamente relacionados à destruição dos objetos, antes que num membro sobre o qual podemos basear qualquer controle de fluxo. Eis o esqueleto do destrutor cujo código pode lançar exceções:

```
Class::~~Class()
{
    try
    {
        pode_lançar_exceções();
    }
    catch (...)
    {}
}
```

8.9: Os blocos 'try' das Funções

As exceções podem ser geradas enquanto o construtor está iniciando seus membros. Como exceções geradas nessa situação deve ser apanhadas? pelo próprio construtor ou fora dele? A solução intuitiva, aninhando a construção do objeto num bloco 'try' aninhado não resolve a situação (já que a exceção deixou o construtor) e não é um modelo elegante, porque resulta de um nível adicional aninhado (e algo artificial).

O seguinte exemplo ilustra o uso de um bloco 'try' aninhado onde main() define um objeto da classe Base_de_Dados. Assumindo que o construtor de Base_de_Dados pode lançar uma exceção, não há modo de apanhar a exceção num 'bloco exterior' (i.e., chamando main()), como não temos um bloco mais externo nesta situação. conseqüentemente, devemos resolver com uma solução menos elegante, como a que segue:

```
int main(int argc, char **argv)
{
    try
    {
        Base_de_Dados db(argc, argv);    // pode lançar exceções
        ...                             // outro código de main()
    }
    catch(...)                          // e/ou outros manipuladores
    {
        ...
    }
}
```

Este modelo potencialmente pode produzir código muito complexo. Se são definidos múltiplos objetos no bloco 'try' ou chegamos a uma série complexa de manipuladores de exceções ou fazemos blocos aninhados de blocos 'try' cada um usando seu próprio conjunto de manipuladores de recepção.

Nenhum destes modelos, contudo, resolve o problema básico: Como fazer com que as exceções geradas num contexto sejam apanhadas antes que o contexto desapareça?

O contexto local de uma função permanece acessível quando é definido como função do bloco 'try'. Uma função do bloco 'try' consiste de um bloco 'try' e seus manipuladores associados, definindo o corpo da função. Quando se usa uma função bloco 'try' a própria função pode receber qualquer exceção que seu código gera, mesmo mesmo que essas exceções sejam geradas nas listas de iniciação de membros em construtores.

O seguinte exemplo mostra uma função bloco 'try' disposta na função main() acima. Note

como o bloco 'try' e seus manipuladores agora substituem o corpo todo da função:

```
int main(int argc, char **argv)
try
{
    Base_de_Dados db(argc, argv);    // pode lançar exceções
    ...                             // o resto do código de main()
}
catch(...)                          // e/ou outros manipuladores
{
    ...
}
```

Claro que isto ainda não nos habilita ter exceções lançadas e recebidas pelo construtor de Base_de_Dados. A função bloco 'try', contudo, pode ser usada na implantação dos construtores. Nesse caso, as exceções lançadas pelos iniciadores básicos da classe (veja Capítulo 13) ou membros iniciadores também podem ser apanhadas pelos manipuladores de exceções do construtor. O seguinte exemplo mostra como uma função bloco 'try' deve ser usada ao implantarmos um construtor. Em particular, note que a palavra chave 'try' precede a lista de iniciação de membros (os dois pontos):

```
#include <iostream>

class Throw
{
public:
    Throw(int value)
    try
    {
        throw value;
    }
    catch(...)
    {
        std::cout << "Lança uma exceção manipulada localmente por
Throw()\n";
        throw;
    }
};

class Composer
{
    Throw d_t;
public:
    Composer()
    try                                // NOTE: 'try' precede a lista de iniciação
    :
        d_t(5)
    {}
}
```

```

        catch(...)
        {
            std::cout << "Composer() apanha a exceção também\n";
        }
    };

    int main()
    {
        Composer c;
    }
    /*
        Saída Gerada:

        Lança uma exceção manipulada localmente por Throw()
        Composer() apanha a exceção também
    */

```

Neste exemplo a exceção lançada pelo objeto `Throw` primeiro é apanhada pelo próprio objeto. Então é relançada. Como o construtor de `Composer` usa uma função bloco `'try'` o relançamento da exceção pelo objeto `Throw` também é apanhada pelo manipulador de exceções de `Composer`, a exceção foi gerada dentro de sua lista de iniciação de membros.

Uma nota final: Se um construtor ou função que use uma função bloco `'try'` também declara os tipos de exceções que pode lançar, então a função bloco `'try'` precisa seguir a lista de exceções especificada.

8.10: Exceções Estandartes

Todos os tipos de dados podem lançar exceções. Apesar de que as exceções estandartes derivam da classe `'exception'`. A derivação de classe é vista no Capítulo 13, mas os conceitos que jazem atrás de herança não são requeridos na presente seção.

Todas as exceções estandartes (e todas as classes definidas pelo usuário derivadas de `std::exception`) oferecem o membro:

```
char const *what() const;
```

Descrevendo numa curta mensagem textual a estrutura da exceção.

São oferecidas quatro classes derivadas de `std::exception` pela linguagem:

- `str::bad_alloc`: Lançada quando o operador `'new'` falha;

- `std::bad_exception`: Lançada quando uma função intenta gerar outro tipo de exceção além das declaradas na lista de lançamento da função;
- `std::bad_cast`: Lançada no contexto de polimorfismo (veja seção 14.5.1);
- `std::bad_typeid`: Também lançada no contexto de polimorfismo(veja seção 14.5.1).

Capítulo 9: Mais Sobrecarga a Operadores

No Capítulo 7 cobrimos a sobrecarga de operadores e mostramos muitos exemplos da sobrecarga de operadores também (i.e., 5), daremos, agora, uma olhada a diversos outros exemplos interessantes de sobrecarga de operadores.

9.1: Sobrecarga do `operator[]()`

Como próximo exemplo de sobrecarga de um operador apresentamos uma classe que opera sobre um conjunto de inteiros. Indexando os elementos do conjunto com o operador de conjuntos [], mas adicionalmente a classe examina se ocorreu sobre-passagem das fronteiras. Ainda mais, o índice do operador (`operator[]()`) é interessante em dois aspectos, produz um valor e aceita um valor quando usado respectivamente como um valor à direita (rvalue) e um valor à esquerda (lvalue) nas expressões. Eis um exemplo usando a classe:

```
int main()
{
    IntArray x(20);                // 20 inteiros

    for (int i = 0; i < 20; i++)
        x[i] = i * 2;             // adjudica valores aos elementos

    for (int i = 0; i <= 20; i++) // produz overflow das fronteiras
        cout << "No índice " << i << ": o valor é: " << x[i] << endl;
}
```

Primeiro, o construtor é usado para criar um objeto que contém 20 inteiros. Os elementos armazenados no objeto podem ser adjudicados ou retirados: o primeiro laço 'for' adjudica os valores aos elementos usando o operador de índice; o segundo laço 'for' retira os valores, mas produzirá, na execução, um erro, já que endereça o valor não existente `x[20]`. A interface da classe `IntArray` é:

```
class IntArray
{
    int      *d_data;
    unsigned d_size;

public:
    IntArray(unsigned size = 1);
    IntArray(IntArray const &outro);
}
```

```

~IntArray();
IntArray const &operator=(IntArray const &outro);

// operador de índice sobrecarregado
operators:
    int &operator[](unsigned index);           // primeiro
    int const &operator[](unsigned index) const; // segundo
private:
    void boundary(unsigned index) const;
    void copy(IntArray const &outro);
    int &operatorIndex(unsigned index) const;
};

```

Esta classe tem as seguintes características:

- Um de seus construtores tem um parâmetro sem sinal com valor de argumento padrão, que especifica o número de elementos do objeto.
- A classe usa internamente um ponteiro para alcançar a memória alocada. Daí que as ferramentas necessárias são fornecidas: um construtor de cópias, um operador sobrecarregado de adjudicação e um destrutor.
- Note que há dois operadores de indexação sobrecarregados. Porque dois deles?

O primeiro operador de indexação sobrecarregado permite-nos alcançar e modificar os elementos de objetos IntArray não constantes. Este operador sobrecarregado tem como seu protótipo uma função que retorna uma referência a inteiro. Isto nos permite usar expressões como `x[10]` como 'lvalues' ou 'rvalues'.

Assim podemos usar a mesma função para adjudicar e retirar valores. Além disso note que o valor retornado pelo operador sobrecarregado não é um 'int const', mas sim um 'int &'. Nesta situação não usamos 'const', já que devemos estar habilitados a modificar o elemento acessado quando o operador é usado como um valor à esquerda.

Contudo todo este esquema falha se não há nada a adjudicar. Considere a situação onde tenhamos um 'IntArray const stable(5)'. Tal objeto é um objeto constante, que não pode ser modificado. O compilador deteta isto e recusará compilar esta definição de objeto se só o primeiro operador de indexação estiver disponível. Daí a necessidade do segundo operador sobrecarregado de indexação. Aqui o valor retornado é um 'const &' antes que um 'int &' e a função membro é uma função membro constante. Esta segunda forma do operador sobrecarregado de indexação não é usada com objetos constantes. É usado para retirar valores, não para adjudicar valores, mas isto é exatamente o que precisamos ao usar objetos constantes. Aqui os membros são sobrecarregados só em seu atributo

constante. Esta forma de sobrecarga de funções foi introduzida anteriormente nas Anotações (seção 2.5.11 e 6.2).

Note também que desde que os valores guardados em `IntArray` são valores primitivos do tipo `'int'` está bem retornar tipo de valor. Mas com os objetos não se quer, usualmente, uma cópia extra que implica no retorno de tipos. Nestes casos retornar valores `'const &'` é preferível para funções membro constantes. Assim, na classe `IntArray` um retorno de `'int'` também poderia ter sido usado. O segundo operador sobrecarregado de indexação, então, usaria o seguinte protótipo:

```
int IntArray::operator[](int index) const;
```

- Como há um só membro de dados apontador, o destrutor da memória alocada do objeto é um simples `'delete data'`. Dessa forma o destrutor estandarte a função `'destroy()'` não foi usada.
- Como os elementos de dados são inteiros, não é necessário `'delete[]'`.

Agora a implantação dos membros fica:

```
#include "intarray.ih"

IntArray::IntArray(unsigned size)
:
    d_size(size)
{
    if (d_size < 1)
    {
        cerr << "IntArray: O tamanho do conjunto precisa ser >= 1\n";
        exit(1);
    }
    d_data = new int [d_size];
}

IntArray::IntArray(IntArray const &outro)
{
    copy(outro);
}

IntArray::~IntArray()
{
    delete d_data;
}

IntArray const &IntArray::operator=(IntArray const &outro)
{
    if (this != &outro)
```

```

    {
        delete d_data;
        copy(outro);
    }
    return *this;
}

void IntArray::copy(IntArray const &outro)
{
    d_size = other.d_size;
    d_data = new int [d_size];
    memcpy(d_data, other.d_data, d_size * sizeof(int));
}

int &IntArray::operatorIndex(size_t index) const
{
    boundary(index);
    return d_data[index];
}

int &IntArray::operator[](size_t index)
{
    return operatorIndex(index);
}

int const &IntArray::operator[](size_t index) const
{
    return operatorIndex(index);
}

void IntArray::boundary(size_t index) const
{
    if (index >= d_size)
    {
        cerr << "IntArray: Overflow de limite, index = " <<
            index << ", pode ser de 0 a " << d_size - 1 << endl;
        exit(1);
    }
}

```

Note especialmente a implantação da função 'operator[]()': como membros não constantes podem chamar funções membros constantes e como a implantação da função membro constante é idêntica à não constante podemos implantar ambos operadores inline usando uma função auxiliar 'int &operatorIndex(size_t index) const'. É interessante notar que as funções membro constantes podem retornar referências não constantes (ou ponteiros) referentes a um dos membros de dados de seu objeto. Este é um fato perigoso escondido do encobrimento de dados. Contudo, como os membros na interface

pública evitam esta brecha, sentimo-nos confiantes em defender 'int &operatorIndex() const' como função privada, sabendo que não será usada para este propósito indesejado.

9.2: Sobrecarga dos Operadores Inserção e Extração

Esta seção descreve como adotar que uma classe, de tal forma que pode ser usada com as streams de C++ cout, cerr e o operador inserção (<<). Adaptando uma classe dessa maneira, o operador de istream extração (>>) pode ser usado, é implantado similarmente e é simples, mostrado num exemplo.

A implantação de um 'operator<<()' sobrecarregado no contexto de cout ou cerr envolve suas classes, que é ostream. Esta classe está declarada no arquivo cabeçalho iostream e define só funções operadoras sobrecarregadas para os tipos 'básicos', tais como int, char *, etc.. O propósito desta seção é mostrar como um operador inserção pode ser sobrecarregado para um objeto de qualquer classe, digamos Pessoa (veja o Capítulo 7), possa ser inserido numa ostream. Depois de dispor de dito operador sobrecarregado, o seguinte será possível:

```
Pessoa kr("Kernighan and Ritchie", "desconhecido", "desconhecido");
```

```
cout << "Nome, endereço e número telefônico da Pessoa kr:\n" << kr << endl;
```

O comando 'cout << kr' envolve a 'operator<<()'. Esta função membro tem dois operandos: um ostream '&' e um Pessoa '&'. O propósito está definido num operador global sobrecarregado 'operator<<()' que espera dois argumentos:

```
// assume-se que está declarado em `pessoa.h'
ostream &operator<<(ostream &, Pessoa const &);

// definido em algum arquivo fonte
ostream &operator<<(ostream &stream, Pessoa const &pers)
{
    return
        stream <<
            "Nome:      " << pers.nome() <<
            "Endereço:  " << pers.endereço() <<
            "Fone:     " << pers.fone();
}
```

Note as seguintes características do operador 'operator<<()':

- A função retorna uma referência a um objeto 'ostream', para habilitar o 'encadeamento' do operador inserção;

- Os dois operandos de 'operator<<()' agem como argumentos da função sobrecarregada. No exemplo anterior, o parâmetro 'stream' é iniciado por 'cout', o parâmetro 'pess' é iniciado por 'kr'.

Para sobrecarregar o operador extração, p.ex., para a classe Pessoa, é necessário que os membros modifiquem dados privados. Tais modificadores, normalmente, são incluídos na interface de classe. Na classe 'Pessoa' dever-se-ia agregar os seguintes membros:

```
void setNome(char const *nome);
void setEndereço(char const *endereço);
void setFone(char const *fone);
```

A implantação destes membros seria francamente: a memória apontada pelo membro de dados correspondente seria liberada e o membro de dados apontaria a uma cópia do texto apontada pelo parâmetro. P.ex.:

```
void Pessoa::setEndereço(char const *endereço)
{
    delete d_endereço;
    d_endereço = strdupnew(endereço);
}
```

Uma função mais elaborada poderia examinar se o novo 'endereço' é razoável. Esta elaboração, contudo, não é mostrada aqui. Em seu lugar daremos uma olhada no operador extração sobrecarregado final (>>). Uma implantação simples é:

```
istream &operator>>(istream &str, Pessoa &p)
{
    string nome;
    string endereço;
    string fone;

    if (str >> nome >> endereço >> fone) // extrai três strings
    {
        p.setNome(nome.c_str());
        p.setEndereço(endereço.c_str());
        p.setFone(fone.c_str());
    }
    return str;
}
```

Note a aproximação passo a passo seguida com o operador extração: primeiro a informação requerida é extraída, usando os operadores de extração disponíveis (como um extrator de strings, então, se bem sucedido, são usados modificadores de membros do objeto a ser extraído. Finalmente a stream é retornada como referência.

9.3: Conversão de Operadores

Uma classe pode ser construída ao redor de um tipo básico. P.ex., a classe 'string' foi construída ao redor do tipo 'char *'. Tal classe pode definir todo tipo de operações, como adjudicações. Dê uma olhada na seguinte interface de classe, projetada sobre a classe 'string':

```
class String
{
    char
        *d_string;
public:
    String();
    String(char const *arg);
    ~String();
    String(String const &outro);
    String const &operator=(String const &rvalue);
    String const &operator=(char const *rvalue);
};
```

Os objetos desta classe podem ser iniciados de um 'char const *' e também de uma 'String'. Existe um operador de adjudicação sobrecarregado, que permite a adjudicação de um objeto 'String' e de um 'char const *' (Note que a adjudicação de um 'char const *' também inclui o ponteiro null. Uma adjudicação como 'stringObjeto = 0' é perfeitamente normal).

Normalmente numa classe que está menos diretamente acoplada aos seus dados que esta classe 'String', terá uma função membro acessora, como 'char const *String::c_str() const'. Apesar de que a necessidade de usar este último membro não apela à nossa intuição quando um conjunto de objetos 'String' é definido por, P.ex., uma classe 'StringArray'. Se esta última classe disponibiliza o operador '[]' para acessar membros individuais 'String', teríamos a seguinte interface para 'StringArray':

```
class StringArray
{
    String *d_store;
    size_t d_n;

public:
    StringArray(size_t size);
    StringArray(StringArray const &outro);
    StringArray const &operator=(StringArray const &rvalue);
    ~StringArray();

    String &operator[](size_t index);
};
```

Usando o 'StringArray::operator[]', adjudicações entre os elementos 'String' podem, muito

simplesmente, serem feitas:

```
StringArray sa(10);  
  
sa[4] = sa[3]; // adjudicação de String a String
```

Também é possível adjudicar um 'char const *' a um elemento de sa:

```
sa[3] = "Olá Mundo";
```

Aqui se passa pelas seguintes fases:

- Primeiro, sa[3] é avaliada. Isto resulta numa referência a uma 'String'
- Em seguida a classe 'String' é inspecionada em busca de uma adjudicação sobrecarregada, que aceite 'char const *' como seu elemento à direita. Este operador é encontrado e o objeto 'String' 'sa[3]' recebe seu valor.

Agora tentaremos fazer de outra maneira: como acessar o 'char const *' em 'sa[3]'? Tentaremos seguindo o código:

```
char const  
*cp = sa[3];
```

Isto, contudo, não funciona: necessitamos um operador sobrecarregado de adjudicação para 'class char const *'. Infelizmente não existe tal classe e por isso não podemos construir esse operador (veja também seção 9.11). Ainda mais, um 'cast' não funciona: o compilador não sabe como fazer um 'cast' de 'String' a 'char const *'. Como seguir daqui?

A solução ingênua é recorrer à função membro acessora 'c_str()':

```
cp = sa[3].c_str();
```

Essa solução funcionará, mas aparenta tão desajeitada.... Uma solução muito melhor é usar o operador de conversão.

Um operador de conversão é uma espécie de operador sobrecarregado, mas usado para transformar o tipo do objeto a outro. Usando um operador de conversão um objeto 'String' pode ser interpretado como 'char const *', que, então, pode ser adjudicado a outro 'char const *'. Os operadores de conversão podem ser implantados para todo tipo de variáveis que necessitam ser convertidas.

No exemplo atual, a classe 'String' necessita um operador de conversão para 'char const *'. Nas interfaces de classe, a forma geral de um operador de conversão é:

```
operator <type>();
```

Em nossa classe 'String', seria:

```
operator char const *();
```

A implantação do operador de conversão é diretamente:

```
String::operator char const *()  
{  
    return d_string;  
}
```

Notas:

- Não há menção de um tipo de retorno. O operador de conversão retorna o tipo mencionado depois da palavra chave operador.
- Em algumas situações o compilador necessita uma mão para se sobrepor a ambigüidades de nossas intenções. Num comando como:

```
cout.form("%s", sa[3])
```

o compilador fica confuso: estamos passando uma 'String &' ou um 'char const *' para a função membro 'form()'? Para auxiliar o compilador entregamos um 'static_cast':

```
cout.form("%s", static_cast<char const *>(sa[3]));
```

Ficamos deslumbrados em saber que aconteceria se a um objeto, p.ex., aplicássemos um operador de conversão a 'string' fosse definido e inserido, p.ex., num objeto 'ostream', permitindo inserir objetos 'string'. Neste caso o compilador não buscará por um operador de conversão apropriado (como 'operator string()'), mas reportará erro: Por exemplo, o seguinte exemplo produz um erro de compilação:

```
#include <iostream>  
#include <string>  
using namespace std;  
  
class NoInsertion  
{  
public:  
    operator string() const;  
};  
  
int main()  
{  
    NoInsertion object;  
  
    cout << object << endl;
```

```
}
```

O problema é causado pelo fato de que o compilador é informado de uma inserção, aplicada a um objeto. Buscará uma versão apropriada de um operador de inserção. Como não encontra, reporta erro de compilação, no lugar de realizar uma inserção em dois passos: primeiro usando o operador de inserção 'string()', seguido pela inserção dessa 'string' no objeto 'ostream'.

Os operadores de inserção são usados quando o compilador não tem escolha: uma adjudicação de um objeto 'NoInsertion' a um objeto 'string' é uma tal situação. O problema de como inserir um objeto em, p.ex., uma 'ostream' é simplesmente resolvido definindo um operador de inserção sobrecarregado apropriado, antes que recorrendo a um operador de conversão.

Diversas considerações se aplicam aos operadores de conversão:

- Em geral, uma classe pode ter mais que um operador de conversão. Quando são definidos múltiplos operadores de conversão, imediatamente se introduz ambigüidades.
- Um operador de conversões é uma 'extensão natural' das facilidades do objeto. Por exemplo, as classes 'stream' têm definido um operador 'bool()', permitindo construções como 'if(cin)'.
- Um operador de conversão pode retornar um 'rvalue'. Para, não só reforçar o encobrimento de dados, mas também porque implantando um operador de conversão como 'lvalue' não funciona. O seguinte programa é um caso como o dito: o compilador não executará uma conversão em dois passos e tenta (em vão) encontrar um 'operator=(int)':

```
#include <iostream>

class Lvalue
{
    int d_value;

public:
    operator int&()
    {
        return d_value;
    }
};

int main()
{
    Lvalue lvalue;

    lvalue = 5;        // não compila: não existe 'lvalue::operator=(int)'
```

```
};
```

- Um operador de conversão deve ser definido como uma função membro constante se não se quer modificar seus objetos membros de dados.
- Os operadores de conversão que retornam objetos compostos podem retornar referências a tais objetos antes que os objetos. O objeto pleno poderia forçar o compilador a chamar o construtor do objeto no lugar da referência ao objeto. Por exemplo, no programa seguinte, o construtor de cópias de 'std::string' não é chamado. Seria chamado se o operador de conversão fosse declarado como 'operator string()':

```
#include <string>

class XString
{
    std::string d_s;

public:
    operator std::string const &() const
    {
        return d_s;
    }
};

int main()
{
    XString x;
    std::string s;

    s = x;
};
```

9.4: A palavra chave *'explicit'*

As conversões são feitas não só por operadores de conversão, mas também por construtores com um parâmetro (ou diversos parâmetros com argumentos com valores padrão depois do primeiro parâmetro).

Considere a classe Pessoa, introduzida no Capítulo 7. Esta classe tem o construtor:

```
Pessoa(char const *nome, char const *endereço, char const *fone)
```

A este construtor se poderia dar valores padrão aos argumentos:

```
Pessoa(char const *nome, char const *endereço = "<desconhecido>",
        char const *fone = "<desconhecido>");
```

Em muitas situações pode usar intencionalmente, possivelmente dando ao padrão <desconhecido> textos para o endereço e o telefone. Por exemplo:

```
Pessoa frank("Frank", "Room 113", "050 363 9281");
```

As funções também podem usar objetos 'Pessoa' como parâmetros, p.ex., o seguinte membro numa classe fictícia Dados_Pessoa poderia existir:

```
Dados_Pessoa &Dados_Pessoa::operator+=(Pessoa const &pessoa);
```

Agora, combinando os dois pedaços de código acima podemos fazer algo como:

```
Dados_Pessoa dbase;
```

```
dbase += frank;      // adiciona frank à base de dados
```

Tanto mais, tanto melhor. Contudo, como o construtor de 'Pessoa' também pode ser usado como operador de conversão, é, também, possível fazer:

```
dbase += "karel";
```

Aqui 'char const * text 'karel' é convertido a um objeto (anônimo) de 'Pessoa' usando o construtor acima mencionado de 'Pessoa': O segundo e terceiro parâmetros usam os valores padrão. Aqui tem lugar uma conversão implícita de 'char const *' a um objeto 'Pessoa', que pode não ser o que o programador teria em mente quando construiu a classe 'Pessoa'.

Outro exemplo, considere a situação onde uma classe representando um recipiente seja construída. Assumamos que a construção inicial dos objetos é complexa e consome muito tempo, mas expandir um objeto, como acomoda mais elementos, consome ainda mais tempo. Tal situação acontece quando uma tabela hash é inicialmente construída para conter n elementos: tudo vai bem enquanto a tabela não está repleta, mas quando a tabela precisa ser expandida, normalmente todos seus elementos precisam passar novamente pela função hash para o novo tamanho da tabela.

Tal classe poderia ser definida (parcialmente) como segue:

```
class HashTable
{
    size_t d_maxsize;

public:
    HashTable(size_t n);    // n: tamanho inicial da tabela
    size_t size();         // retorna o número atual de elementos
```

```

// adiciona nova chave e valor
void add(std::string const &key, std::string const &value);
};

```

Agora considere a seguinte implantação de 'add()':

```

void HashTable::add(string const &key, string const &value)
{
    if (size() > d_maxsize * 0.75) // tabela quase cheia
        *this = size() * 2;      // Oh!: não é o que queremos!

    // etc.
}

```

Na primeira do corpo de 'add()' o programador primeiro determina quanto está cheia a tabela: se está cheia mais de 75% então a intenção é dobrar o tamanho da tabela. Contudo a tabela falha completamente em cumprir seu propósito: o programador, acidentalmente adjudica um valor sem sinal, pretendendo dizer qual será o novo tamanho da tabela. Isto lhe resulta na seguinte surpresa indesejada:

- O compilador lhe informa que não há 'operator=(size_t newsize)' para 'HashTable'.
- Há, contudo, um construtor que aceita 'size_t' e o operador padrão sobrecarregado de adjudicação ainda está válido, esperando uma 'HashTable' como seu operando à direita.
- Assim, o 'rvalue' da adjudicação (uma 'HashTable') é obtido construindo (implicitamente) uma 'HashTable' (vazia) que possa acomodar 'size() * 2' elementos.
- A recém construída 'HashTable' vazia é adjudicada à atual, removendo todos os elementos até então alojados.

Se o uso implícito de um construtor não é apropriado ou perigoso, pode ser evitado usando-se o modificador 'explicit' com o construtor. Os construtores com o modificador explícito só podem ser usados para a construção explícita de objetos e já não como conversor implícito. Por exemplo, para evitar a conversão implícita do valor sem sinal da interface de classe da classe 'HashTable' pode-se declarar o construtor assim:

```
explicit HashTable(size_t n);
```

Agora o compilador apanhará o erro na compilação de 'HashTable::add()', produzindo a mensagem de erro como:

```

error: no match for 'operator=' in
      *this = (this->HashTable::size() () * 2) '

```


9.5: Sobrecarga dos operadores incrementar e decrementar

A sobrecarga dos operadores incrementar ('operator++()') e decrementar ('operator--()') cria um pequeno problema: Existem duas versões de cada operador, já que podem ser usados como operadores pós-fixos ou pré-fixos (p.ex., `x++` e `++x`).

Usado como operador pós-fixos, o valor do objeto é retornado como 'rvalue', que é uma expressão com valor fixo: a variável pós-incrementada desaparecerá de vista. Usado como operador pré-fixos a variável é incrementada e tem seu valor retornado como 'lvalue', assim pode ser alterado imediatamente depois. Estas características não são requeridas quando o operador é sobrecarregado, é altamente importante ter em conta estas características em qualquer sobrecarga dos operadores incrementar e decrementar.

Suponhamos que definimos um envoltório (wrapper) de classe em torno dos tipos de valores sem sinal. A classe pode ter a seguinte interface de classe (parcialmente mostrada):

```
class Unsigned
{
    size_t d_value;

public:
    Unsigned();
    Unsigned(size_t init);
    Unsigned &operator++();
}
```

Isto define o operador incrementar pré-fixos sobrecarregado. Um 'lvalue' é retornado, como podemos deduzir o tipo retornado é 'Unsigned &'.

A implantação da função acima poderia ser:

```
Unsigned &Unsigned::operator++()
{
    ++d_value;
    return *this;
}
```

Para definir o operador pós-fixos uma versão sobrecarregada do operador é definida, que espera um argumento inteiro. Isto pode ser considerado (kludge?? ou) uma consequência da aplicação da noção de sobrecarga de uma função. Qualquer que seja sua opinião disto, podemos concluir o seguinte:

- Os operadores sobrecarregados incrementar e decrementar sem parâmetros são operadores pré-fixos e retornam referência ao objeto.

- Os operadores sobrecarregados incrementar e decrementar com um parâmetro inteiro são operadores pós-fixos e retornam o valor do objeto no momento em que o operador foi chamado, como valor constante.

Para acrescentar um operador incrementar pós-fixo ao envoltório da classe 'Unsigned' adicione a seguinte linha à interface de classe:

```
Unsigned const operator++(int);
```

A implantação do operador incrementar pós-fixo ficará assim:

```
Unsigned const Unsigned::operator++(int)
{
    return d_value++;
}
```

A simplicidade da implantação é enganadora. Note que:

- O valor é usado com um incremento pós-fixo na expressão de retorno. Por isso o valor da expressão de retorno é o valor de 'value' antes de ser incrementado; o que está correto.
- O valor de retorno da função é um inteiro sem sinal. Este objeto anônimo é implicitamente iniciado no valor de 'value', assim um construtor é aqui chamado veladamente.
- Os objetos anônimos sempre são constantes, portanto, o valor de retorno do operador incrementar pós-fixo é um valor 'rvalue'.
- O parâmetro não é usado. É apenas parte da implantação para evitar ambigüidade nos operadores pré-fixos e pós-fixos.

Quando o objeto tem uma organização dos dados mais complexa deve-se preferir usar um construtor de cópias. Por exemplo, assumamos que queremos implantar o operador incrementar pós-fixo na classe 'Dados_Pessoa', mencionada na seção 9.4. É de se esperar que a classe 'Dados_Pessoa' mantenha uma organização de entrada de dados. Se a classe 'Dados_Pessoa' possui um ponteiro a 'Pessoa *atual' ao objeto selecionado, então o operador incrementar pós-fixo para a classe 'Dados_Pessoa' pode ser implantado como segue:

```
Dados_Pessoa Dados_Pessoa::operator++(int)
{
    Dados_Pessoa tmp(*this);

    incrementCurrent();    // incrementa 'current', em alguma forma.
    return tmp;
}
```

Algo concernente a esta implementação é que esta operação requer duas chamadas ao construtor de cópias: Primeiro para conservar o estado atual, então para copiar o objeto 'tmp' ao (anônimo) valor retornado. Em alguns casos esta chamada dupla do construtor de cópias pode ser evitada, definindo-se um construtor especializado. P.ex.:

```
Dados_Pessoa Dados_Pessoa::operator++(int)
{
    return Dados_Pessoa(*this, incrementCurrent());
}
```

Está sub-entendido que `incrementCurrent()` retorna a informação que permite ao construtor por o membro de dados no valor pré-incremento, ao mesmo tempo incrementar o objeto 'Dados_Pessoa'. O construtor acima ficaria:

- Inicia seus membros de dados, copiando os valores desse objeto.
- Adjudica o valor de retorno de seu segundo parâmetro, que pode ser, p.ex., um índice.

Ao mesmo tempo '`incrementCurrent()`' terá incrementado o objeto 'Dados_Pessoa' atual.

A regra geral é que a chamada dupla ao construtor de cópias pode ser evitada se um construtor especializado for definido, iniciando um objeto no valor pré-incremento. O objeto atual terá seus membros de dados, necessários, incrementados por uma função que retorna os valores passados como argumento ao construtor, informando o construtor o estado anterior ao incremento dos membros de dados envolvidos. O operador incrementar pós-fixado retorna, então o objeto (anônimo) construído e nenhum construtor de cópias é chamado.

Finalmente é de se notar que as chamadas aos operadores incrementar ou decrementar usando o nome de suas funções sobrecarregadas requerem um argumento inteiro para informar o compilador que queremos a função incrementar pós-fixado. P.ex.:

```
Dados_Pessoa p;

p = outro.operator++(); // incrementa `outro`, e adjudica a `p`
p = outro.operator++(0); // adjudica a `p`, e incrementa `outro`
```

9.6: Sobrecarga de operadores binários

Em várias classes a sobrecarga dos operadores binários (como `operator+()`) pode serem extensões bastante naturais da funcionalidade da classe. Por exemplo, a classe '`std::string`' possui várias formas do '`operator+()`' como a maioria dos recipientes abstratos, estudados no Capítulo 12.

A maioria dos operadores binários vêm em dois modelos: Operador binário pleno (como operador +) e a variante de adjudicação aritmética (como o operador +=). O operador binário pleno retorna valores constantes, os operadores de adjudicação aritmética retornam uma referência a valores não constantes do objeto ao qual a operação foi aplicada. Por exemplo, com objetos 'std::string' o seguinte código (anotado abaixo do exemplo) é usado:

```
std::string s1;
std::string s2;
std::string s3;

s1 = s2 += s3;           // 1
(s2 += s3) + " pós-fixos"; // 2
s1 = "pré-fixos " + s3;  // 3
"pré-fixos " + s3 + "pós-fixos"; // 4
("pré-fixos " + s3) += "pós-fixos"; // 5
```

- Em //1 o conteúdo de s3 é adicionado a s2. Em seguida s2 é retornado e seu novo conteúdo somado a s1. Note que '+' retorna s2.
- Em //2 o conteúdo de s3 também é somado a s2, mas como '+' retorna s2 é possível adicionar algo mais a s2
- Em //3 o operador '+' retorna um objeto 'std::string' contendo a concatenação do texto pré-fixos e o conteúdo de s3. A string retornada por '+' é adjudicada a s1.
- Em //4 o operador '+' é aplicado duas vezes. O efeito é

1. O primeiro '+' retorna um 'std::string' contendo a concatenação do texto pré-fixos e o conteúdo de s3.

2. O segundo operador '+' toma o retorno como seu valor esquerdo e retorna uma string contendo o texto concatenado dos operandos esquerdo e direito.

3. A string retornada pelo segundo operador '+' representa o valor da expressão.

- O comando //5 pode não compilar (mas o fará com o compilador Gnu versão 3.1.1). Pode não compilar porque o operador '+' retorna uma string constante, que, portanto, não pode ser modificada pelo operador '+=' seguinte.

Abaixo seguiremos consequentemente esta linha de raciocínio e comprovaremos que os operadores binários sobrecarregados sempre retornam valores constantes.

Agora considere o seguinte código onde uma classe 'Binário' suporta o operador 'operator+()':

```
class Binário
{
public:
    Binário();
    Binário(int value);
    Binário const operator+(Binário const &rvalue);
};

int main()
{
    Binário b1;
    Binário b2(5);

    b1 = b2 + 3;           // 1
    b1 = 3 + b2;           // 2
}
```

A compilação deste pequeno programa falha no comando //2, com a mensagem do compilador:

```
error: no match for 'operator+' in '3 + b2'
```

Porque o comando //1 compila corretamente enquanto o comando //2 não compila?

Para entendermos a noção de promoção é introduzida. Como vimos na seção 9.4, os construtores que requerem um simples argumento são ativados implicitamente quando um objeto aparentemente é iniciado por um argumento de tipo correspondente. Encontramos este fato repetidamente em objetos 'std::string', quando uma string ASCII-Z era usada para iniciar objetos 'std::string'.

Em situações onde uma função membro espera um 'const &' para um objeto de sua classe (como 'Binário::operator+', membro mencionado acima), o tipo do argumento agora usado pode ser também qualquer tipo de argumento usado num construtor de um só argumento dessa classe. Esta chamada implícita ao construtor para se obter um objeto de um tipo próprio é chamada promoção.

Assim, no comando //1, o operador '+' é chamado para o objeto b2. Este operador espera outro objeto 'Binário' como seu operando à direita. Contudo um inteiro é fornecido. Como existe um construtor 'Binário(int)', o valor inteiro é promovido primeiro a um objeto 'Binário'. Em seguida o objeto 'Binário' é passado como argumento ao membro 'operator+()'.

Note que no comando //2 não há promoções: Aqui o operador '+' é aplicado a um valor do

tipo inteiro, que não possui um 'construtor', 'função membro' ou 'promoção'.

Como então as promoções dos operandos à esquerda se realizam em comandos como “pré-fixado” + s3”? Desde que sejam aplicadas promoções a argumentos de funções, devemos nos assegurar de que ambos operandos de operadores binários sejam argumentos. Isto significa que os operadores binários são declarados como funções sem classe, também conhecidos como funções livres. Contudo pertencem, conceitualmente, à classe para a qual foram implantados e assim devem ser declarados no arquivo cabeçalho da classe. Rapidamente veremos sua implantação, mas esta é a primeira revisão que cobre a declaração da classe 'Binário', onde declara um operador '+' sobrecarregado como uma função livre:

```
class Binário
{
public:
    Binário();
    Binário(int value);
    Binário const operator+(Binário const &rvalue);
};
```

Binário const operator+(Binário const &esquerda, Binário const &direita);

Definindo operadores binários como funções livres as seguintes promoções são possíveis:

- Se o operando à esquerda é da classe intencionada, o argumento da direita será promovido sempre que possível;
- Se o operando à direita é do tipo da classe intencionada, o argumento da esquerda será promovido sempre que possível;
- Não haverá promoções quando nenhum dos operandos forem do tipo da classe intencionada;
- Uma ambigüidade ocorre quando forem possíveis promoções de diferentes classes para os dois operandos. Por exemplo:

```
class A;

class B
{
public:
    B(A const &);
};

class A
{
public:
```

```

        A();
        A(B const &);
};

A const operator+(A const &a, B const &b);
B const operator+(B const &b, A const &a);

int main()
{
    A a;

    a + a;
};

```

Aqui ambos operadores '+' sobrecarregados são possíveis na compilação do comando 'a + a'. A ambigüidade deve ser resolvida explicitamente promovendo um dos argumentos, p.ex., 'a + B(a)', assim permitirá ao compilador resolver a ambigüidade para o primeiro operador sobrecarregado '+'.

- passo seguinte é implantar o correspondente operador aritmético sobrecarregado. Como este operador sempre possui um operando à esquerda que é um objeto de sua própria classe é implantado como uma função membro verdadeira da classe. Ainda mais, o operador aritmético de adjudicação pode retornar uma referência ao objeto ao qual a operação aritmética se aplica, já que o objeto pode ser modificado no mesmo comando. P.ex., '(s2 += s3) + “ pós-fixado”'. Eis nossa segunda revisão da classe 'Binário', mostrando ambas declarações do operador binário pleno e do operador aritmético de adjudicação correspondente:

```

class Binário
{
public:
    Binário();
    Binário(int value);
    Binário const operator+(Binário const &rvalue);

    Binário &operator+=(Binário const &outro);
};

```

```

Binário const operator+(Binário const &esquerdo, Binário const &direito);

```

Finalmente dispondo do operador aritmético de adjudicação, a implantação plena do operador binário fica extremamente simples. Contém um simples comando de retorno, onde um objeto anônimo é construído ao qual o operador aritméticos de adjudicação é aplicado. O objeto anônimo é retornado pelo operador binário pleno como seu valor constante. Já que sua implantação consiste num único comando, seu uso imediato aumenta sua eficiência:

```

class Binário

```

```

{
    public:
        Binário();
        Binário(int value);
        Binário const operator+(Binário const &rvalue);

        Binário &operator+=(Binário const &outro);
};
Binário const operator+(Binário const &esquerdo, Binário const &direito)
{
    return Binário(esquerdo) += direito;
}

```

Podemos achar maravilhoso saber onde os valores temporários estão localizados. A maioria dos compiladores, nestes casos, aplica um procedimento chamado 'otimização do valor retornado': O objeto anônimo é criado no lugar onde o objeto eventualmente retornado será guardado. Assim, no lugar de primeiro criar um objeto temporário separado e então copiá-lo no lugar do valor de retorno, inicia o valor de retorno usando um argumento à esquerda e então aplica o operador '+' para agregar o argumento à direita a ele. Sem o retorno a otimização do valor retornado teria que:

- Criar um espaço separado para acomodar o valor de retorno;
- Iniciar um um objeto temporário, usando um valor à esquerda;
- Adicionar um valor à direita;
- Usar o construtor de cópias para copiar o objeto temporário sobre o valor de retorno.

A otimização do valor de retorno não é obrigatória, mas opcionalmente está disponível nos compiladores. Como não tem efeitos colaterais negativos a maioria dos compiladores usa-a.

9.7: Sobrecarga do 'operador new(size_t)'

Quando o operador 'new' é sobrecarregado precisa retornar o tipo 'void *' e pelo menos um argumento do tipo 'size_t'. O tipo 'size_t' está definido no arquivo cabeçalho 'cstddef', que, portanto, deve ser incluído quando o operador 'new' é sobrecarregado.

Também é possível definir versões múltiplas do operador 'new', desde que cada versão tenha seu próprio conjunto de argumentos. O operador 'new' global segue sendo utilizável, como '::operator'. Se uma classe X sobrecarrega o operador 'new' o operador fornecido pelo sistema é ativado por:

```
X *x = ::new X();
```


A sobrecarga 'new[]' é discutida na seção 9.9.

O seguinte exemplo mostra uma versão sobrecarregada do operador 'new':

```
#include <cstddef>

void *X::operator new(size_t sizeofX)
{
    void *p = new char[sizeofX];

    return memset(p, 0, sizeof(X));
}
```

Agora vejamos o que ocorre quando o operador 'new' é sobrecarregado na classe X. Assuma que essa classe é definida assim:

A seguir\ (Por simplicidade violamos o princípio de encapsulamento aqui. O princípio de encapsulamento, contudo, é imaterial para a discussão presente de como funciona o operador 'new'.):

```
class X
{
public:
    void *operator new(size_t sizeofX);

    int d_x;
    int d_y;
};
```

Agora considere o seguinte fragmento de programa:

```
#include "x.h" // interface da classe X
#include <iostream>
using namespace std;

int main()
{
    X *x = new X();

    cout << x->d_x << ", " << x->d_y << endl;
}
```

O pequeno programa produz a seguinte saída:

0, 0

Ao chamar 'new X()', nosso pequeno programa realizou as seguintes ações:

- Primeiro, o operador 'new' foi chamado o que alojou um bloco iniciado de memória, do

tamanho de um objeto X.

- Em seguida um apontador para este bloco de memória foi passado ao construtor padrão de X(). Como não foi definido um construtor, não se fez nada.

Devido à iniciação do bloco de memória pelo operador 'new' o objeto X já estava iniciado em zero quando o construtor foi chamado.

As funções membro não estáticas são passadas como um ponteiro (vazio) sobre o qual operam. Este ponteiro vazio se torna o ponteiro 'this' em funções membro não estáticas. Este procedimento também é seguido pelos construtores. Nos pedaços seguintes de pseudo-código C++ o ponteiro é visível. Na primeira parte um objeto 'X', 'x', é definido diretamente, na segunda parte o operador (sobrecarregado 'new' é usado:

```
X::X(&x); // o endereço de x é passado ao
// construtor
void *ptr = X::operator new(); // new aloja a memória

X::X(ptr); // em seguida o construtor opera sobre
a
// memória retornada pelo 'operator
new'
```

Note que no pseudo-fragmento C++ a função membro foi tratada como uma função membro estática da classe 'X'. Agora o operador 'new' é uma função estática de sua classe: Não pode alcançar os membros de dados de seus objetos, enquanto sua tarefa normal, como operador 'new', é criar espaço e iniciar a área requerida. Ela pode fazer isto alojando suficiente memória. Em seguida a memória é passada (como ponteiro 'this') ao construtor para processamentos posteriores. O fato de um operador 'new' sobrecarregado seja agora uma função estática, não requerendo um objeto de sua classe, pode ser ilustrado no seguinte fragmento de programa (desaprovado em situações normais):

```
int main()
{
    X x;

    X::operator new(sizeof x);
}
```

A chamada ao operador 'X::operator new()' retorna um 'void *' a um bloco de memória com o tamanho de um objeto 'X'.

O operador 'new' pode ter múltiplos parâmetros. O primeiro é iniciado com um argumento implícito e é sempre o parâmetro 'size_t', outros parâmetros são iniciados por argumentos explícitos, especificados quando o operador 'new' seja usado. Por exemplo:

```

class X
{
    public:
        void *operator new(size_t p1, size_t p2);
        void *operator new(size_t p1, char const *fmt, ...);
};

int main()
{
    X
        *p1 = new(12) X(),
        *p2 = new("%d %d", 12, 13) X(),
        *p3 = new("%d", 12) X();
}

```

O ponteiro 'p1' aponta a um objeto 'X' para o qual foi alojada memória pela chamada ao primeiro operador 'new' sobrecarregado, seguida pela chamada ao construtor 'X()' para esse bloco de memória. O ponteiro 'p2' aponta a um objeto 'X' para o qual foi alojada memória com uma chamada ao segundo operador 'new' sobrecarregado, novamente seguida pela chamada ao construtor 'X()' para este bloco de memória. Note que o ponteiro 'p3' também usa o segundo operador 'new' sobrecarregado, já que o operador aceita um número variável de argumentos, o primeiro dos quais é um 'char const *'.

Finalmente, note que, nenhum argumento explícito é passado como primeiro parâmetro de 'new', já que este parâmetro é implicitamente passado pela especificação do tipo requerido pelo operador 'new'.

9.8: Sobrecarga do `operador delete(void *)'

O operador 'delete' também pode ser sobrecarregado. O operador 'delete' precisa ter um argumento 'void *' e um argumento opcional do tipo 'size_t', que é o tamanho em bytes dos objetos da classe para a qual o operador 'delete' foi sobrecarregado. O tipo de retorno do operador 'delete' sobrecarregado é 'void'.

Assim, numa classe o operador 'delete' pode ser sobrecarregado usando o seguinte protótipo:

```
void operator delete(void *);
```

ou

```
void operator delete(void *, size_t);
```

A sobrecarga do operador 'delete[]' é discutida na seção 9.9.

O operador 'feito em casa' 'delete' é chamado depois de se ter executado o destrutor da classe associada. Portanto o comando:

```
delete ptr;
```

Sendo 'ptr' um apontador a um objeto da classe 'X', para o qual o operador 'delete' foi sobrecarregado, falha nos seguintes comandos:

```
X::~X(ptr);           // chama o destrutor

                        // e faz algo com a memória apontada por ptr
X::operator delete(ptr, sizeof(*ptr));
```

O operador 'delete' o que seja com a memória apontada por 'ptr'. Pode, p.ex., simplesmente eliminá-la. Se isto fosse o preferido a fazer, então o operador 'delete' padrão pode ser ativado usando o operador de resolução '::'. Por exemplo:

```
void X::operator delete(void *ptr)
{
    // qualquer operação considerada necessária, então:
    ::delete ptr;
}
```

9.9: Os Operadores 'new[]' e 'delete[]'

Nas seções 7.1.1, 7.1.2 e 7.2.1 introduzimos os operadores 'new[]' e 'delete[]'. Como os operadores 'new' e 'delete' os operadores 'new[]' e 'delete[]' podem ser sobrecarregados. Como é possível sobrecarregar esses operadores, bem como 'new' e 'delete', devemos ter cuidado ao eleger os operadores apropriados. A seguinte regra prática deve ser seguida:

Se 'new' é usado para alojar memória, 'delete' deve ser usado para desalojar a memória. Se 'new[]' é usado para alojar memória, 'delete[]' deve ser usado para desalojá-la.

A maneira padrão de comportamento desses operadores é a seguinte:

o O operador 'new' é usado para alojar um simples objeto ou valor primitivo. O construtor de objetos é chamado para um objeto. O operador 'delete' é usado para desalojar a memória reservada por 'new'. Outra vez, o destrutor de objetos da classe é chamado para um objeto. O operador 'new[]' é usado para alojar uma série de valores primitivos ou objetos. Note que se uma série de objetos é alojada, o construtor padrão é chamado para iniciar cada objeto individual. O operador 'delete[]' é usado para eliminar a memória previamente reservada por 'new[]'. Se foram alojados anteriormente objetos, então o

destrutor será chamado para cada objeto individual. Contudo, se foram alojados apontadores aos objetos, não se chama o destrutor, já que um apontador é considerado um tipo primitivo e certamente não um objeto.

Os operadores 'new[]' e 'delete[]' só podem ser sobrecarregados em classes. Conseqüentemente quando alojando tipos primitivos ou ponteiros a objetos somente a linha padrão da ação é seguida: Quando conjuntos de ponteiros a objetos são eliminados ocorre uma evasão de memória, amenos que os objetos apontados foram eliminados antes.

Nesta seção apresentamos a mera sintaxe dos operadores 'new[]' e 'delete[]'. Deixamos como exercício ao leitor fazer bom uso desses operadores sobrecarregados.

9.9.1: Sobrecarga de 'new[]'

Para sobrecarregar o operador 'new[]' para um objeto da classe 'Objeto' sua interface deve conter as seguintes linhas, mostrando as múltiplas formas de sobrecarregar o operador 'new[]':

```
class Objeto
{
    public:
        void *operator new[](size_t size);
        void *operator new[](size_t index, size_t extra);
};
```

A primeira forma mostra a forma básica do operador 'new[]'. Retorna 'void *' e define pelo menos um parâmetro 'size_t'. Quando o operador 'new[]' é chamado 'size' contém o número de bytes que deve ser alojado para o número de objetos. Estes objetos são iniciados pelo operador global 'new[]' usando a forma:

```
::new Objeto [size / sizeof(Objeto)]
```

Ou, alternativamente, a quantidade de memória (não iniciada) pode ser alojada usando:

```
::new achar [size]
```

Um exemplo de uma função membro operador sobrecarregado 'new[]', que retorna um conjunto de objetos Objetos todos preenchidos com bytes 0:

```
void *Objeto::operator new[](size_t size)
{
    return memset(new char[size], 0, size);
}
```

Construído o operador sobrecarregado 'new[]' ele será usado automaticamente em comandos

como:

```
Objeto *op = new Objeto[12];
```

O operador 'new[]' pode ser sobrecarregado usando parâmetros adicionais. A segunda maneira de sobrecarregar o operador 'new[]' mostra um parâmetro adicional sem sinal. A definição de tal função é padrão e pode ser:

```
void *Objeto::operator new[](size_t size, size_t extra)
{
    size_t n = size / sizeof(Objeto);
    Objeto *op = ::new Objeto[n];

    for (size_t idx = 0; idx < n; idx++)
        op[idx].value = extra;           // assume um membro `value'

    return op;
}
```

Para usar este operador sobrecarregado, somente o parâmetro adicional precisa ser fornecido. É dado numa lista de parâmetros junto ao nome do operador:

```
Objeto *op = new(100) Objeto[12];
```

Isto tem como resultado um conjunto de 12 objetos 'Objeto', todos com valor 100.

9.9.2: Sobrecarga de 'delete[]'

Como o operador 'new[]', o operador 'delete[]' pode ser sobrecarregado. Para sobrecarregar o operador 'delete[]' na classe 'Objeto' a interface deve conter múltiplas formas de sobrecarga do operador 'delete[]':

```
class Objeto
{
public:
    void operator delete[](void *p);
    void *operator delete[](void *p, size_t index);
    void *operator delete[](void *p, int extra, bool yes);
};
```

9.9.2.1: 'delete[](void *)'

A primeira forma mostra a forma básica do operador 'delete[]'. Seu parâmetro é iniciado com o endereço do bloco de memória previamente alojada por 'Objeto::new[]'. Estes objetos podem ser

eliminados pelo operador global 'delete[]'. Contudo, o compilador espera '::delete[]' para receber um ponteiro a 'Objeto', portanto é necessário um tipo de 'cast':

```
::delete[] reinterpret_cast<Object *>(p);
```

Um exemplo do operador 'delete[]' sobrecarregado é:

```
void Object::operator delete[](void *p)
{
    cout << "operador delete[] para Objetos chamados\n";
    ::delete [] reinterpret_cast<Object *>(p);
}
```

Construído o operador sobrecarregado 'delete[]', ele será usado automaticamente em comandos como:

```
delete[] new Object[5];
```

9.9.2.2: 'delete[](void *, size_t)'

O operador 'delete[]' pode ser sobrecarregado com o uso de parâmetros adicionais. Contudo, se sobrecarregado como:

```
void *operator delete[](void *p, size_t size);
```

Então 'size' é automaticamente iniciado no tamanho (em bytes) do bloco de memória para o qual '*p' aponta. Se esta forma for definida, então a primeira forma não pode ser definida, para evitar ambigüidade. Um exemplo desta forma do operador 'delete[]' é:

```
void Object::operator delete[](void *p, size_t size)
{
    cout << "eliminando " << size << " bytes\n";
    ::delete [] reinterpret_cast<Object *>(p);
}
```

9.9.2.3: Formas alternativas de sobrecarga do operador 'delete[]'

Se definimos parâmetros alternativos, como em:

```
void *operator delete[](void *p, int extra, bool yes);
```

Uma lista explícita de argumentos necessita ser fornecida. Com 'delete[]', a lista de argumentos é especificada entre parênteses:

```
delete[](new Object[5], 100, false);
```

9.10: Objetos Funções

Objetos funções são criados sobrecarregando a chamada ao operador 'operator()()'. Ao definirmos a chamada ao operador um objeto fica mascarado como se fosse uma função, daí o termo objetos funções.

Os objetos funções desempenham um importante papel em algoritmos genéricos e seu uso é preferível a alternativas como ponteiros a funções. O fato de serem importantes no contexto de algoritmos genéricos constitui uma sorte de dilema didático: neste ponto seria bom se houvésssemos visto os algoritmos genéricos, mas para a discussão dos algoritmos genéricos se requer o conhecimento de objetos funções. Este dilema é resolvido de modo bem conhecido: ignorando-se a dependência.

Objetos funções são objetos para os quais o operador 'operator()()' foi definido. Os objetos funções comumente são usados em combinação com algoritmos genéricos, mas também em situações onde se usaria apontadores para funções. Outra razão para o uso de objetos funções é dar suporte a funções 'inline', que não podem ser usadas em combinação com apontadores para funções.

Assuma que temos uma classe 'Pessoa' e um conjunto de objetos 'Pessoa'. Além disso, assumamos que o conjunto não está ordenado. Um procedimento bem conhecido para encontrar um objeto particular 'Pessoa' num conjunto é usar a função 'lsearch()', que executa uma busca linear no conjunto. Um fragmento de programa que usa esta função é:

```
Pessoa &alvo = PessoaAlvo();      // determina a pessoa a encontrar
Pessoa *pArray;
size_t n = fillPessoa(&pArray);

cout << "A pessoa alvo é";

if (!lsearch(&target, pArray, &n, sizeof(Person), compareFunction))
    cout << ": não ";

cout << "encontrada\n";
```

A função 'PessoaAlvo()' é chamada para se determinar a pessoa que buscamos e a função 'fillPessoa()' é chamada para preencher a 'pArray'. A função 'lsearch()' é usada para localizar a pessoa alvo.

A função de comparação deve estar disponível, já que é um dos argumentos de 'lsearch()'. Poderia ser algo como:


```

int compareFunction(Pessoa const *p1, Pessoa const *p2)
{
    return *p1 != *p2;          // lsearch() necessita 0 para objetos
iguais
}

```

Isto, claro, assume que o operador '!=()' foi sobrecarregado na classe Pessoa, já que é quase seguro que uma comparação entre bytes não será apropriada aqui. Sobrecarregar o operador '!=()' não é grande trabalho e assumimos que está disponível também.

Com lsearch() (e amigos, com parâmetros que são apontadores a funções) uma função 'compare() inline' não pode ser usada: já que o endereço da função 'compare()' tem que ser conhecido pela função 'lsearch()'. Assim, numa média de $n/2$ vezes pelo menos a ação terá lugar:

1. Os dois argumentos da função de combinação são puxados para a pilha;
2. O valor do parâmetro final de 'lsearch()' é determinado, obtendo o endereço de 'compareFunction()';
3. A função de comparação é chamada;
4. Então, dentro da função de comparação o endereço do argumento à direita de `Pessoa::operator!=()` é posto na pilha;
5. A função 'Pessoa::operator!=()' é avaliado;
6. O argumento de 'Pessoa::operator!=()' é retirado da pilha;
7. Os dois argumentos da função de comparação são retirados da pilha.

Quando usamos objetos funções emerge outro quadro. Assuma que construímos uma função 'BuscaPessoa()', tendo o seguinte protótipo (tendo presente que esta não é a solução preferível. Normalmente seria preferível um algoritmo genérico a uma função 'feita em casa'. Mas por agora a função é usada para ilustrar a implantação de objetos funções):

```

Pessoa const *BuscaPessoa(Pessoa *base, size_t nmemb,
                          Pessoa const &target);

```

Esta função pode ser usada da seguinte maneira:

```

Pessoa &target = PessoaAlvo();
Pessoa *pArray;
size_t n = fillPerson(&pArray);

```

```

cout << "A pessoa alvo é";

if (!BuscaPessoa(pArray, n, target))
    cout << ": não";

cout << "encontrada\n";

```

No demais pouco mudou. Substituímos a chamada a 'lsearch()' pela chamada a outra função: **BuscaPessoa()**. Agora veremos o que acontece dentro de 'BuscaPessoa()':

```

Pessoa const *BuscaPessoa(Pessoa *base, size_t nmemb,
                          Pessoa const &target)
{
    for (int idx = 0; idx < nmemb; ++idx)
        if (target(base[idx]))
            return base + idx;
    return 0;
}

```

A implantação mostra uma busca linear plena. Contudo, no laço 'for' a expressão **target(base[idx])** mostra o uso de nosso objeto alvo como um objeto função. Sua implantação é simples:

```

bool Pessoa::operator() (Pessoa const &outro) const
{
    return *this != outro;
}

```

Note a sintaxe algo peculiar: 'operator()()'. O primeiro conjunto de parâmetros define o operador em particular que é sobrecarregado: a função chamada operador. O segundo parênteses define os parâmetros requeridos pela função. O 'operator()()' aparece no arquivo cabeçalho da classe como:

```

bool operator() (Pessoa const &outro) const;

```

Agora, 'Pessoa::operator()()' é uma simples função. Contém só um comando, assim, podemos considerar fazê-la 'inline'. Assumindo que o fazemos, então isto é o que acontece quando a chamamos:

- O endereço do argumento da direita de 'Pessoa::operator()!==(())' é levado à pilha;
- A função 'operator!==(())' é avaliada;
- O argumento da função 'operator!==(())' é retirado da pilha;

Note que devido ao fato do 'operator()()' ser uma função 'inline', não é chamada. Pelo contrário, 'operator!==(())' é chamado imediatamente. Note também que as operações com a pilha são sensivelmente modestas.

Assim, os objetos funções podem ser definidos 'inline'. Isto não é possível para funções chamadas indiretamente (i.e., usando apontadores a funções). Por isso, mesmo que o objeto função faça um pequeno trabalho, deve ser definido como uma função ordinária se tiver que ser chamada via apontadores. A sobrecarga da chamada indireta anula a vantagem da flexibilidade da chamada indireta de funções. Nesses casos a definição de objetos funções 'inline' pode resultar no aumento da eficiência do programa.

Finalmente, os objetos funções podem acessar os dados privados de seus objetos diretamente. Um algoritmo de busca, onde uma comparação é usada (como 'lsearch()') o objeto alvo e o conjunto são passados à função de comparação usando apontadores, envolvendo manipulação extra da pilha. Quando se usa objetos função, a pessoa alvo é passada ao construtor do objeto função que faz a comparação. Isto é, de fato, o que aconteceu na expressão 'target(base[idx])', onde um só argumento é passado à função membro 'operator()' do objeto função alvo.

Como foi notado, os objetos função desempenham um papel central em algoritmos genéricos. No Capítulo 17 discutimos esses algoritmos genéricos em detalhe. Ainda mais, nesse capítulo introduzimos objetos funções pré-definidos, enfatizando mais ainda a importância do conceito de objeto função.

9.10.1: Construção de manipuladores

No Capítulo 5 vimos construções como 'cout << hex << 13 << endl' para mostrar o valor 13 no formato hexadecimal. Podemos nos maravilhar com a magia do manipulador 'hex' para fazer isso. Nesta seção a construção de manipuladores como 'hex' é vista.

A construção de manipuladores é simples. Para começar é necessária uma definição do manipulador. Assumamos que queremos criar o manipulador 'w10' que põe o tamanho do campo a ser escrito a um objeto 'ostream' em 10. O manipulador é construído como uma função. A função 'w10' deverá saber sobre os objetos 'ostream' aos quais deve dar tamanho. Ao fornecer, à função, uma 'ostream' e um parâmetro, passamos esse conhecimento. Agora que a função sabe sobre objetos 'ostream', que nos estamos referindo, pode dar o tamanho ao objeto.

Em seguida, deve ser possível usar o manipulador numa sequência de inserção. Isto implica que o valor de retorno do manipulador deve fazer referência a um objeto 'ostream' também.

Das considerações acima estamos capacitados a construir nossa função 'w10':

```
#include <iosfwd>
#include <iomanip>
```

```
std::ostream &w10(std::ostream &str)
{
    return str << std::setw(10);
}
```

A função 'w10', claro, pode ser usada em modo independente, mas também pode ser usada como um manipulador. P.ex.:

```
#include <iostream>
#include <iomanip>

using namespace std;

extern ostream &w10(ostream &str);

int main()
{
    w10(cout) << 3 << " batatas fritas vendidas a América" << endl;
    cout << "E " << w10 << 3 << " mais batatas fritas vendidas." <<
endl;
}
```

A função 'w10' pode ser usada como manipulador porque a classe 'ostream' possui um operador '<<()' sobrecarregado, que aceita um apontador a uma função que espera um 'ostream &' e retorna um 'ostream &'. Sua definição é:

```
ostream& operator<<(ostream & (*func)(ostream &str))
{
    return (*func)(*this);
}
```

O procedimento acima não funciona para manipuladores com argumentos: é possível, claro, sobrecarregar o operador 'operator<<()' para aceitar uma referência a uma 'ostream' e o endereço de uma função que espere um 'ostream &' e, p.ex., um inteiro, mas enquanto o endereço dessa função pode ser especificado com '<<-operator', o argumento não pode ser especificado. Assim, nos admiramos que a seguinte construção foi implantada:

```
cout << setprecision(3)
```

Neste caso o manipulador é definido como uma macro. As macros são do domínio do pré-processador e pode facilmente sofrer de efeitos colaterais. Nos programas em linguagem C++ sempre devemos evitar macros, quando possível. A seção seguinte introduz uma forma de implantar manipuladores com argumentos sem recorrer a macros, mas usando objetos anônimos.

9.10.1.1: Manipuladores com argumentos

Os manipuladores com argumentos são implantados como macros: são manipulados pelo pré-processador e não estão disponíveis além do estágio de pré-processamento. O problema parece ser que não se pode chamar uma função numa sequência de inserção: numa sequência de chamada a 'operator<<()' o compilador primeiro chamará as funções e então usa seu valor de retorno na sequência de inserção. Isto invalida a ordem dos argumentos passados ao '<<-operator'.

Assim, considera-se construir outro 'operator<<()' sobrecarregado que aceite o endereço de uma função que receba não só a referência à 'ostream', mas uma série de outros argumentos também. O problema agora é que não está claro como a função receberá seus argumentos: não se pode só chamá-la, já que isto produz o problema acima mencionado e não se pode só passar seu endereço na sequência de inserção, como normalmente se faz com um manipulador....

Contudo, eis uma solução baseada no uso de objetos anônimos:

- Primeiro, uma classe é construída, p.ex., 'Alinhe', cujo construtor espera múltiplos argumentos. No nosso exemplo representam respectivamente o campo tamanho e o alinhamento.
- Além disso definimos a função:

```
ostream &operator<< (ostream &ostr, Alinhe const &alinhe)
```

Assim podemos inserir um objeto 'Alinhe' na 'ostream'.

Eis um exemplo de um pequeno programa usando tal manipulador 'feito em casa' que espera múltiplos argumentos:

```
#include <iostream>
#include <iomanip>

class Alinhe
{
    unsigned d_width;
    std::ios::fmtflags d_alinhamento;

public:
    Alinhe(unsigned width, std::ios::fmtflags alinhamento)
    :
        d_width(width),
        d_alinhamento(alinhamento)
    {}
    std::ostream &operator() (std::ostream &ostr) const
```

```

        {
            ostr.setf(d_alinhamento, std::ios::ajustecampo);
            return ostr << std::setw(d_width);
        }
};

std::ostream &operator<<(std::ostream &ostr, Alinhe const &alinhe)
{
    return alinhe(ostr);
}

using namespace std;

int main()
{
    cout
        << "'" << Alinhe(5, ios::left) << "olá" << "'"
        << "'" << Alinhe(10, ios::right) << "lá" << "'" << endl;
}

/*
Saída Gerada:

'olá  ' '    lá'
*/

```

Note que para inserir um objeto anônimo 'Alinhe' na 'ostream' a função 'operator<<' precisa definir um parâmetro 'Alinhe const &' (atenção ao modificador 'const').

9.11: Operadores Sobrecarregáveis

Os seguintes operadores podem ser sobrecarregados:

+	-	*	/	%	^	&	
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	
+=	-=	*=	/=	%=	^=	&=	=
<<=	>>=	[]	()	->	->*	new	new[]
delete	delete[]						

Quando estão disponíveis alternativas textuais (p.ex., and for &&) também são sobrecarregáveis.

Muitos desses operadores só podem ser sobrecarregados como funções membro dentro de uma classe. Isto é verdade para os operadores: '=', '()' e '->'. Conseqüentemente não é possível redefini-los, p.ex., globalmente o operador adjudicação, de tal forma que aceite um 'char const *' como um valor à esquerda e uma 'String &' como um valor à direita. Afortunadamente tão pouco é necessário, como vimos na seção 9.3.

Finalmente os seguintes operadores não são sobrecarregáveis em absoluto:

. .* :: ?: sizeof typeid

Capítulo 10: Funções e Dados Estáticos

Nos capítulos anteriores vimos exemplos de classes onde cada objeto de uma classe tem seu próprio conjunto de dados públicos e privados. Cada membro público ou privado pode acessar qualquer outro membro de qualquer objeto de sua classe.

Em algumas situações é desejável que um ou mais campos de dados comuns existam, acessíveis a todos os objetos da classe. Por exemplo, o nome do diretório de partida, usado por um programa que recursivamente varre a árvore de diretórios de um disco. Um segundo exemplo é uma flag variável que estabelece onde alguma iniciação específica ocorreu: só o primeiro objeto da classe pode realizar a iniciação necessária e colocar a flag como ``feito'`.

Tais situações são análogas ao código de C, onde diversas funções precisam acessar a mesma variável. Uma solução comum em C é definir todas estas funções num único arquivo fonte e declarar a variável como estática: o nome da variável, então, é desconhecido fora do escopo do arquivo fonte. Esta solução é perfeitamente válida, mas viola nosso princípio de usar só uma função por arquivo fonte. Outra solução em C é dar à variável um nome inusitado, p.ex., `_6uldy8`, na esperança de que outras partes do programa não use este nome por acidente. Nenhuma das soluções em C é elegante.

A solução da linguagem C++ é definir membros estáticos: dados e funções, comuns a todos os objetos da classe e inacessíveis fora da classe. Estes membros estáticos são o tópico deste capítulo.

10.1: Dados Estáticos

Um membro de dados de uma classe pode ser declarado estático; estando na seção pública ou privada da definição da classe. Tais membros de dados são criados e iniciados só uma vez, em contraste com os membros não estáticos que são criados uma e outra vez para cada membro separado da classe.

Os membros de dados estáticos são criados quando o programa parte. Note, contudo sempre são criados como verdadeiros membros de suas classes. Sugerimos que o nome dos membros estáticos tragam o prefixo `'s_'` para se distinguirem (das funções membro da classe) dos membros de dados da classe (que de preferência comecem por `'d_'`).

Os membros de dados estáticos são como variáveis globais ``normais'`: podem ser acessados por todo o código do programa, simplesmente usando seus nomes de classe, o operador de resolução de

escopo e seus nomes de membro. Isto está ilustrado no seguinte exemplo:

```
class Test
{
    static int s_private_int;

    public:
        static int s_public_int;
};

int main()
{
    Test::s_public_int = 145;    // ok

    Test::s_private_int = 12;    // Errado, não toca a
                                // parte privada

    return 0;
}
```

Este fragmento de código não é desejável a um compilador C++: somente ilustra a interface e não a implantação de membros estáticos de dados, que é discutida em seguida.

10.1.1: Dados Privados estáticos

Para ilustrar o uso de um membro de dados estático que é uma variável privada de uma classe, considere o seguinte exemplo:

```
class Directory
{
    static char s_path[];

    public:
        // construtores, destrutores, etc. (não mostrado)
};
```

O membro de dados 's_path[]' é privado estático. Durante a execução do programa somente um 'Directory::s_path[]' existe, mesmo que mais que um objeto da classe 'Directory' exista. Este membro de dados pode ser inspecionado ou alterado pelo construtor, destrutor ou qualquer outra função membro da classe 'Directory'.

Enquanto para cada novo objeto de uma classe o construtor é chamado, os membros de dados estáticos nunca são iniciados pelos construtores. Quando muito são modificados. A razão é que os membros estáticos existem antes de que qualquer construtor da classe seja chamado. Os membros de dados estáticos são iniciados ao serem definidos, fora de qualquer função membro, do mesmo modo que

qualquer outra variável global é iniciada.

A definição e iniciação de um membro de dados estático usualmente ocorre em um dos arquivos fonte das funções da classe, de preferência num arquivo fonte dedicado à definição dos membros de dados estáticos, chamado `data.cc`.

O membro de dados 's_path[]', usado acima, deveria então ser definido e iniciado como segue num arquivo 'data.cc':

```
include "directory.ih"

char Directory::s_path[200] = "/usr/local";
```

Na interface de classe o membro estático só é declarado. Em sua implantação (definição) seu tipo e nome da classe são explicitamente mencionados. Note também que a especificação de tamanho pode estar na interface, como mostra acima. Contudo, seu tamanho é (explícita ou implicitamente) requerida quando definido.

Note que qualquer arquivo fonte pode conter a definição dos membros de dados estáticos de uma classe. Uma fonte 'data.cc' separada é notificada, mas a fonte que contém, p.ex., 'main()' também pode ser usada. De certo que qualquer arquivo fonte que define dados estáticos de uma classe precisa incluir o arquivo cabeçalho dessa classe, para dar conhecimento dos membros de dados estáticos ao compilador.

Um segundo exemplo de um membro de dados estático útil é dado abaixo. Assume que uma classe 'Graphics' define a comunicação do programa com um dispositivo gráfico (p.ex., uma tela VGA). A iniciação do dispositivo, que no caso poderia ser chavear de modo texto a modo gráfico, é uma ação do construtor e depende de uma variável de flag estática 's_nobjects'. A variável 's_nobjects' simplesmente conta o número de objetos 'Graphics' presentes de uma vez. Similarmente o destrutor da classe pode chavear de volta a modo texto quando o último objeto 'Graphics' deixe de existir. A interface de classe para 'Graphics' poderia ser:

```
class Graphics
{
    static int s_nobjects;           // conta o # de objetos

public:
    Graphics();
    ~Graphics();                    // outros membros não aparecem.
private:
    void setgraphicsmode();          // chaveia para o modo gráfico
    void settextmode();              // chaveia para o modo texto
```

```
}
```

O propósito da variável 's_nobjects' é contar o número de objetos existentes num determinado momento. Quando o primeiro objeto é criado, o dispositivo gráfico é iniciado. Na destruição do último objeto 'Graphics', o chaveamento do modo gráfico para o modo texto é feito:

```
int Graphics::s_nobjects = 0;           // o membro de dados estático

Graphics::Graphics()
{
    if (!s_nobjects++)
        setgraphicsmode();
}

Graphics::~~Graphics()
{
    if (--s_nobjects)
        settextmode();
}
```

Obviamente quando a classe 'Graphics' define mais que um construtor, cada construtor necessita incrementar a variável 's_nobjects' e possivelmente tem que iniciar o modo gráfico.

10.1.2: Dados Públicos estáticos

Pode-se declarar dados na seção públicos de uma classe, contudo não seja prática comum (já que violaria o princípio de encobrimento de dados). P.ex., quando o membro de dados estáticos 's_path[]' da seção 10.1, estiver na seção pública da definição da classe, todos o programa poderia acessar esta variável:

```
int main()
{
    getcwd(Directory::s_path, 199);
}
```

Note que a variável 's_path' poderia ainda estar por definir. Como antes, a interface de classe só declararia o conjunto 's_path[]'. Isto significa que a definição de 's_path[]' deve estar em algum arquivo fonte de todas formas.

10.1.3: Iniciando Dados Estáticos Constantes

Os membros de dados estáticos constantes podem ser iniciados na interface de classe se esses

membros são de tipo de dado integral. Assim, no exemplo seguinte os primeiros três membros de dados estáticos podem ser iniciados, já que 'enum' e os tipo duplos são membros de dados integrais. O último membro de dados estático não pode ser iniciado na interface de classe, porque 'string' não é um tipo de dado integral:

```
class X
{
    public:
        enum Enum
        {
            FIRST,
        };

        static int const s_x = 34;
        static Enum const s_type = FIRST;

        static double const s_d = 1.2;
        static string const s_str = "a";    // não compila
};
```

Os membros de dados estáticos constantes iniciados na interface de classe não são variáveis endereçáveis. São meros nomes simbólicos para seus valores associados. Como não são variáveis, não é possível determinar seus endereços. Note que isto não é um problema de compilação, mas um problema de linkagem. A variável estática constante iniciada na interface de classe não existe como entidade endereçável.

Um comando como 'int *ip = &X::s_x' compilará corretamente, mas falhará na linkagem. Variáveis Estáticas que são explicitamente num arquivo fonte linkam corretamente. Assim, no exemplo seguinte o endereço de 'X::s_x' não pode ser resolvido pelo linkador, mas o endereço de 'X::s_y' é resolvido:

```
class X
{
    public:
        static int const s_x = 34;
        static int const s_y;
};

int const X::s_y = 12;

int main()
{
    int const *ip = &X::s_x;    // compiles, mas falha na linkagem
    ip = &X::s_y;               // compila e linka corretamente
}
```

10.2: Funções Membro Estáticas

Além de membros de dados estáticos, a linguagem C++ permite a definição de funções membro estáticas. Similar ao conceito de dados estáticos, onde estas variáveis são compartilhadas por todos os objetos da classe, funções membro estáticas existem sem qualquer objeto associado de sua classe.

As funções membro estáticas podem acessar qualquer membro estático de sua classe, mas também membros (privados ou públicos) de objetos de sua classe se estão informadas da existência desses objetos, como no exemplo que vem. As funções membro estáticas não estão associadas a qualquer objeto de sua classe. Conseqüentemente não possuem um ponteiro 'this'. De fato uma função membro estática é completamente comparável a uma função global, não associada a nenhuma classe (i.e., na prática o são. Veja a seção 10.2.1 para uma nota sutil). Como as funções membro estáticas não requerem um objeto associado, quando declaradas na seção pública da interface de classe podem ser chamadas sem a menção de um objeto de sua classe. O exemplo seguinte ilustra esta característica das funções membro estáticas:

```
class Directory
{
    string d_currentPath;
    static char s_path[];

    public:
        static void setpath(char const *newpath);
        static void preset(Directory &dir, char const *path);
}

// veja o texto abaixo
char Directory::s_path[200] = "/usr/local"; // 1

void Directory::setpath(char const *newpath)
{
    if (strlen(newpath) >= 200)
        throw "newpath too long";

    strcpy(s_path, newpath); // 2
}

void Directory::preset(Directory &dir, char const *newpath)
{
    dir.d_currentPath = newpath; // 3
}

int main()
```

```

{
    Directory dir;

    Directory::setpath("/etc");           // 4
    dir.setpath("/etc");                 // 5

    Directory::preset(dir, "/usr/local/bin"); // 6
    dir.preset(dir, "/usr/local/bin");     // 7
}

```

- Em 1 um conjunto relativamente longo é definido para ser capaz de acomodar longos caminhos. Alternativamente se poderia usar um ponteiro a memória dinâmica;
- Em 2 um (possivelmente longo, mas não demais) novo caminho é guardado no membro de dados estáticos 's_path[]'. Note que só são usados membros estáticos.
- Em 3 uma função membro estática modifica um membro de dados de um objeto. Contudo o objeto cujo membro precisa ser modificado é dado à função membro como parâmetro de referência;
- Em 4 é chamada 'setpath()'. É um membro estático, portanto não necessita de objeto. Mas o compilador precisa saber a que classe a função pertence, assim a classe é mencionada usando-se o operador de resolução de escopo;
- Em 5 se realiza o mesmo que em 4. Mas aqui 'dir' é usado para dizer ao compilador de que se trata de uma função da classe 'Directory'. Assim, as funções membro estáticas podem ser chamadas como funções membros normais;
- Em 6 é alterado o membro de 'dir' 'currentPath'. Como em 4 são usados o nome da classe e o operador de resolução de escopo;
- Em 7 se realiza o mesmo que em 6. Mas aqui se usa 'dir' para dizer ao compilador que nos referimos a uma função da classe 'Directory'. Note que, em particular, aqui 'preset()' não está sendo usada como uma função membro ordinária de 'dir': A função não permite o ponteiro 'this', assim, 'dir' precisa ser passado como argumento para informar a função membro estática 'preset' sobre qual objeto 'currentPath' modificar.

No exemplo se usou só funções membro estáticas públicas. A linguagem C++ também permite a definição de funções membro estáticas privadas: Estas funções só podem ser chamadas por funções membro de sua classe.

Finalmente é de se notar que as funções membro estáticas não podem ser definidas como

funções 'inline'. Enquanto uma função estática pode ser usada como função normal a nível global, precisa ter um endereço. O código de uma função 'inline' normalmente é inserido no código do programa: Não é uma verdadeira função, mas alguns comandos. Nas funções 'inline' não temos o 'endereço da função'. Consequentemente as funções 'inline' não possuem um requerimento básico das funções membro estáticas: Não possuem endereço.

10.2.1: Convenções de Chamada

Como notamos na seção anterior, as funções membro estáticas (públicas) são comparáveis a funções sem classe. Contudo formalmente esta afirmação não é verdadeira, já que o estandarte C++ não prescreve as mesmas convenções de chamada para funções membro e para funções globais sem classe.

Na prática estas convenções de chamada são idênticas, implicando que o endereço de uma função membro estática pode ser usada como argumento em funções com parâmetros que são ponteiros a funções (globais).

Se queremos evitar surpresas desagradáveis a todo custo, é sugerido criar uma função envoltório global sem classe em volta das funções membro estáticas que precisa ser usada como endereço de retorno de outras funções.

Reconhecendo que a situação tradicional onde funções de endereço de retorno são usadas na linguagem C se mantém em C++ usando algoritmos modelo (veja Capítulo 17), assumamos que temos uma classe 'Pessoa' com membros de dados que representam o nome, endereço, telefone e peso de uma pessoa. Mais ainda, assuma que queremos ordenar um conjunto de ponteiros a objetos 'Pessoa', comparando os objetos 'Pessoa' para os quais esses ponteiros apontam. Para manter as coisas simples, assumimos que uma pública estática existe:

```
int Pessoa::compare(Pessoa const *const *p1, Pessoa const *const *p2);
```

Uma característica útil deste membro é que pode inspecionar diretamente os membros de dados requeridos de dois objetos 'Pessoa' passados à função membro usando dois ponteiros.

Muitos compiladores permitem passar o endereço desta função como o endereço da função de comparação a função da C 'qsort()'. P.ex.:

```
qsort
(
    pessoaArray, nPessoas, sizeof(Pessoa *),
    reinterpret_cast<int (*) (const void *, const void
*)>(Pessoa::compare)
);
```

Contudo se o compilador usa uma convenção de chamada diferente para membros estáticos e para funções sem classe, isto não funcionará. Nesse caso uma função envolvente como a seguinte pode ser de proveito:

```
int compareWrapper(void const *p1, void const *p2)
{
    return
        Pessoa::compare
        (
            reinterpret_cast<Pessoa const *const *>(p1),
            reinterpret_cast<Pessoa const *const *>(p2)
        );
}
```

Resultando na seguinte chamada da função 'qsort()':

```
qsort(pessoaArray, nPessoas, sizeof(Person *), compareWrapper);
```

Note:

- A função envolvente cuida de qualquer engano na convenção de chamada a funções membro estática e funções sem classe;
- A função envolvente manipula os 'casts' de tipos requeridos;
- A função envolvente pode realizar pequenos serviços adicionais (como eliminar ponteiros de referência se a função membro estática espera referências a objetos 'Pessoa' antes que duplos ponteiros);
- Como notamos antes: Nas funções dos programas da linguagem C++ como 'qsort()', que requer a especificação de um endereço de retorno são raramente usadas a favor de algoritmos modelo genéricos existentes (veja Capítulo 17).

Capítulo 11: Amigos (Friends)

Em todos os exemplos que discutimos até agora vimos que os membros privados só são acessíveis pelos membros de suas classes. Isto é bom, já que força os princípios de encapsulamento e encobrimento de dados: Encapsulando os dados de um objeto podemos evitar que o código externo à classe se torne dependente de implantação dos dados de uma classe e encobrindo os dados do código externo podemos controlar as modificações dos dados, ajudando-nos a manter a integridade dos dados.

Neste pequeno capítulo introduziremos a palavra chave 'friend' como um meio de permitir que as funções externas tenham acesso a membros privados de uma classe. As situações onde é natural usar a amizade entre classes são discutidas nos capítulos 16 e 18.

A amizade (i.e., o uso da palavra chave 'friend') é um tópico complexo e perigoso por várias razões:

- A amizade quando aplicada no projeto do programa é um mecanismo de escape que nos permite transpor os princípios de encapsulamento e encobrimento de dados. O uso de 'friend' deve, portanto, ser minimizado a situações onde pode ser usado naturalmente;
- Se usarmos 'friend' percebemos que as funções ou classes amigas de tornam dependentes da implantação da classe que as declaram como amigas. Uma vez que a organização interna dos dados de uma classe declarada como 'friend' muda, todos os seus amigos precisam ser re-compilados (e possivelmente modificados) também.
- Por isso, como regra prática: Não use 'friend' em funções ou classes.

Contudo há situações onde a palavra chave 'friend' pode ser usada com segurança e naturalmente. O propósito deste capítulo é introduzir a sintaxe requerida e desenvolver princípios que permitam reconhecer os casos onde a palavra chave 'friend' pode ser usada com bem pouco peigo.

Permita-nos considerar uma situação onde seria bom para uma dada classe ter acesso a outra classe. Tal situação pode ocorrer quando queremos dar acesso a uma classe desenvolvida antes historicamente a uma classe desenvolvida posteriormente.

Desafortunadamente ao desenvolver a classe mais antiga, ainda não se sabia que a nova classe seria desenvolvida. Conseqüentemente não se previu onde, na antiga classe, acessar a informação da nova

classe.

Considere a seguinte situação. O operador de inserção pode ser usado para inserir informação numa 'stream'. Este operador aceita dados de diferentes tipos: 'int', 'double', 'char *', etc.. Antes (Capítulo 7) introduzimos a classe 'Pessoa'. A classe 'Pessoa' tem membros para retirar dados guardados em objetos de 'Pessoa', como 'char const *Pessoa::nome()'. Estes membros podem ser usados para 'inserir' um objeto 'Pessoa' numa 'stream', como mostra na seção 9.2.

Com a classe 'Pessoa' a implantação dos operadores inserção e extração é pouco otimizada. O operador inserção usa membros acessórios que podem ser implantados como funções 'inline', isto disponibiliza os membros de dados privados a uma inspeção direta. O operador extração requer o uso de membros modificadores que dificilmente podem ser implantados diferentemente: A memória antiga sempre tem que ser apagada e o novo valor tem que ser copiado à memória recém fornecida.

Mas vamos dar uma olhada na classe 'Dados_Pessoa', introduzida na seção 9.4. Parece que esta classe tem pelo menos os seguintes membros de dados (privados):

```
class Dados_Pessoa
{
    Pessoa *d_pessoa;
    size_t d_n;
};
```

Quando construímos um operador de inserção sobrecarregado para um objeto 'Dados_Pessoa', p.ex., que insira a informação de todas as suas pessoas numa 'stream', o operador de inserção sobrecarregado é implantado ineficientemente quando uma pessoa individualmente precisa ser acessada usando o operador índice.

Nestes casos, onde o acessor e modificador de membros tende a ficar complexo, o acesso direto aos membros de dados privados pode incrementar a eficiência. Assim, no contexto da inserção e extração, buscamos funções membro de implantação das operações inserção e extração tendo acesso aos membros de dados privados a serem inseridos ou extraídos. Para se implantar tais funções não membro elas têm que ter acesso aos membros privados da classe. A palavra chave 'friend' é usada para isto.

11.1: Funções 'Friend'

Concentrando-nos na classe 'Dados_Pessoa', nossa implantação inicial do operador inserção é:

```
ostream &operator<<(ostream &str, Dados_Pessoa const &pd)
```

```

{
    for (size_t idx = 0; idx < pd.nPeasos(); idx++)
        str << pd[idx] << endl;
}

```

Esta implantação realizará sua tarefa como o esperado: Usando o operador de inserção (sobrecarregado) da classe 'Pessoa', a informação cobre cada pessoa guardada no objeto 'Dados_Pessoa' será escrito numa linha separada.

Contudo, repetidas chamadas ao operador índice reduz a eficiência da implantação. No lugar de usar diretamente o conjunto 'd_Pessoa' aumentaria a eficiência da função acima.

Neste ponto nos perguntamos se considerar o operador acima 'operator<<()' primeiro como uma extensão da função 'operator<<()' existente globalmente ou de fato uma função membro da classe 'Dados_Pessoa'. Dito de outra forma: Assuma que fôssemos capazes de fazer 'operator<<()' um membro verdadeiro da classe 'Dados_Pessoa', poderíamos objetar? Provavelmente não, já que a tarefa da função está muito ligada à classe 'Dados_Pessoa'. Nesse caso, a função, pode sensivelmente se tornar um amigo ('friend') da classe 'Dados_Pessoa', para isso permitindo a função acessar aos dados privados da classe 'Dados_Pessoa'.

Funções amigas devem ser declaradas como 'friend' na interface de classe. Essas declarações de amizade não se referem nem às funções privadas nem às públicas, assim, a declaração 'friend' pode estar em qualquer parte da interface de classe. A convenção dita que as declarações 'friend' estão listadas no topo da interface de classe. Desse modo, para a classe 'Dados_Pessoa' obtemos:

```

class Dados_Pessoa
{
    friend ostream &operator<<(ostream &stream, Dados_Pessoa &pd);
    friend istream &operator>>(istream &stream, Dados_Pessoa &pd);

    public:
        // o resto da interface
};

```

A implantação do operador de inserção agora pode ser alterada para permitir ao operador de inserção o acesso direto aos membros de dados privados dos objetos 'Dados_Pessoa':

```

ostream &operator<<(ostream &str, Dados_Pessoa const &pd)
{
    for (size_t idx = 0; idx < pd.d_n; idx++)
        str << pd.d_pessoa[idx] << endl;
}

```

Mais uma vez, as funções amigas são consideradas aceitáveis ou não ficam como um tema de preferência: Se a função é de fato considerada uma função membro, mas não pode ser definida como função membro devido à natureza da gramática de C++, então é aceitável o uso da palavra chave 'friend'. Em outros casos devemos evitar seu uso, devido aos princípios de encapsulamento e encobrimento dos dados.

Explicitamente note que se queremos estar habilitados a inserir objetos 'Dados_Pessoa' em objetos 'ostream' sem usar a palavra chave 'friend' o operador inserção não deve estar na classe 'Dados_Pessoa'. Neste caso 'operator<<()' é uma variante normal sobrecarregada do operador inserção, que deve ser declarado e definido fora da classe 'Dados_Pessoa'. Esta situação é a, p.ex., do exemplo no início desta seção.

11.2: 'Inline friends'

Na seção anterior estabelecemos que amigas podem ser consideradas funções membro de uma classe, embora dadas as características da função evitamos defini-la como função membro. Nesta seção estenderemos esta linha de raciocínio um pouco mais.

Se consideramos, conceitualmente, uma função amiga como função membro, poderíamos criar uma verdadeira função membro, esta realizaria as mesmas tarefas que a função amiga. Por exemplo, podemos construir uma função que insira um objeto 'Dados_Pessoa' numa 'ostream':

```
ostream &Dados_Pessoa::insertor(ostream &str) const
{
    for (size_t idx = 0; idx < d_n; idx++)
        str << d_pessoa[idx] << endl;
    return str;
}
```

Esta função membro pode ser usada por um objeto 'Dados_Pessoa' para inserir o objeto em 'ostream str':

```
Dados_Pessoa pd;

cout << "A informação da Pessoa no objeto Dados_Pessoa é:\n";
pd.insertor(str);
cout << "=====\n";
```

Tendo em conta que esse 'insertor()' faz o mesmo que o operador de inserção definido atrás como uma amiga, poderíamos simplesmente chamar o membro 'insertor()' no código da função amiga 'operator<<()'. Agora esta função 'operator<<()' precisa só um comando: Chamar 'insertor()'.

Conseqüentemente:

- A função 'insertor()' pode esconder-se na classe fazendo-a privada, já que não há necessidade de ser chamada em outra parte;
- A 'operator<<()' pode ser construída como função 'inline', como contém só um comando.

Assim a parte relevante da interface de classe de 'Dados_Pessoa' fica:

```
class Dados_Pessoa
{
    friend ostream &operator<<(ostream &str, Dados_Pessoa const &pd)
    {
        return pd.insertor(str);
    }
private:
    ostream &insertor(ostream &str) const;
};
```

O exemplo acima ilustra o passo final no desenvolvimento das funções amigas. Nos permite formular os seguintes princípios:

Apesar de que as funções amigas possuem acesso aos membros privados de uma classe, esta característica não pode ser usada indiscriminadamente, pois resulta numa severa brecha no princípio de encapsulamento, por isso devemos fazer funções fora da classe dependentes da implantação dos dados da classe.

Em vez disso, se a tarefa de uma função amiga pode ser implantada por uma função membro, podemos responder que essa amiga é um mero sinônimo sintático ou apelido para a função membro.

A interpretação de uma função amiga como um sinônimo de uma função membro se concretiza construindo a função amiga como função 'inline'.

Assim, instituímos como princípio que as funções amigas devem ser evitadas, a menos que possam ser construídas como funções 'inline', com um só comando, onde é chamado um membro apropriado.

Usando este princípio nos asseguramos que todo o código que tem acesso aos dados privados de uma classe restarão confinados na classe. Isto, mesmo para a função amiga, já que são definidas como meras funções 'inline'.

Capítulo 12: Recipientes Abstratos (Containers)

A linguagem C++ oferece diversos tipos de dados pré-definidos, todos parte da Standard Template Library (Biblioteca Estandarte de Modelos), que podem ser usados para implantar soluções onde freqüentemente ocorrem problemas. Os tipos de dados discutidos neste capítulo são todos recipientes: Pode-se por coisas dentro deles e se pode retirar a informação guardada neles.

A parte interessante é que a natureza dos dados que podem guardar esses recipientes não foi especificada em sua construção. Por isso dizemos a seu respeito recipientes abstratos.

OS recipientes abstratos estão solidamente relacionados a modelos, que são vistos perto do fim das Anotações C++, no Capítulo 18. Contudo, para se usar os recipientes abstratos só uma mínima parte do conceito de modelo é necessária. Em C++ um modelo é de fato um recipiente para se construir uma função ou uma classe completa. O recipiente tenta abstrair a funcionalidade da classe ou função tanto quanto possível dos dados sobre os quais a classe ou função opera. Como os tipos de dados sobre os quais o modelo opera não era conhecido em sua construção, os tipos de dados são inferidos do contexto onde um modelo de função é usado ou são mencionados explicitamente quando um modelo de classe é usado (o termo usado aqui é ilustrativo). Em situações onde os tipos são mencionados explicitamente os parênteses quadrados (<>) são usados para indicar que tipos de dados são requeridos. Por exemplo, abaixo (na seção 12.2) encontraremos o par recipiente, que requer a menção explícita dos dois tipos de dados. P.ex., para definir um par de variáveis com um 'int' e uma 'string' é usada a notação:

```
pair<int, string> myPair;
```

Aqui, 'myPair' é definido como uma variável 'par', contendo um 'int' e uma 'string'.

Os parênteses quadrados são usados intensamente na discussão de recipientes abstratos. O entendimento desta parte dos modelos é o único requerimento real para se usar recipientes abstratos. Agora que introduzimos a notação podemos pós-por o resto da discussão de modelos ao Capítulo 18 e concentrar-nos em seu uso neste capítulo.

Muitos dos recipientes abstratos são seqüenciais: Representam uma série de dados que são guardados e retirados de forma seqüencial. Exemplos são vetores, que representam um conjunto extensível, listas, que implantam uma estrutura de dados que permite fácil inserção e eliminação, filas, também chamada 'FIFO' (first in first out), onde o primeiro elemento a entrar será o primeiro a sair, e pilha, que é uma estrutura primeiro a entrar último em sair (FILO ou LIFO).

Além dos recipientes seqüenciais diversos outros especiais estão disponíveis. O par é um

recipiente básico no qual um par de valores (de tipos em aberto para especificação posterior) pode ser guardado, como duas 'strings', dois 'ints', uma 'string' e um 'double', etc.. Os pares são freqüentemente usados para retornar dados que naturalmente vêm em pares.

Uma variante do par é um recipiente complexo, que implanta operações sobre números complexos.

Todos os recipientes descritos neste capítulo e o tipo 'string' discutido no Capítulo 4 fazem parte da Biblioteca Estandarte de Modelos. Existem também recipientes abstratos para implantação de uma tabela 'hash', mas esse recipiente não é (ainda) aceito pelo estandarte ANSI/ISO. Apesar de tudo, a seção final deste capítulo cobrirá as tabelas 'hash' em certa extensão. É esperado que recipientes como mapas 'hash' e outros, ainda considerados extensões, se tornem parte do padrão ANSI/ISO na próxima distribuição: Aparentemente, neste momento, a estandarização congelou esses recipientes que ainda não estavam completamente prontos. Agora que estão disponíveis não podem ser parte oficial, contudo são extensões.

Todos os recipientes suportam os seguintes operadores:

- O operador de adjudicação sobrecarregado, assim, podemos igualar dois recipientes de mesmo tipo um ao outro.
- Teste de igualdade: '==' e '!='. Este operador aplicado a dois recipientes retorna verdadeiro se os dois recipientes possuem o mesmo número de elementos.
- Operadores de ordem: '<', '<=', '>' e '>='. O operador '<' retorna verdadeiro se cada elemento no recipiente à esquerda é menor que o correspondente no recipiente à direita. Se o recipiente à direita tem mais elementos que o à esquerda, estes são ignorados. Os outros operadores agem analogamente.
- Note que antes de um tipo definido pelo usuário (usualmente um tipo de classe) pode ser guardado num recipiente, o tipo do usuário precisa suportar pelo menos:
 - Um valor padrão (p.ex., um construtor padrão);
 - operador igualdade (==);
 - O operador menor que (<).

São muito ligados aos algoritmos genéricos da Biblioteca Estandarte de Modelos. Estes algoritmos podem ser usados para realizar tarefas freqüentes ou mais complexas que as possíveis com os

recipientes, como contagem, preenchimentos, uniões, filtrações, etc.. Uma visão geral dos algoritmos genéricos é dada no Capítulo 17. Os algoritmos genéricos usualmente contam com a disposição de `iterators`, que representam o ponto inicial e final do processamento de dados armazenados em recipientes. Os recipientes abstratos usualmente suportam construtores e membros que esperam `iterators` (comparáveis aos membros `string::begin()` e `string::end()`). No restante deste capítulo o conceito de `iterator` não é coberto. Refira-se ao Capítulo 17 para tal.

O sítio `http://www.sgi.com/Technology/STL` é de valor para os leitores que buscam mais informação sobre recipientes abstratos e a Biblioteca Estandarte de modelos que as Anotações C++ possam ser.

Os recipientes coletam dados durante suas existências. Quando um recipiente sai fora de escopo seu destrutor tenta destruir seus elementos de dados. Isto só sucede se os elementos de dados estão guardados no recipiente. Se os elementos de dados do recipiente são ponteiros os dados apontados não serão destruídos, resultando numa fuga de memória. Uma consequência deste esquema é que os dados guardados num recipiente podem ser considerados a `propriedade` do recipiente: O recipiente deve ser capaz de destruir seus elementos quando seu destrutor for chamado. Assim, normalmente, os recipientes não devem conter ponteiros. Um recipiente também não deve conter dados constantes, já que dados constantes evitam o uso de muitos membros dos recipientes, como o operador de adjudicação.

12.1: Notações usadas neste capítulo

Neste capítulo sobre recipientes é usada a seguinte convenção de notação:

- Os recipientes vivem no espaço nomeado `std`. Nos exemplos isto ficará claramente visível, mas no texto a referência `std::` usualmente é omitida;
- Um recipiente sem parênteses quadrados representa qualquer recipiente desse tipo. Mentalmente ponha o tipo requerido no parêntese quadrado. P.ex, `pair<string, int>`;
- A notação `Tipo` representa um tipo genérico. `Tipo` pode ser `int`, `string`, etc.;
- Identificadores de objetos e recipiente representam objetos do tipo de recipiente em discussão;
- O identificador de valor representa um valor do tipo que está armazenado no recipiente;
- Identificadores com uma simples letra, como `n` representam valores sem sinal;
- Identificadores longos representam iteradores. São exemplos `pos`, `from`, `beyond`.

Alguns recipientes, p.ex, o recipiente mapa, contém pares de valores, usualmente ditos `chaves` e `valores`. Para tais recipientes a seguinte notação é usada:

- O identificador `chave` indica um valor do tipo da chave;
- O identificador `valor_da_chave` indica um `valor do tipo` do valor usado com o recipiente particular.

12.2: O recipiente `pair`

O recipiente par é básico. É usado para conter dois elementos, ditos primeiro e segundo. Antes de poderem ser usados a seguinte diretiva ao pré-processador deve ser especificada:

```
#include <utility>
```

Os tipos de dados de um par são especificados quando o par de variáveis é definido (ou declarado), usando-se o modelo estandarte (veja o capítulo Modelos) parênteses quadrado:

```
pair<string, string> piper("PA28", "PH-ANI");  
pair<string, string> cessna("C172", "PH-ANG");
```

Aqui as variáveis 'piper' e 'cessna' são definidas como pares variáveis que contêm duas `strings`. Ambas `strings` podem ser retiradas usando-se 'first' para o primeiro campo e 'second' para o segundo campo do tipo par:

```
cout << piper.first << endl <<          // mostra 'PA28'  
      cessna.second << endl;           // mostra 'PH-ANG'
```

O primeiro e o segundo membros também podem ser usados para re-adjudicar valores:

```
cessna.first = "C152";  
cessna.second = "PH-ANW";
```

Se um objeto par precisa ser completamente re-adjudicado, um par anônimo de objetos pode ser usado como operando à direita da operação. Uma variável anônima define uma variável temporária (que não recebe nome) somente para o propósito de (re)adjudicação a outra variável do mesmo tipo. sua forma genérica é:

```
type(initializer list)
```

Note que quando um par de objetos é usado, a especificação de tipo não é completada, somente se menciona o nome do par recipiente. Requer também a especificação do tipo de dados guardados no par. Para isto a notação (modelo) com parênteses quadrados novamente é usada. P.ex., a re-

adjudicação do par de variáveis 'cessna' seria como segue:

```
cessna = pair<string, string>("C152", "PH-ANW");
```

Em casos como estes a especificação de tipo pode se tornar muito elaborada, o que causou o reviver do interesse na possibilidade oferecida pela palavra chave 'typedef'. Se muitas cláusulas 'pair<type1, type2>' são usadas numa fonte, o esforço de digitação pode ser reduzido e a legibilidade ser aumentada se definirmos primeiro um nome para a cláusula e então usar o nome definido. P.ex.:

```
typedef pair<string, string> pairStrStr;  
cessna = pairStrStr("C152", "PH-ANW");
```

Além disto (e o conjunto básico de operações (adjudicação e comparações)) o par oferece nenhuma outra funcionalidade. Contudo, é um ingrediente básico para se tornar nos recipientes mapa, multi-mapa e hash_mapa.

12.3: Recipientes Seqüenciais

12.3.1: O Recipiente 'vector'

A classe 'vector' implanta um conjunto expansível. Antes de se usar o recipiente vetor a seguinte diretiva ao pré-processador deve ser especificada:

```
#include <vector>
```

os seguintes construtores, operadores e funções membro estão disponíveis:

- Construtores:
 - Um vetor pode ser construído vazio:

```
vector<string> object;
```

Note a especificação do tipo de dados a serem guardados no vetor: O tipo de dados é dado entre parênteses quadrados, bem depois do nome 'vector'. Esta é uma prática comum entre os recipientes.

- Um vetor pode ser iniciado a um certo número de elementos. Uma das boas características dos vetores (e outros recipientes) é que inicia o tipo de dados em seu valor padrão. O construtor com tipos padrão é usado para isto. Com tipos de dados sem classe é usado zero. Assim para um vetor de inteiros sabemos que seus valores iniciais são zeros. Alguns exemplos:

```
vector<string> object(5, string("Alô")); // inicia com 5 Alô  
vector<string> container(10);           // e com 10 strings vazias
```

- Um vetor pode ser iniciado usando iteradores. Para iniciar um vetor de 5 até 10 (incluindo o último) de um vetor existente 'vector<string>' deve-se usar a seguinte construção:

```
extern vector<string> container;  
vector<string> object(&container[5], &container[11]);
```

- Note que o último elemento apontado pelo segundo iterador ('&container[11]') não é guardado no objeto. Este é um simples exemplo do uso de iteradores, no qual a gama de valores usada com o primeiro valor e inclui todos os elementos até o último excluído a que se refere o segundo iterador. A notação padrão para isto é [início, fim].

- Um vetor pode ser iniciado usando-se um construtor de cópias:

```
extern vector<string> container;  
vector<string> object(container);
```

- Além dos operadores padrão para recipientes, o vetor suporta o operador índice, que pode ser usado para retirar ou re-adjudicar elementos individuais do vetor. Note que os elementos indexados precisam existir. Por exemplo, definindo um vetor vazio um comando como 'ivec[0] = 18' produz um erro, já que o vetor está vazio. Portanto, o vetor não é expandido automaticamente e isto respeito às suas margens. Neste caso o vetor deve ser re-dimensionado primeiro ou pode-se usar 'ivect.push_back(18)' (veja abaixo);
- A classe 'vector' tem as seguintes funções membro:
 - 'Tipo &vector::back()': Este membro retorna uma referência ao último elemento do vetor. É de responsabilidade do programador usar o membro só se o vetor não estiver vazio;
 - 'vector::iterator vector::begin()': Este membro retorna um iterador que aponta o primeiro elemento do vetor, retorna 'vector::end()' se o vetor está vazio;
 - 'vector::clear()': Apaga todos os elementos do vetor;
 - 'bool vector::empty()': Retorna verdadeiro se o vetor está vazio;
 - 'vector::iterator vector::end()': Retorna um iterador que aponta além do último elemento do vetor;

- `'vector::iterator vector::erase()'`: Usado para apagar uma gama de elementos específica do vetor;
- `'erase(pos)'`: Apaga o elemento apontado pelo iterador 'pos'. O valor '++pos' é retornado;
- `'erase(firsr, beyond)'`: Apaga os elementos indicados pelo iterador 'range[first, beyond]'. 'beyond' é retornado;
- `'Tipo &vector::front()'`: Retorna uma referência ao primeiro elemento do vetor. É responsabilidade do programador usar o membro só se o vetor não estiver vazio;
- `'... vetor::insert()'`: Elementos a inserir começando de certa posição. O valor de retorno depende da versão de 'insert()' que é chamada:
 - `'vector::iterator insert(pos)'`: insere um valor padrão de tipo 'Tipo' na posição 'pos', retorna 'pos';
 - `'vector::iterator insert(pos, value)'`: insere 'value' na posição 'pos', pos é retornado;
 - `'void insert(pos, first, beyond)'`: insere os elementos no rango do iterador '[first, beyond]';
 - `'void insert(pos, n, value)'`: insere n elementos com valor 'value' na posição 'pos';
- `'void vector::pop_back()'`: Remove o último elemento do vetor. Com um vetor vazio nada acontece;
- `'void vector::push_back(value)'`: Adiciona 'value' ao fim do vetor;
- `'void vrtor::resize()'`: Altera o número de elementos no vetor:
 - `'resize(n,value)'` para re-dimensionar o vetor para n elementos. 'value' é opcional. Se o vetor é expandido e 'value' não é fornecido, os elementos adicionais são iniciados com o valor padrão do tipo usado, de outra forma são iniciados com 'value';
- `'vector::reverse_iterator vector::rbegin()'`: Retorna um iterador que aponta para o último elemento do vetor;

- 'vector::reverse_iterator vector::rend()': Retorna um iterador que aponta para antes do primeiro elemento do vetor;
- 'size_t vector::size()': Retorna o número de elementos do vetor.
- 'void vector::swap()': Troca dois vetores com mesmo tipo. P.ex.:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1(7);
    vector<int> v2(10);

    v1.swap(v2);
    cout << v1.size() << " " << v2.size() << endl;
}
/*
    Produz a saída:
10 7
*/
```

12.3.2: O recipiente 'list'

Este recipiente implanta uma estrutura de dados em lista. Para se usar o recipiente 'list' deve-se especificar a diretiva ao pré-processador:

```
#include <list>
```

A organização de uma lista é mostrada na figura 7.

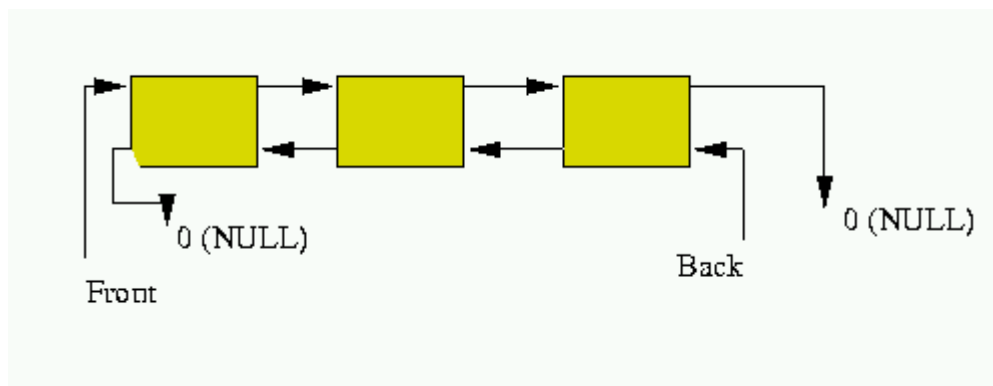


Figura 7(Uma estrutura de dados lista)

Na figura 7 mostra-se que uma lista compõe-se de elementos em lista separados, conectados um ao outro por ponteiros. A lista pode ser percorrida em duas direções: Iniciando na Frente da lista indo-se da esquerda à direita, até encontrar um apontador nulo no Fim, o elemento mais à direita. A lista também pode ser percorrida da direita para a esquerda: Iniciando no Fim, até o apontador nulo mais à esquerda, na Frente.

Como sutileza note que a representação dada na figura 7 não necessariamente é usada na implantação da lista. Por exemplo, considere o pequeno programa seguinte:

```
int main()
{
    list<int> l;
    cout << "tamanho: " << l.size() << ", primeiro elemento: " <<
        l.front() << endl;
}
```

Quando este programa é executado produzirá a seguinte saída:

```
tamanho: 0, primeiro elemento: 0
```

Ao seu elemento frontal pode-se mesmo dar um valor. Neste caso foi escolhido um valor vazio, que é uma lista circular, onde o elemento vazio serve como terminal, substituindo o apontador nulo da figura 7. Como se fez notar, esta é uma sutileza que não afeta a noção conceitual de lista como uma estrutura de dados começando e terminando com apontadores nulos. Note também que é possível, como bem sabido, diferentes implantações da estrutura de uma lista (veja Aho, A.V., Hopcroft J.E. and Ullman, J.D., (1983) Data Structures and Algorithms (Addison-Wesley)).

Os vetores e as listas, freqüentemente são estruturas de dados apropriadas para guardar um número desconhecido de elementos de dados. Contudo, há certas regras práticas a seguir na escolha de qual das duas estruturas empregar.

- Quando a maioria dos acessos é aleatória, um vetor é a estrutura preferida. P.ex., um programa que conta a freqüência dos caracteres num arquivo de texto, um 'vector<int>frequencias(256)' fará o trabalho, já que os caracteres recebidos podem servir de índices de entrada ao vetor frequências.
- O exemplo anterior ilustra uma segunda regra prática, que também favorece o vetor: Se o número de elementos é conhecido de partida (e não muda notavelmente durante a execução do programa), o vetor é favorecido sobre a lista.
- Em casos onde a inserção ou eliminação prevalece, a lista é geralmente preferível. Na minha experiência, as listas não são tão úteis e freqüentemente a implantação será mais rápida quando

usado um vetor, talvez com buracos.

Outras considerações relativas à escolha entre listas e vetores se poderia sugerir. Embora seja verdade que o vetor seja capaz de crescer dinamicamente, a dinâmica de seu crescimento envolve muita cópia de dados. Claro que a cópia de um milhão de grandes estruturas de dados toma um tempo considerável, mesmo em computadores rápidos. Por outro lado, inserir um grande número de elementos numa lista não requer a cópia de dados não envolvidos. Inserir um novo elemento numa lista requer somente o malabarismo com alguns ponteiros. Na figura 8 isto é mostrado: Um novo elemento é inserido entre o segundo e o terceiro elementos, criando uma nova lista de quatro elementos

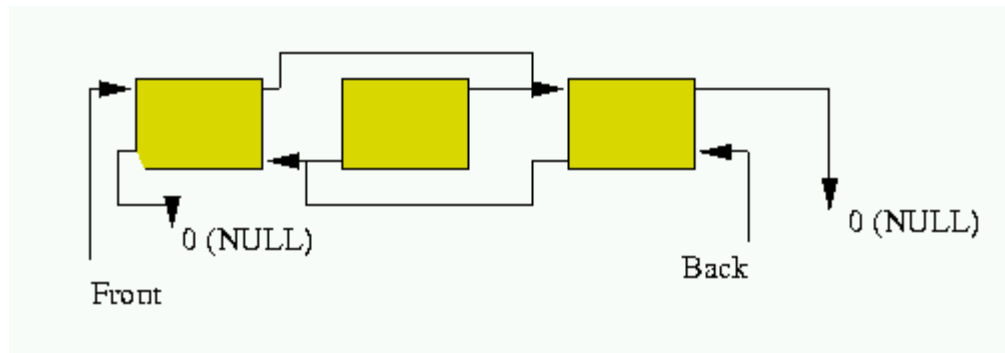
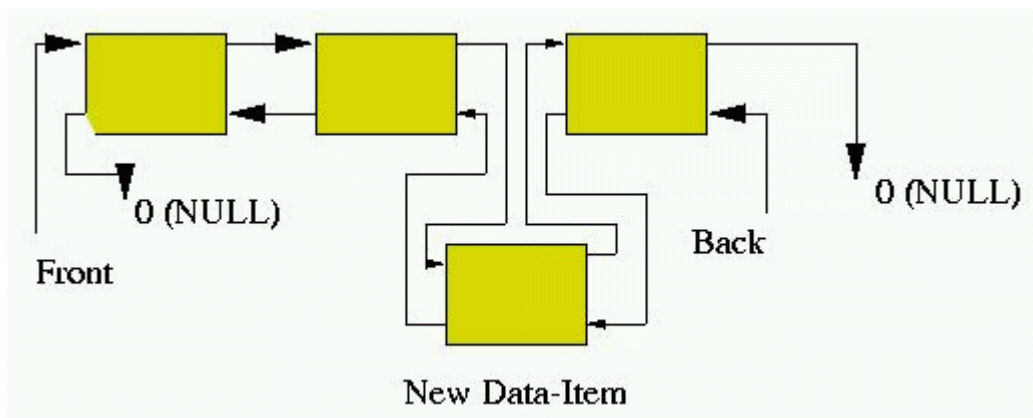


Figura 8 (Adicionando um novo elemento à lista)

Remover um elemento de uma lista também é simples. Partindo outra vez da situação mostrada na figura 7, a figura 9 mostra o que acontece se o elemento 2 é removido da lista. Outra vez: Somente malabarismos com os apontadores. Neste caso é ainda mais simples que na inserção: Só dois ponteiros necessitam ser re-avaliados.



Sumarizando, a comparação entre listas e vetores é provável que o melhor é concluir que não há uma linha clara de corte à questão entre qual dos dois preferir. Existem regras práticas, às que podemos aderir. Mas se ocorrer o pior temos que descobrir o que é melhor.

De qualquer maneira o recipiente lista está disponível, assim, vejamos que fazer com ele. Os seguintes construtores, operadores e funções membro estão disponíveis:

Construtores:

- Uma lista pode ser construída vazia:

```
list<string> object;
```

Como com os vetores produz um erro a referência a uma lista vazia:

- Uma lista pode ser iniciada com um certo número de elementos. Como padrão, se o valor de iniciação não é explicitado, o valor padrão ou construtor padrão para o tipo de dados será usado. Pro exemplo:

```
list<string> object(5, string("Alô")); // inicia com 5 Alô's
list<string> container(10);           // e 10 `strings` vazias
```

- Uma lista pode ser iniciada usando dois iteradores. Para iniciar uma lista com os elementos de 5 a 10 (incluindo o último) de um 'vector<string>' a seguinte construção é usada:

```
extern vector<string> container;
list<string> object(&container[5], &container[11]);
```

- Uma lista pode ser iniciada usando-se um construtor de cópias:

```
extern list<string> container;
list<string> object(container);
```

- Não existem operadores especiais para listas além dos operadores padrão para recipientes;

As seguintes funções membro estão disponíveis para as listas:

- 'Tipo &list::back()': Retorna uma referência ao último elemento da lista. É responsabilidade do programador usar este membro só se a lista não estiver vazia;
- 'list::iterator list::begin()': Este membro retorna um iterador que aponta ao primeiro elemento da lista, retornando 'list::end()' se a lista estiver vazia;

- `'list::clear()'`: Elimina todos os elementos da lista.
- `'bool list::empty()'`: Retorna verdadeiro se a lista não contém elementos;
- `'list::iterator list::end()'`: Retorna um iterador que aponta além do último elemento da lista;
- `'list::iterator list::erase()'`: Usado para eliminar um rango específico de elementos da lista;
 - `'erase(pos)'`: Elimina o elemento apontado por 'pos'. O iterador ++pos é retornado;
 - `'erase(first, beyond)'`: Elimina os elementos indicados pelos iteradores '[first, beyond]'. Beyond é retornado;
- `'Tipo &list::front()'`: Retorna uma referência ao primeiro elemento da lista. É responsabilidade do programador usar este membro só se a lista não esteja vazia.
- `'... list::insert()'`: Insere elementos numa lista. O valor depende da versão de 'insert()' chamada:
 - `'list::iterator insert(pos)'`: Insere um valor padrão do tipo 'Tipo' na posição 'pos' e retorna 'pos';
 - `'list::iterator insert(pos, value)'`: insere 'value' em 'pos' e retorna 'pos';
 - `'void insert(pos, first, beyond)'`: insere o rango de elementos do iterador '[first, beyond]';
 - `'void insert(pos, n, value)'`: insere n elementos com valor 'value' na posição 'pos';
- `'void list<Tipo>::merge(list<Tipo> outra)'`: Esta função membro assume que as listas atual e 'outra' estão ordenadas (veja abaixo o membro 'sort()'), e inserirá, baseado no assumido, os elementos da 'outra' na atual de tal maneira que a lista permanece ordenada. Se ambas listas não estão ordenadas, a resultante estará ordenada 'tanto quanto possível', dada a ordem inicial dos elementos nas duas listas. O membro 'list<Tipo>::merge()' usa o operador 'Tipo::operator<()'' para ordenar os dados na lista, que deve estar disponível. O seguinte exemplo ilustra o uso de 'merge()': O 'objeto' lista não é ordenado, assim resulta ordenado 'tanto quanto possível':

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

void showlist(list<string> &target)
{
```

```

        for
        (
            list<string>::iterator from = target.begin();
            from != target.end();
            ++from
        )
            cout << *from << " ";

        cout << endl;
    }

int main()
{
    list<string> first;
    list<string> second;

    first.push_back(string("alpha"));
    first.push_back(string("bravo"));
    first.push_back(string("golf"));
    first.push_back(string("quebec"));

    second.push_back(string("oscar"));
    second.push_back(string("mike"));
    second.push_back(string("november"));
    second.push_back(string("zulu"));

    first.merge(second);
    showlist(first);
}

```

Uma sutileza aqui é que 'merge()' não altera a lista se a lista é usada como argumento: 'object.merge(object)' não muda a lista 'objeto'.

- 'void list::pop_back()': Remove o último elemento da lista. Com uma lista vazia não ocorre nada;
- 'void list::pop_front()': Remove o primeiro elemento da lista. Com uma lista vazia nada acontece;
- 'void list::push_back(value)': Agrega um valor ao fim da lista;
- 'void list::push_front(value)': Agrega um valor antes do primeiro elemento da lista;
- 'void list::resize()': Usado para alterar o número de elementos da lista;

- 'resize(n, value)': Usado para dar à lista o tamanho de n elementos. O valor é opcional. Se a lista for expandida e 'value' não é dado, os elementos extra são iniciados com o valor padrão do tipo de dados, se for dado os elementos extra serão iniciados com 'value';
- list::reverse_iterator list::rbegin(): Retorna um iterador que aponta para o último elemento da lista;
- 'void list::remove(value)': Remove todas as ocorrências de 'value' da lista. No seguinte exemplo as duas 'strings' 'Alô' são removidas da lista:

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
    list<string> object;

    object.push_back(string("Alô"));
    object.push_back(string("Mundo"));
    object.push_back(string("Alô"));
    object.push_back(string("Mundo"));

    object.remove(string("Alô"));

    while (object.size())
    {
        cout << object.front() << endl;
        object.pop_front();
    }
}
/*
    Saída Gerada:
    Mundo
    Mundo
*/
```

- 'list::reverse_iterator list::rend()': Retorna um iterador que aponta para antes do primeiro elemento da lista;
- 'size_t list::size()': Retorna o número de elementos da lista;
- 'void list::reverse()': Inverte a ordem dos elementos da lista. O último elemento fica o primeiro e vice versa;

- 'void list::sort()': Ordena a lista. Um exemplo de seu uso é dado abaixo, na descrição da função membro 'unique()'. A função 'list<Tipo>::sort()' usa 'Tipo::operator<()' para ordenar os dados da lista, o qual deve estar disponível;
- 'void list::splice(pos, object)': Transfere o conteúdo de 'object' à lista corrente, começando a inserção na posição do iterador 'pos' do objeto. Em seguida o objeto de 'splice()' estará vazio. Por exemplo:

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
    list<string> object;

    object.push_front(string("Alô"));
    object.push_back(string("Mundo"));

    list<string> argument(object);

    object.splice(++object.begin(), argument);

    cout << "O objeto contém " << object.size() << " elementos, " <<
        "O argumento contém " << argument.size() <<
        " elementos," << endl;

    while (object.size())
    {
        cout << object.front() << endl;
        object.pop_front();
    }
}
```

Alternativamente o argumento pode ser seguido por um iterador do argumento, indicando o primeiro argumento a ser transferido, ou por dois iteradores início e fim, definindo o rango da iteração '[begin, end]' que transferirá ao objeto;

- 'void list::swap()': Usado para a troca entre duas listas de tipos de dados idênticos;
- 'void list::unique()': opera sobre uma lista ordenada, remove todos os elementos consecutivos idênticos de uma lista. A função '<Tipo>::unique()' usa o operador 'Tipo::operator=='() para identificar elementos idênticos, o qual deve estar disponível. Eis um exemplo de remoção de elementos idênticos de uma lista:

```

#include <iostream>
#include <string>
#include <list>
using namespace std;

// veja o exemplo de merge()
void showlist(list<string> &target);
void showlist(list<string> &target)
{
    for
    (
        list<string>::iterator from = target.begin();
        from != target.end();
        ++from
    )
        cout << *from << " ";

    cout << endl;
}

int main()
{
    string
        array[] =
        {
            "charley",
            "alpha",
            "bravo",
            "alpha"
        };

    list<string>
        target
        (
            array, array + sizeof(array)
            / sizeof(string)
        );

    cout << "Inicialmente temos: " << endl;
    showlist(target);

    target.sort();
    cout << "Depois de sort() temos: " << endl;
    showlist(target);

    target.unique();
    cout << "Depois de unique() temos: " << endl;
}

```

```

        showlist(target);
    }
    /*
    Saída Gerada:

    Inicialmente temos:
    charley alpha bravo alpha
    Depois de sort() temos:
    alpha alpha bravo charley
    Depois de unique() temos:
    alpha bravo charley
    */

```

12.3.3: O recipiente 'queue' (fila)

A classe 'queue' implanta uma estrutura de dados em fila. Antes do uso da classe é necessária a diretiva ao pré-processador:

```
#include <queue>
```

Na figura 10 está representada graficamente uma fila.

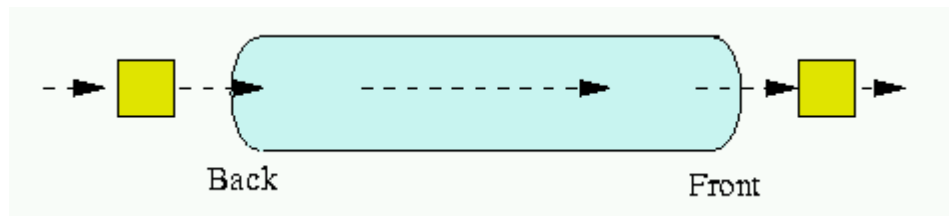


Figura 10 (A queue data-structure)

Na figura 10 se mostra que uma fila tem um ponto (a dianteira) onde se pode juntar itens à fila e um ponto (a traseira) de onde se pode retirar itens (ler) da fila. Esta fila é também chamada uma estrutura de dados FIFO (First In, First Out). É a mais freqüentemente usada em situações onde os eventos podem ser manipulados na mesma ordem que são gerados.

Os seguintes construtores, operadores e funções membro estão disponíveis para o recipiente 'queue':

Construtores:

- Uma fila pode ser construída vazia:

```
queue<string> object;
```

Como com o vetor, produz um erro a referência a um elemento de uma fila vazia;

- Uma fila pode ser iniciada usando-se um construtor de cópia:

```
extern queue<string> container;
queue<string> object(container);
```

O recipiente fila só comporta os operadores básicos para recipientes;

As seguintes funções membro estão disponíveis para as filas:

- 'Tipo &queue::back()': Retorna o último elemento da fila. É responsabilidade do programador usar o membro só se a fila não está vazia;
- 'bool queue::empty()': Retorna verdadeiro se a fila está vazia;
- 'Tipo &queue::front()': Retorna uma referência do primeiro elemento da fila. É responsabilidade do programador usar a função só se a fila não está vazia;
- 'void queue::push(value)': Adiciona 'value' ao fim da fila;
- 'void queue::pop()': Remove o primeiro elemento da fila. Note que o elemento não é retornado pela função. Nada acontece se o membro é chamado para uma fila vazia. Pode parecer estranho que 'pop()' não retorne o valor 'Tipo' (como 'front()'). Devido a isto, é imperativo usar primeiro 'front()' e então 'pop()' para examinar e remover o elemento da frente da fila. Contudo há uma boa razão para este comportamento. Se 'pop()' retornasse o primeiro elemento, teria que retorná-lo como valor em lugar de referência, já que seria um ponteiro perigoso, pois 'pop()' remove o elemento apontado. O retorno em valor é ineficiente neste caso: Envolve pelo menos um construtor de cópia. Como é impossível a 'pop()' retornar um valor correta e eficientemente é mais sensível 'pop()' não retornar nada e requerer aos clientes que usem 'front()' para inspecionar o valor do início da fila;
- 'size_t queue::size()': Retorna o número de elementos na fila;

Note que a fila não suporta iteradores ou operadores subscritos. Os únicos elementos acessíveis são o primeiro e o último. Uma fila pode ser esvaziada por:

- Remoção repetida de seu primeiro elemento;

- Adjudicação de uma fila vazia com o mesmo tipo de dados;
- Chamar seu destrutor.

12.3.4: O Recipiente `priority_queue`

A classe 'priority_queue' implanta uma estrutura de dados de fila com prioridade. Antes do uso da classe é necessária a referência à diretiva ao pré-processador:

```
#include <queue>
```

Uma fila com prioridade é idêntica a uma fila, mas permite a entrada de dados de acordo com regras de prioridades. Um exemplo de uma situação onde a fila com prioridade é encontrada na vida real encontramos nos terminais de ingresso nos aeroportos. Os passageiros normalmente esperam em fila seu turno, mas passageiros atrasados saltam a fila: Recebem uma prioridade maior que outros passageiros.

A fila com prioridade usa o operador 'operator<()' do tipo de dado guardado na fila com prioridade para decidir sobre a prioridade dos elementos de dados. Assim, a fila com prioridade pode ser usada para ordenar valores segundo sua chegada. Um exemplo simples de tal fila com prioridade é o seguinte programa: Lê palavras de 'cin' e escreve numa lista ordenada de palavras em 'cout':

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main()
{
    priority_queue<string> q;
    string word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        cout << q.top() << endl;
        q.pop();
    }
}
```

Infelizmente as palavras estão listadas na ordem inversa: Porque para o operador '<-operator'

as palavras aparecem depois que a seqüência ASCII apareça na fila com prioridade. Uma solução a esse problema é definir uma classe de encapsulamento para o tipo de dados da 'string', onde o operador 'operator<()' foi definido de acordo ao desejado, i.e., assegurando-se de que as palavras que aparecem antes na seqüência ASCII apareçam primeiro na fila. Eis o programa modificado:

```
#include <iostream>
#include <string>
#include <queue>

class Text
{
    std::string d_s;

public:
    Text(std::string const &str)
    :
        d_s(str)
    {}
    operator std::string const &() const
    {
        return d_s;
    }
    bool operator<(Text const &right) const
    {
        return d_s > right.d_s;
    }
};

using namespace std;

int main()
{
    priority_queue<Text> q;
    string word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        word = q.top();
        cout << word << endl;
        q.pop();
    }
}
```

No programa acima a classe de encapsulamento define o operador 'operator<()' de outra

maneira que a classe 'string', resultando na ordem preferencial. Outra possibilidade seria guardar o conteúdo da fila com prioridade em, p.ex., num vetor, a partir do qual os elementos podem ser lidos em ordem inversa.

Os seguintes construtores, operadores e funções membro estão disponíveis para os recipientes de fila com prioridade:

Construtores:

- Uma fila com prioridade pode ser construída vazia:

```
priority_queue<string> object;
```

Como com os vetores, provoca um erro referir-se a um elemento de uma fila com prioridade vazia.

- Uma fila com prioridade pode ser iniciada com um construtor de cópia:

```
extern priority_queue<string> container;  
priority_queue<string> object(container);
```

A fila com prioridade só suporta os operadores básicos dos recipientes.

As seguintes funções membro estão disponíveis para as filas com prioridade:

- 'bool priority_queue::empty()': Retorna verdadeiro se a fila está vazia;
- 'void priority_queue::push(value)': Insere um valor na posição apropriada na fila;
- 'void priority_queue::pop()': Remove o elemento do topo da fila com prioridade. Note que o elemento não é retornado. Nada acontece se chamada para uma fila vazia. Veja seção 12.3.3 sobre as razões de porque 'pop()' retorna void;
- 'size_t priority_queue::size()': Retorna o número de elementos na fila com prioridade;
- 'Tipo &priority_queue::top()': Retorna uma referência ao primeiro elemento da fila. É responsabilidade do programador usar este membro só se a fila não estiver vazia;

Note que a fila com prioridade não suporta iteradores ou um operador de subscrição. O único elemento que se pode acessar é o do topo da fila. Uma fila com prioridade pode ser esvaziada por:

- Removendo repetidamente o elemento do topo;

- Adjudicando uma fila vazia usando o mesmo tipo de dados;
- Chamando seu destrutor.

12.3.5: O recipiente `deque`

A classe 'deque' (se pronuncia: `dec`) implanta uma estrutura de dados em fila de duplo fim (estrutura `deque`). Antes de usar este recipiente é necessária a seguinte diretiva ao pré-processador:

```
#include <deque>
```

Um 'deque' é comparável a uma fila, mas permite leitura e escritura nos dois extremos. um 'deque' suporta muitas mais funcionalidades que uma fila, como ficará claro de suas funções membro. Um 'deque' é a combinação de um vetor e duas filas, operando em ambos extremos do vetor. Em situações onde inserções aleatórias e adições e/ou remoções de elementos em um ou dois extremos do vetor ocorrem com frequência, o uso de 'deque' deve ser considerado.

'deque' possui os seguintes construtores, operadores e funções membro:

Construtores:

- O 'deque' pode ser construído vazio:

```
deque<string> object;
```

Como com os vetores, produz um erro a referência a um elemento de um 'deque' vazio.

- Um 'deque' pode ser iniciado com um certo número de elementos. Se na iniciação não é explicitado um valor, o valor padrão ou o construtor padrão para o tipo de dado é usado. Por exemplo:

```
deque<string> object(5, string("Alô")), // inicia com 5 Alôs
deque<string> container(10);           // e com 10 strings vazias
```

- Um 'deque' pode ser iniciado usando-se dois iteradores. Para se iniciar um 'deque' com os elementos de 5 a 10 (incluindo o último) de um 'vector<string>', usa-se a seguinte construção:

```
extern vector<string> container;
deque<string> object(&container[5], &container[11]);
```

- Um 'deque' pode ser iniciado usando-se um construtor de cópia:

```
extern deque<string> container;
deque<string> object(container);
```

- Além dos operadores estandartes dos recipientes, o 'deque' suporta o operador indexação, que é usado para retirar ou re-adjudicar elementos aleatoriamente. Note que os elementos indexados devem existir.

As seguintes funções membro estão disponíveis para o 'deque':

- 'Tipo &deque::back()': Retorna uma referência ao último elemento do 'deque'. É responsabilidade do programador só usar a função se o 'deque' não estiver vazio;
- 'deque::iterator deque::begin()': Retorna um iterador que aponta para o primeiro elemento do 'deque';
- 'void deque::clear()': Elimina todos os elementos do 'deque';
- 'bool deque::empty()': Retorna verdadeiro se o 'deque' estiver vazio.
- 'deque::iterator deque::end()': Retorna um iterador que aponta para além do último elemento do 'deque';
- 'deque::iterator deque::erase()': Usado para eliminar um rango determinado de elementos do 'deque':
 - 'erase(pos)': Elimina o elemento apontado por 'pos';
 - 'erase(first, beyond)': Elimina os elementos indicados pelo iterador '[first, beyond]'. Retorna 'beyond'.
- 'Tipo &deque::front()': Retorna uma referência ao primeiro elemento do 'deque'. É responsabilidade do programador não usar a função em 'deque' vazio;
- '... deque::insert()' Insere elementos a partir de uma certa posição. O retorno depende da versão de 'insert()' usada:
 - 'deque::iterator insert(pos)': Insere um valor padrão do tipo em 'pos', retorna 'pos';
 - 'deque::iterator insert(pos, value)': Insere 'value' em 'pos', retorna 'pos';
 - 'void insert(pos, first, beyond)': Insere os elementos no rango do iterador '[first, beyond)';
 - 'void insert(pos, n, value)': Insere n elementos com valor 'value' começando da posição

'pos'.

- `'void deque::pop_back()'`: Remove o último elemento do 'deque'. Com um 'deque' vazio nada acontece;
- `'void deque::pop_front()'`: Remove o primeiro elemento do 'deque'. Com um 'deque' vazio nada acontece;
- `'void deque::push_back(value)'`: Agrega 'value' no fim do 'deque';
- `'void deque::push_front(value)'`: Agrega 'value' antes do primeiro elemento do 'deque';
- `'void deque::resize()'`: Altera o número de elementos do 'deque';
- `'resize(n, value)'`: Re-dimensiona o 'deque' para n elementos. O valor 'value' é opcional. Se o 'deque' é expandido e 'value' não é mencionado, os elementos agregados são iniciados com o valor padrão do tipo, do contrário é usado 'value'.
- `'deque::reverse_iterator deque::rbegin()'`: Retorna um iterador que aponta para o último elemento do 'deque';
- `'deque::reverse_iterator deque::rend()'`: Retorna um iterador que aponta para antes do primeiro elemento do 'deque';
- `'size_t deque::size()'`: Retorna o número de elementos do 'deque';
- `'void deque::swap(argument)'`: Troca os elementos de dois 'deques' com mesmo tipo de dados.

12.3.6: O recipiente 'map'

A classe 'map' implanta um conjunto (ordenado) associativo. Para se usar os recipientes 'map' a seguinte diretiva ao pré-processador tem que ser especificada:

```
#include <map>
```

Um 'map' é preenchido com pares chave/valor, que podem ser de qualquer tipo aceitável pelos recipientes. Como os tipos estão associados a ambos, chaves e valores, é necessário especificar-se dois tipos dentro de parênteses quadrados, comparáveis às especificações vistas com recipientes de pares (seção 12.2). O primeiro tipo corresponde ao tipo da chave e o segundo ao valor. Por exemplo, um mapa onde uma chave é uma 'string' e o valor um 'double' pode ser definido assim:

```
map<string, double> object
```

A chave serve para acessar sua informação associada. Essa informação é chamada valor. Por exemplo, uma lista telefônica usa o número telefônico e outra informação (p.ex., código postal, endereço, profissão) como valor. Como um mapa ordena suas chaves, o operador '`operator<()`' precisa estar definido e sensível ao seu uso. Por exemplo, geralmente é uma má idéia usar ponteiros para as chaves, já que ordenar ponteiros é algo diferente que ordenar os valores para os quais apontam.

As duas operações fundamentais sobre os mapas são sua armazenagem e recuperação. O operador de indexação usando a chave como índice pode ser usado para ambos. Se o operador de indexação é usado como '`lvalue`', se realizará inserções. Se for usado como '`rvalue`' o valor da chave associada será recuperado. Cada chave só pode figurar uma vez num mapa. Se a mesma chave for entrada outra vez, o novo valor substituirá o anterior, que é perdido.

Uma combinação de chave/valor pode ser inserida implícita ou explicitamente num mapa. Se for requerida uma inserção explícita, a combinação chave/valor tem que ser construída antes. Para isto, todo mapa define um tipo de valor que se usa para criar valores que podem ser guardados no mapa. Por exemplo, um valor para um '`map<string, int>`' é construído como segue:

```
map<string, int>::value_type siValue("Alô", 1);
```

O tipo de valor é associado com o '`map<string, int>`': O tipo da chave é '`string`', o tipo do valor é '`int`'. Objetos anônimos com tipo de valor são freqüentemente usados. Pex.:

```
map<string, int>::value_type("Hello", 1);
```

Em vez de se usar a linha '`map<string, int>::value_type(...)`' uma e outra vez, a definição de um tipo ('`typedef`') é usada para reduzir a digitação e aumentar a legibilidade:

```
typedef map<string, int>::value_type StringIntValue
```

Usando-se essa definição a construção fica:

```
StringIntValue("Hello", 1);
```

Finalmente, usa-se pares para representar combinações chave/valor usadas pelos mapas

```
pair<string, int>("Hello", 1);
```

Os seguintes construtores, operadores e funções membro pertencem à classe de recipientes '`map`':

Construtores:

- Um mapa pode ser construído vazio:

```
map<string, int> object;
```

Note que os valores guardados nos mapas podem ser recipientes. Por exemplo, o seguinte define um mapa onde o valor é um 'pair': Um recipiente aninhado em outro recipiente:

```
map<string, pair<string, string> > object;
```

Note o espaço entre o parêntese quadrado fechado '>': É obrigatório, já que a concatenação imediata de dois parênteses quadrados seria interpretada pelo compilador como o operador de deslocamento à direita ('operator>>()'), que não cabe aqui.

- Um mapa pode ser iniciado usando dois iteradores. Os iteradores podem apontar a tipos de valores para a construção do mapa ou a pares de objetos (veja seção 12.2). Se são usados pares, seus primeiros elementos representam a chave e seus segundos elementos os valores. Por exemplo:

```
pair<string, int> pa[] =
{
    pair<string, int>("um", 1),
    pair<string, int>("dois", 2),
    pair<string, int>("três", 3),
};

map<string, int> object(&pa[0], &pa[3]);
```

Neste exemplo 'map<string, int>::value_type' poderia ter sido escrito 'pair<string, int>' também.

Quando 'begin' é o primeiro iterador usado para construir um mapa e 'end' o segundo iterador, '[begin, end)' será usado para iniciar o mapa. Pode ser contrário à intuição, o construtor do mapa só entrará novas chaves. Se o último elemento de 'pa' fosse "um", 3 só dois elementos teriam entrado no mapa: "um", 1 e "dois", 2. O valor "um", 3 teria sido ignorado silenciosamente.

O mapa recebe suas próprias cópias dos dados aos quais os iteradores apontam. Isto está ilustrado no seguinte exemplo:

```
#include <iostream>
#include <map>
using namespace std;

class MinhaClasse
{
public:
    MinhaClasse()
    {
        cout << "Construtor de MinhaClasse\n";
    }
};
```



```

    }
    MinhaClasse(const MinhaClasse &outro)
    {
        cout << "Construtor de cópia de MinhaClasse\n";
    }
    ~MinhaClasse()
    {
        cout << "Destrutor de MinhaClasse\n";
    }
};

int main()
{
    pair<string, MinhaClasse> pairs[] =
    {
        pair<string, MyClass>("um", MinhaClasse()),
    };
    cout << "pares construídos\n";

    map<string, MinhaClasse> mapsm(&pairs[0], &pairs[1]);
    cout << "mapsm construídos\n";
}
/*
    Saída Gerada:
    Construtor de MinhaClasse
    Construtor de cópia de MinhaClasse
    Construtor de MinhaClasse
    pares construídos
    Construtor de cópia de MinhaClasse
    Construtor de cópia de MinhaClasse
    Destrutor de MinhaClasse
    mapsm construídos
    Destrutor de MinhaClasse
*/

```

- Ao percorrer a saída deste programa vemos, primeiro, o construtor de um objeto de 'MinhaClasse' é chamado para iniciar um elemento anônimo do conjunto de pares. Este objeto é então copiado no primeiro elemento do conjunto de pares pelo construtor de cópia. Em seguida, o elemento original já não é necessário e destruído. Aqui o conjunto de pares está construído. Nesse ponto o mapa constroi um objeto par temporário, que é usado para construir o elemento do mapa. Tendo construído o elemento do mapa, o par temporário é destruído. Eventualmente quando o programa termina, o par de elementos guardados no mapa é destruído também.
- Um mapa pode ser iniciado usando-se um construtor de cópia:

```
extern map<string, int> container;
```

```
map<string, int> object(container);
```

- Além dos operadores estandartes dos recipientes, o mapa suporta o operador de indexação, que pode ser usado para retirar ou re-adjudicar elementos individuais do mapa. Aqui o argumento do índice é uma chave. Se a chave não se encontra no mapa um novo elemento de dado é automaticamente agregado ao mapa, usando o valor padrão ou o construtor padrão para iniciar a parte do valor do novo elemento. Este valor padrão é retornado se a indexação use um 'rvalue'.

Quando de inicie um novo ou re-adjudique um elemento do mapa, o tipo do lado direito da adjudicação deve ser igual ao (ou promovível ao) tipo da parte do valor do mapa. P.ex., para acrescentar ou mudar o valor do elemento “dois” do mapa, o seguinte comando é usado:

```
mapsm["dois"] = MinhaClasse();
```

A classe 'map' possui as seguintes funções membro:

- 'map::iterator map::begin()': Retorna um iterador que aponta para o primeiro elemento do mapa;
- 'map::clear()': Elimina todos os elementos do mapa;
- 'size_t map::count(key)': Retorna 1 se a chave provida está disponível, do contrário retorna zero (0);
- 'bool map::empty()': Retorna verdadeiro se o mapa está vazio;
- 'map::iterator map::end()': Retorna um iterador que aponta para depois do último elemento do mapa;
- 'pair<map::iteratod, map::iterator> map::equal_range(key)': Retorna um par de iteradores, sendo respectivamente o valor de retorno das funções 'lower_bound()' e 'upper_bound()', introduzidas abaixo. Um exemplo ilustra estas funções membro na discussão da função 'upper_bound()';
- '... map::erase()': Elimina um elemento específico ou um rango de elementos do mapa:
 - 'bool erase(key)': Elimina o elemento com a chave dada do mapa. Retorna verdadeiro se o valor for removido, falso se o mapa não contém a chave;
 - 'void erase(pos)': Elimina o elemento apontado pelo iterador 'pos';

- 'void erase(first, beyond)': Elimina todos os elementos entre 'first' e 'beyond'.
- 'map::iterator map::find(key)': Retorna um iterador que aponta para o elemento com a chave dada. Se não existir retorna 'end()'. O seguinte exemplo ilustra o uso da função 'find()':

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<string, int> object;

    object["um"] = 1;

    map<string, int>::iterator it = object.find("one");

    cout << "`um` " <<
        (it == object.end() ? "não " : "") << "encontrado\n";

    it = object.find("três");

    cout << "`três` " <<
        (it == object.end() ? "não " : "") << "encontrado\n";
}
/*
    Saída Gerada:
    `um` encontrado
    `três` não encontrado
*/
```

- '... map::insert()': Usado para inserir elementos no mapa. Contudo não substitui os valores associados a chaves já existentes. Retorna um valor dependendo da versão de 'insert()' chamada:

- 'pair<map::iterator, bool> insert(keyvalue)': Insere um novo 'map::value_type' no mapa. O valor retornado é um 'pair<map::iterator, bool>'. Se retorna verdadeiro, 'keyvalue' foi inserido no mapa. O valor falso indica que a chave indicada em 'keyvalue' já existia no mapa e assim não foi inserida. Em ambos casos o campo 'map::iterator' aponta ao elemento de dados com chave 'keyvalue'. O uso desta variante de 'insert()' está ilustrado no exemplo abaixo:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
```

```

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("um", 10),
        pair<string,int>("dois", 20),
        pair<string,int>("três", 30),
    };
    map<string, int> object(&pa[0], &pa[3]);

        // {quatro, 40} e `verdadeiro` é retornado
    pair<map<string, int>::iterator, bool>
        ret = object.insert
            (
                map<string, int>::value_type
                ("quatro", 40)
            );

    cout << boolalpha;

    cout << ret.first->first << " " <<
        ret.first->second << " " <<
        ret.second << " " << object["quatro"] << endl;

        // {quatro, 40} and `falso` é retornado
    ret = object.insert
        (
            map<string, int>::value_type
            ("quatro", 0)
        );

    cout << ret.first->first << " " <<
        ret.first->second << " " <<
        ret.second << " " << object["quatro"] << endl;
}
/*
Saída Gerada:

quatro 40 verdadeiro 40
quatro 40 falso 40
*/

```

Note as construções algo peculiar como:

```
cout << ret.first->first << " " << ret.first->second << ...
```

Realizar esse 'ret' é igual ao par retornado pela função membro 'insert()'. Seu campo 'first' é um iterador em 'map<string, int>', assim pode ser considerada um ponteiro a

'map<string, int>::value_type'. Esses valores são pares também, tendo os campos 'first' e 'second'. Conseqüentemente 'ret.first->first' é a chave do valor do mapa (uma `string`) e 'ret.first->second' é o valor (um 'int');

- 'map::iterator insert(pos, keyvalue)': Assim um 'map::value_type' pode ser inserido num mapa. 'pos' é ignorado e é retornado um iterador ao elemento inserido;
- 'void insert(first, second)': Insere o elemento ('map::value_type')
- 'map::iterator map::lower_bound(key)': Retorna um iterador que aponta para o valor de chave do primeiro elemento que tem chave igual à especificada. Se não existir tal elemento a função retorna 'map::end()';
- 'map::reverse_iterator map::rbegin()': Retorna um iterador que aponta para o último elemento do mapa;
- 'map::reverse_iterator map::rend()': Retorna um iterador que aponta para antes do primeiro elemento do mapa;
- 'size_t map::size()': Retorna o número de elementos do mapa.
- 'void map::swap(argument)': Usado para trocar os dados de dois mapas, com tipos de chave/valor idênticos;
- 'map::iterator map::upper_bound(key)': Retorna um iterador que aponta para o primeiro elemento com valor de chave que exceda o especificado. Se não existir tal elemento, a função retorna 'map::end()'. O seguinte exemplo ilustra as funções membro 'equal_range()', 'lower_bound()' e 'upper_bound()':

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string, int>("um", 10),
        pair<string, int>("dois", 20),
        pair<string, int>("três", 30),
    };
    map<string, int> object(&pa[0], &pa[3]);
```

```

map<string, int>::iterator it;

if ((it = object.lower_bound("tw")) != object.end())
    cout << "existe limite inferior `tw`, é: " <<
        it->first << endl;

if (object.lower_bound("twoo") == object.end())
    cout << "limite inferior `twoo` não existe" << endl;

cout << "limite inferior dois: " <<
    object.lower_bound("dois")->first <<
    " existe\n";

if ((it = object.upper_bound("tw")) != object.end())
    cout << "limite superior `tw` existe, é: " <<
        it->first << endl;

if (object.upper_bound("twoo") == object.end())
    cout << "limite superior `twoo` não existe" << endl;

if (object.upper_bound("two") == object.end())
    cout << "limite superior `two` não existe" << endl;

pair
<
    map<string, int>::iterator,
    map<string, int>::iterator
>
p = object.equal_range("dois");

cout << "igual: `first` aponta para " <<
    p.first->first << ", `second` é " <<
    (
        p.second == object.end() ?
            "não existe"
        :
            p.second->first
    ) <<
    endl;
}
/*

```

Saída Gerada:

```

limite inferior `tw` existe, é: dois
limite inferior `twoo` não existe
limite inferior dois: dois existe
limite superior `tw` existe, é: dois

```

```

    limite superior `twoo` não existe
    limite superior `dois` não existe
    igual: `first` aponta para dois, `second` não existe
*/

```

Como foi mencionado no início desta seção, o mapa representa uma associação de conjuntos ordenada. Num mapa as chaves estão ordenadas. Se uma aplicação necessita percorrer todos os elementos de um mapa (ou só as chaves ou valores) os iteradores 'begin()' e 'end()' devem ser usados. O seguinte exemplo mostra como fazer uma tabela simples que liste todas as chaves e valores do mapa:

```

#include <iostream>
#include <iomanip>
#include <map>

using namespace std;

int main()
{
    pair<string, int>
    pa[] =
    {
        pair<string,int>("um", 10),
        pair<string,int>("dois", 20),
        pair<string,int>("três", 30),
    };
    map<string, int>
    object(&pa[0], &pa[3]);

    for
    (
        map<string, int>::iterator it = object.begin();
        it != object.end();
        ++it
    )
        cout << setw(5) << it->first.c_str() <<
            setw(5) << it->second << endl;
}
/*
    Saída Gerada:
    um      10
    três    30
    dois    20
*/

```

12.3.7: O Recipiente 'multimap'

Como o mapa, a classe multimap implanta um conjunto associativo (ordenado). Antes de se poder usar recipientes multimap é necessário especificar a diretiva ao pré-processador:

```
#include <map>
```

A diferença principal entre mapa e multimap é que o multimap suporta múltiplos valores associados à mesma chave. Note que o multimap aceita também múltiplas chaves com valores idênticos associados a chaves idênticas.

O mapa e o multimap possuem o mesmo conjunto de funções membro, com exceção do operador indexação ('operator[]()'), que o multimap não suporta. Isto é compreensível: Se múltiplas entradas com a mesma chave são permitidas, qual seriam os possíveis valores retornados por 'object[key]'?

Para uma visão das funções membro de multimap refira-se à seção 12.3.6. Contudo, algumas funções membro merecem uma atenção especial quando usadas no contexto do recipiente multimap. Estes membros são discutidos abaixo:

- 'size_t map::count(key)': Retorna o número de entradas no multimap associadas à chave dada;
- '... multimap::erase()': Usada para eliminar elementos do mapa:
 - 'size_t erase(key)': Elimina todos os elementos com a chave dada. O número de elementos eliminados é retornado;
 - 'void erase(pos)': Elimina um simples elemento apontado por 'pos'. Outros elementos possivelmente com a mesma chave não são eliminados;
 - 'void erase(first, beyond)': Elimina todos os elementos indicados pelo iterador de limites '[first, beyond)';
- 'pair<multimap::iterator, multimap::iterator> multimap::equal_range(key)': Retorna um par de iteradores, respectivamente o valor de retorno de 'multimap::lower_bound()' e multimap::upper_bound(), vistos mais abaixo. A função entrega um meio simples de determinar todos os elementos do multimap com a mesma chave. Um exemplo, no fim da seção, ilustra o uso destas funções membro;
- 'multimap::iterator multimap::find(key)': Retorna um iterador apontando para o primeiro valor com chave 'key'. Se não há nenhum retorna 'multimap::end()'. O iterador pode ser incrementado

para visitar todos os elementos com mesma chave até que chegue o fim ou encontre um elemento com outra chave.

- 'multimap::iterator multimap::insert()': Esta função como norma é bem sucedida e 'multimap::iterator' é retornado, em lugar de 'pair<multimap::iterator, bool>' como com o recipiente mapa. O iterador retornado aponta para o elemento recém agregado.

Apesar de que as funções 'lower_bound()' e 'upper_bound()' atuem identicamente em mapa como em multimap, sua operação em multimap merece alguma atenção adicional. O seguinte exemplo ilustra a aplicação de multimap::lower_bound(), multimap::upper_bound() e multimap::equal_range:

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("alpha", 1),
        pair<string,int>("bravo", 2),
        pair<string,int>("charley", 3),
        pair<string,int>("bravo", 6), // `bravo' com valores
                                     // desordenados
        pair<string,int>("delta", 5),
        pair<string,int>("bravo", 4),
    };
    multimap<string, int> object(&pa[0], &pa[6]);

    typedef multimap<string, int>::iterator msiIterator;

    msiIterator it = object.lower_bound("brava");

    cout << "Limite inferior de `brava': " <<
         it->first << ", " << it->second << endl;

    it = object.upper_bound("bravu");

    cout << "Limite superior de `bravu': " <<
         it->first << ", " << it->second << endl;

    pair<msiIterator, msiIterator>
        itPair = object.equal_range("bravo");

    cout << "Igualdade de `bravo':\n";
```

```

        for (it = itPair.first; it != itPair.second; ++it)
            cout << it->first << ", " << it->second << endl;
        cout << "Limite superior: " << it->first << ", " << it->second <<
endl;

        cout << "Igualdade de `brav':\n";
        itPair = object.equal_range("brav");
        for (it = itPair.first; it != itPair.second; ++it)
            cout << it->first << ", " << it->second << endl;
        cout << "Limite superior: " << it->first << ", " << it->second <<
endl;
    }
    /*
        Saída Gerada:

        Limite inferior de `brava': bravo, 2
        Limite superior de `bravu': charley, 3
        Igualdade de `bravo':
        bravo, 2
        bravo, 6
        bravo, 4
        Limite superior: charley, 3
        Igualdade de `brav':
        Limite superior: bravo, 2
    */

```

Em particular note as seguintes características:

- 'lower_bound()' e 'upper_bound()' produzem o mesmo resultado para chaves não existentes: Ambas retornam o primeiro elemento que a chave excede a chave dada;

Apesar de que as chaves estão ordenadas no multimapa, os valores para chaves iguais não estão ordenados: São retirados na ordem em que foram entradas.

12.3.8: O Recipiente 'set'

A classe 'set' implanta uma coleção ordenada de valores. Antes de se usar o recipiente 'set' deve-se especificar a diretiva ao pré-processador:

```
#include <set>
```

Um 'set' é preenchido com valores de qualquer tipo aceito pelos recipientes. Cada valor só pode entrar uma vez numa coleção.

Um valor específico para ser inserido numa coleção pode ser criado explicitamente: Cada 'set' define um tipo de valor usado para criar valores que são ordenados na coleção. Por exemplo, um valor para 'set<string>' pode ser construído como segue:

```
set<string>::value_type setValue("Olá");
```

O tipo de valor está associado a 'set<string>'. Objetos com tipo de valor anônimo também são usados. P.ex.:

```
set<string>::value_type("Olá");
```

Freqüentemente se usa um comando 'typedef' para evitar a mesma digitação repetidamente:

```
typedef set<string>::value_type StringSetValue
```

Usando este 'typedef' os valores para 'set<string>' podem ser construídos assim:

```
StringSetValue("Hello");
```

Alternativamente, pode-se usar os valores do tipo de 'set' imediatamente. Nesse caso o valor do tipo Tipo é implicitamente convertido a set<Type>::value_type.

Os seguintes construtores, operadores e funções membro estão disponíveis para o recipiente 'set':

Construtores:

- Uma coleção pode ser construída vazia:

```
set<int> object;
```

Um 'set' pode ser iniciado usando-se 2 iteradores. P.ex.:

```
int intarr[] = {1, 2, 3, 4, 5};  
set<int> object(&intarr[0], &intarr[5]);
```

Note que todos os valores na coleção têm que ser diferentes: Não é possível guardar o mesmo valor repetidamente quando o 'set' é construído. Se o mesmo valor se repetir só o primeiro será entrado, os outros valores serão silenciosamente ignorados.

Como o mapa, a coleção recebe sua própria cópia dos dados que contém.

- Uma coleção pode ser iniciada com um construtor de cópia:

```
extern set<string> container;  
set<string> object(container);
```

A coleção só suporta o conjunto padrão de operadores dos recipientes.

A classe 'set' possui as seguintes funções membro:

- 'set::iterator set::begin()': Retorna um iterador que aponta para o primeiro elemento da coleção. Se a coleção estiver vazia retorna 'set::end()';
- 'set::clear()': Elimina todos os elementos da coleção;
- 'size_t set::count(key)': Retorna 1 se 'key' provista está disponível, senão 0;
- 'bool set::empty()': Retorna verdadeiro se a coleção está vazia.
- 'set::iterator set::end()': Retorna um iterador que aponta para além do último elemento do 'set';
- 'pair<set::iterator, set::iterator> set::equal_range(key)': Retorna um par de iteradores, respectivamente o retorno de 'lower_bound()' e 'upper_bound()', vistos mais abaixo.
- '... set::erase()': Usada para eliminar elementos do 'set':
 - 'bool erase(value)': Elimina o elemento com o valor especificado da coleção. Retorna verdadeiro se o valor foi removido e falso caso não se encontre o valor na coleção;
 - 'void erase(pos)': Elimina o elemento apontado or 'pos';
 - 'void erase(first, beyond) Elimina os elementos da posição 'first' até 'beyond';
- 'set::iterator set::find(value)': Retorna um iterador ao elemento com 'value'. Se não existir retorna 'end()';
- '... set::insert()': Usado para inserir elementos numa coleção. Se o elemento já existir fica intocado e a inserção é ignorada. O retorno depende da versão de 'insert()' chamada:
 - 'pair<set::iterator, bool> insert(keyvalue)': insere um novo 'set::value_type' na coleção. O retorno é um 'pair<set::iterator, bool>'. Se o retorno for verdadeiro, o valor foi inserido. Se falso indica que o valor especificado já existia na coleção. Em ambos casos o campo 'set::iterator' aponta para o elemento com o valor especificado;
 - 'set::iterator insert(pos, keyvalue)': Assim um valor também pode ser inserido na coleção. A posição 'pos' é ignorada e retorna um iterador que aponta para o elemento inserido;

- `'void insert(first, beyond)'`: Insere o conjunto de valores dados pelo iterador `[first, beyond)`;
- `'set::iterator set::lower_bound(key)'`: Retorna um iterador que aponta para o elemento com o valor da chave. Se não existir retorna `'set::end()'`;
- `'set::reverse_iterator set::rbegin()'`: Retorna um iterador que aponta para o último elemento da coleção;
- `'set::reverse set::rend()'`: Retorna um iterador que aponta para antes do primeiro elemento da coleção;
- `'size_t set::size()'`: Retorna o número de elementos da coleção;
- `'void set::swap(argument)'`: Usada para trocar os elementos de duas coleções com tipos de dados idênticos (argumento é a segunda coleção);
- `'set::iterator set::upper_bound(key)'`: Retorna um iterador que aponta para o primeiro elemento que exceda o valor da chave. Se não existir retorna `'set::end()'`.

12.3.9: O Recipiente `'multiset'`

Como a coleção, a classe multicoleção implanta uma coleção ordenada de valores. Para se usar a multicoleção deve-se especificar a diretiva ao pré-processador:

```
#include <set>
```

A principal diferença entre a coleção e a multicoleção é que a multicoleção suporta várias entradas com o mesmo valor.

A coleção e a multicoleção têm as mesmas funções membro. Referir-se à seção 12.3.8 para uma visão geral das funções membro de multicoleção. Algumas funções, contudo, merecem uma atenção adicional quando usadas no contexto das multicoleções. Estas são:

- `'size_t set::count(value)'`: Retorna o número de entradas associadas ao valor dado.
- `'... multiset::erase()'`: Elimina elementos de uma multicoleção:
 - `'size_t erase(value)'`: Elimina todos os elementos com o valor dado. O número de elementos eliminados é retornado.

- `'void erase(pos)'`: Elimina o elemento apontado pelo iterador `'pos'`. Outros elementos possivelmente com o mesmo valor não são tocados.
- `'void erase(first, beyond)'`: Elimina todos os elementos dentro dos limites do iterador `[first, beyond)`;
- `'pair<multiset::iterator, multiset::iterator> multiset::equal_range(value)'`: Retorna um par de iteradores, respectivamente o valor retornado por `'multiset::lower()'` e `'multiset::upper()'`, vistos abaixo. A função é um meio simples de determinar todos os elementos da multicoleção com o mesmo valor;
- `'multiset::iterator multiset::find(value)'`: Retorna um iterador que aponta para o primeiro elemento com o valor especificado. Se não existir o elemento retorna `'multiset::end()'`. O iterador pode ser incrementado para visitar todos os elementos com o valor dado até `'multiset::end()'` ou o iterador não apontar mais para o valor;
- `'... multiset::insert()'`: Insere elementos e retorna um `'multiset::iterator'`, no lugar de `'pair<multiset::iterator, bool>'` como no caso do `'set'`. O iterador retornado aponta para o elemento recém agregado.

Apesar de que as funções `'lower_bound()'` e `'upper_bound()'` atuem de modo idêntico ao `'set'`, sua operação num `'multiset'` merece alguma atenção a mais. Em particular, note-se que com o `'multiset'` as funções produzem o mesmo resultado no caso de não existirem o valor dado: Ambos retornam o primeiro elemento com valor excedente ao valor.

Eis um exemplo mostrando o uso de várias funções membro de um `'multiset'`:

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    string
        sa[] =
        {
            "alpha",
            "echo",
            "hotel",
            "mike",
            "romeo"
        };
};
```

```

multiset<string>
    object(&sa[0], &sa[5]);

object.insert("echo");
object.insert("echo");

multiset<string>::iterator
    it = object.find("echo");

for (; it != object.end(); ++it)
    cout << *it << " ";
cout << endl;

cout << "multiset::equal_range(\"ech\")\n";
pair
<
    multiset<string>::iterator,
    multiset<string>::iterator
>
    itpair = object.equal_range("ech");

if (itpair.first != object.end())
    cout << "lower_bound() aponta para " << *itpair.first <<
endl;

for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";

cout << endl <<
    object.count("ech") << " ocorrências de 'ech'" << endl;

cout << "multiset::equal_range(\"echo\")\n";
itpair = object.equal_range("echo");

for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";

cout << endl <<
    object.count("echo") << " ocorrências de 'echo'" << endl;

cout << "multiset::equal_range(\"echoo\")\n";
itpair = object.equal_range("echoo");

for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";

cout << endl <<

```

```

                                object.count("echoo") << "  ocorrências de 'echoo'" <<
endl;
    }
    /*
        Saída Gerada:

        echo echo echo hotel mike romeo
        multiset::equal_range("ech")
        lower_bound() aponta para echo

        0 ocorrências de 'ech'
        Multiset::equal_range("echo")
        echo echo echo
        3 ocorrências de 'echo'
        Multiset::equal_range("echoo")

        0 ocorrências de 'echoo'
    */

```

12.3.10: O Recipiente 'stack'

A classe 'stack' implanta uma estrutura de dados em pilha. A diretiva ao pré-processador para seu uso é:

```
#include <stack>
```

Uma pilha também é chamada FILO ou LIFO (do inglês: first in, last out), já que o primeiro item a entrar na pilha é o último em sair. Uma pilha é uma estrutura extremamente útil em situações onde os dados devem estar temporariamente disponíveis. Por exemplo em programas onde se mantém uma pilha para guardar variáveis locais de funções: o tempo de vida dessas variáveis é determinado pelo tempo de atividade dessas funções, ao contrário, variáveis globais (ou locais estáticas), que vivem enquanto o programa viver. Outro exemplo é encontrado em calculadoras que usam a notação polaca (Reverse Polish Notation – RPN), onde os operandos dos operadores entram na pilha e os operadores retiram seus operandos da pilha e empilham os resultados das operações de volta na pilha.

Como exemplo do uso de uma pilha considere a figura 11, onde o conteúdo da pilha é mostrado enquanto a expressão $(3 + 4) * 2$ é avaliada. Em RPN esta expressão vem a ser $3\ 4\ +\ 2\ *$, e a figura 11 mostra o conteúdo da pilha depois de que cada elemento (isto é, os operandos e os operadores) é lido da entrada. Note que cada operando é posto na pilha, enquanto cada operador muda o conteúdo da pilha.

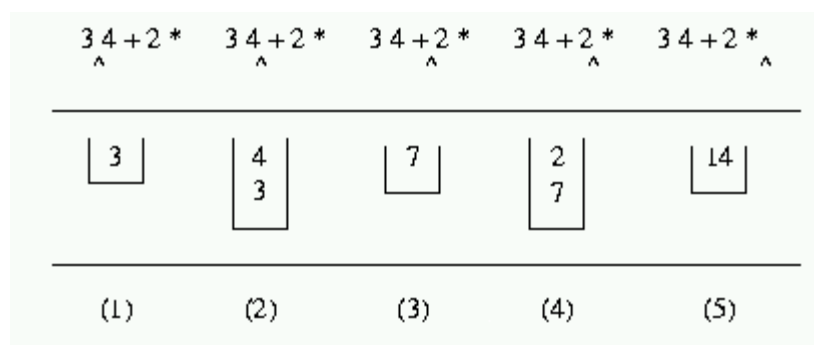


Figure 11(The contents of a stack while evaluating 3 4 + 2 *)

A expressão é avaliada em cinco passos. O sinal entre os símbolos das expressões mostrado na primeira linha da figura 11 mostram qual símbolo acaba de ser lido. A linha seguinte mostra o conteúdo da pilha no momento e a linha final mostra os passos para referência. Note que no passo 2 dois números estão na pilha. O primeiro número (3) está no fundo da pilha. Em seguida, no passo 3, o operador + é lido. O operador retira da pilha dois operandos (assim, a pilha está vazia no momento), calcula a soma e guarda o resultado na pilha (7). Então, no passo 4, o número 2 é lido, que é guardado na pilha. Finalmente, no passo 5, o operador final * é lido, que retira da pilha os valores 2 e 7, computa seu produto e guarda o resultado na pilha. Este resultado (14) pode, então, ser retirado e mostrado de algum meio.

Da figura 11 vemos que uma pilha tem um ponto (o topo) por onde os itens podem ser postos na pilha e retirados da pilha. Este elemento de topo é o único elemento visível imediatamente. Pode ser acessado e modificado diretamente.

Tendo este modelo da pilha em mente, vejamos o que podemos fazer formalmente com ele ao usar o recipiente pilha. Para as pilhas os seguintes construtores, operadores e funções membro estão disponíveis:

Construtores:

- Uma pilha pode ser construída vazia:

```
stack<string> object;
```

- Uma pilha pode ser iniciada com um construtor de cópia:

```
extern stack<string> container;  
stack<string> object(container);
```

Somente o conjunto básico de operadores dos recipientes são suportados pelas pilhas.

As seguintes funções membro estão disponíveis para as pilhas:

- `'bool stack::empty()'`: Retorna verdadeiro se a pilha está vazia;
- `'void stack::push(value)'`: Põe o valor no topo da pilha, escondendo os outros elementos de visão;
- `'void stack::pop()'`: Retira o elemento do topo da pilha. Note que um elemento retirado não é retornado pela função. Nada acontece se `'pop()'` for usada numa pilha vazia. Veja a seção 12.3.3 sobre a discussão das razões de `'pop()'` retornar `'void'`;
- `'size_t stack::size()'`: Retorna o número de elementos na pilha;
- `'Tipo &stack::top()'`: Retorna uma referência ao elemento do topo da pilha (e único visível).

Note que a pilha não suporta iteradores ou operador de indexação. O único elemento acessível é o do topo.

Uma pilha pode ser esvaziada por:

- Remoção repetida de seu último elemento;
- Adjudicação de uma pilha vazia com o mesmo tipo de dados;
- Chamando seu destrutor.

12.3.11: O Recipiente `'hash_map'` e outros Recipientes Aleatórios

O mapa é uma estrutura de dados ordenada. As chaves nos mapas estão ordenadas usando o operador `'operator<()'` do tipo de dados. Geralmente este não é o meio mais rápido de por e tirar dados. O maior benefício da ordem é ser mais legível aos humanos. Um meio muito mais rápido de guardar e retirar dados é o hash.

O método hash usa uma função (dita função hash) para calcular um número (sem sinal) da chave, cujo número é usado como índice na tabela onde as chaves estão depositadas. Retirar uma chave é tão simples como calcular seu valor hash e buscar na tabela no índice calculado: Se a chave está presente, está guardada na tabela e seu valor pode ser retornado. Se não está presente não está guardada.

Ocorrem colisões quando um índice computado já está ocupado por outro elemento. Para

estas situações os recipientes abstratos têm solução, mas este tópico está além do tema deste capítulo.

O compilador Gnu g++ suporta recipiente `'hash_(multi)map'` e `'hash_(multi)set'`. Abaixo o recipiente `'hash_map'` é discutido. Outros recipientes que usam hash (`'hash_multimap'`, `'hash_set'` e `'hash_multiset'`) operam correspondentemente.

Concentrando-nos no `'hash_map'`, seu construtor precisa um tipo da chave, um tipo de valor, um objeto para criar, um valor de hash para a chave e um objeto para comparar as duas chaves para igualdade. As funções hash estão disponíveis para chaves `'char const *'` e para todos os valores escalares `'char'`, `'short'`, `'int'`, etc.. Se outro tipo de dados é usado, uma função hash e um teste de igualdade precisam ser implantados, possivelmente usando funções objetos (ver seção 9.10). Para ambas situações são dados exemplos abaixo.

A classe que implanta a função hash poderia ser chamada hash. Sua função operadora é chamada com `'operator() ()'` e retorna o valor hash da chave passada como argumento.

Existe um algoritmo genérico (veja Capítulo 17) para o teste de igualdade (i.e., `'igual_to()'`), que pode ser usado se o tipo de dado da chave suporta o operador de igualdade. De outra forma uma função objeto especializada poderia ser construída aqui, que suporte o teste de igualdade entre duas chaves. Novamente ambas situações estão ilustradas abaixo.

A classe `'hash_map'` implanta um conjunto associativo onde a chave é guardada segundo um esquema hash. A diretiva ao pré-processador para se usar a classe é:

```
#include <ext/hash_map>
```

O `'hash_(multi)map'` ainda não é parte do estandarte ANSI/ISO. Uma vez que este recipiente se torne parte do estandarte é de se esperar que o prefixo `'ext/'` na diretiva seja removido. Note que a partir da versão 3.2 o compilador Gnu g++, o espaço nomeado `__gnu_cxx` é usado para símbolos definidos em arquivos cabeçalho `ext/`. Ver também a seção 2.1.

Os construtores, operadores e funções membro dos mapas também funcionam com `'hash_map'`. Contudo a eficiência dos `'hash_map'` em termos de velocidade excedem grandemente à dos `'map'`. Conclusões semelhantes pode se tirar a respeito de `'hash_set'`, `'hash_multimap'` e `'hash_multiset'`.

Comparado ao recipiente `'map'`, o `'hash_map'` possui um construtor adicional:

```
hash_map<...> hash(n);
```

Onde `n` é um valor sem sinal, pode ser usado para construir um `'hash_map'` com um número inicial de `n` espaços para a combinação chave/valor. Este número é automaticamente aumentado quando necessário.

O tipo da chave é quase sempre texto. Os tipos de dados das chaves mais comuns são 'char const *' ou 'string'. Se o seguinte arquivo cabeçalho 'hashclasses.h' for instalado no caminho INCLUDE no compilador C++, as fontes podem declarar a seguinte diretiva ao pré-processador para deixar disponível um conjunto de classes usado para usar uma tabela hash:

```
#include <hashclasses.h>
```

Do contrário, as fontes têm que especificar a diretiva ao pré-processador:

```
#include <ext/hash_map>
```

ATENÇÃO: Desde este ponto Arquivo da GPL 'ext/hash_map'

```
#ifndef __INCLUDED_HASHCLASSES_H_
#define __INCLUDED_HASHCLASSES_H_

#include <string>
#include <cctype>

/*
    Note that with the Gnu g++ compiler 3.2 (and beyond?) the ext/ header
    uses the __gnu_cxx namespace for symbols defined in these header
    files.

    When using compilers before version 3.2, do:
        #define __gnu_cxx    std
    before including this file to circumvent problems that may occur
    because of these namespace conventions which were not yet used in
    versions
        before 3.2.

*/

#include <ext/hash_map>
#include <algorithm>

/*
    This file is copyright (c) GPL, 2001-2004
    =====
    august 2004: redundant include guards removed

    october 2002:  provisions for using the hashclasses with the g++ 3.2
                   compiler were incorporated.

    april 2002: espaço nomeado FBB introduced
                abbreviated class templates defined,
                see the END of this comment section for examples of how
```

to use these abbreviations.

jan 2002: redundant include guards added,
 required header files adapted,
 for_each() rather than transform() used

With hash_maps using char const * for the keys:

=====

- * Use 'HashCharPtr' as 3rd template argument for case-sensitive keys
- * Use 'HashCaseCharPtr' as 3rd template argument for case-insensitive keys
- * Use 'EqualCharPtr' as 4th template argument for case-sensitive keys
- * Use 'EqualCaseCharPtr' as 4th template argument for case-insensitive keys

With hash_maps using std::string for the keys:

=====

- * Use 'HashString' as 3rd template argument for case-sensitive keys
- * Use 'HashCaseString' as 3rd template argument for case-insensitive keys
- * OMIT the 4th template argument for case-sensitive keys
- * Use 'EqualCaseString' as 4th template argument for case-insensitive keys

Examples, using int as the value type. Any other type can be used instead

for the value type:

```

// key is char const *, case
sensitive
__gnu_cxx::hash_map<char const *, int, FBB::HashCharPtr,
                    FBB::EqualCharPtr >
    hashtab;

// key is char const *, case
insensitive
__gnu_cxx::hash_map<char const *, int, FBB::HashCaseCharPtr,
                    FBB::EqualCaseCharPtr >
    hashtab;
```

```

// key is std::string, case sensitive
__gnu_cxx::hash_map<std::string, int, FBB::HashString>
    hashtab;

// key is std::string, case
insensitive
__gnu_cxx::hash_map<std::string, int, FBB::HashCaseString,
    FBB::EqualCaseString>
    hashtab;

```

Instead of the above full typedeclarations, the following shortcuts should work as well:

```

FBB::CharPtrHash<int> // key is char const *, case
sensitive
    hashtab;

FBB::CharCasePtrHash<int> // key is char const *, case
insensitive
    hashtab;

FBB::StringHash<int> // key is std::string, case sensitive
    hashtab;

FBB::StringCaseHash<int> // key is std::string, case
insensitive
    hashtab;

```

With these template types iterators and other map-members are also available. E.g.,

```

-----
extern FBB::StringHash<int> dh;

for (FBB::StringHash<int>::iterator it = dh.begin(); it != dh.end();
it++)
    std::cout << it->first << " - " << it->second << std::endl;
-----

```

```

Feb. 2001 - April 2002
Frank B. Brokken (f.b.brokken@rc.rug.nl)
*/

namespace FBB
{

```

```

class HashCharPtr
{
    public:
        size_t operator()(char const *str) const
        {
            return __gnu_cxx::hash<char const *>()(str);
        }
};

class EqualCharPtr
{
    public:
        bool operator()(char const *x, char const *y) const
        {
            return !strcmp(x, y);
        }
};

class HashCaseCharPtr
{
    public:
        size_t operator()(char const *str) const
        {
            std::string s = str;
            for_each(s.begin(), s.end(), *this);
            return __gnu_cxx::hash<char const *>()(s.c_str());
        }
        void operator()(char &c) const
        {
            c = tolower(c);
        }
};

class EqualCaseCharPtr
{
    public:
        bool operator()(char const *x, char const *y) const
        {
            return !strcasecmp(x, y);
        }
};

class HashString
{
    public:
        size_t operator()(std::string const &str) const
        {

```

```

        return __gnu_cxx::hash<char const *>()(str.c_str());
    }
};

class HashCaseString: public HashCaseCharPtr
{
    public:
        size_t operator()(std::string const &str) const
        {
            return HashCaseCharPtr::operator()(str.c_str());
        }
};

class EqualCaseString
{
    public:
        bool operator()(std::string const &s1, std::string const &s2)
const
        {
            return !strcasecmp(s1.c_str(), s2.c_str());
        }
};

template<typename Value>
class CharPtrHash: public
    __gnu_cxx::hash_map<char const *, Value, HashCharPtr,
EqualCharPtr >
{
    public:
        CharPtrHash()
        {}
        template <typename InputIterator>
        CharPtrHash(InputIterator first, InputIterator beyond)
        :
            __gnu_cxx::hash_map<char const *, Value, HashCharPtr,
                EqualCharPtr>(first, beyond)
        {}
};

template<typename Value>
class CharCasePtrHash: public
    __gnu_cxx::hash_map<char const *, Value, HashCaseCharPtr,
        EqualCaseCharPtr >
{
    public:
        CharCasePtrHash()

```



```

    {}
    template <typename InputIterator>
    CharCasePtrHash(InputIterator first, InputIterator beyond)
    :
        __gnu_cxx::hash_map<char const *, Value,
                            HashCaseCharPtr, EqualCaseCharPtr>
                            (first, beyond)
    {}
};

template<typename Value>
class StringHash: public __gnu_cxx::hash_map<std::string, Value,
                                           HashString>
{
    public:
        StringHash()
        {}
        template <typename InputIterator>
        StringHash(InputIterator first, InputIterator beyond)
        :
            __gnu_cxx::hash_map<std::string, Value, HashString>
            (first, beyond)
        {}
};

template<typename Value>
class StringCaseHash: public
    __gnu_cxx::hash_map<std::string, int, HashCaseString,
                       EqualCaseString>
{
    public:
        StringCaseHash()
        {}
        template <typename InputIterator>
        StringCaseHash(InputIterator first, InputIterator beyond)
        :
            __gnu_cxx::hash_map<std::string,
                                int, HashCaseString,
                                EqualCaseString>(first, beyond)
        {}
};

template<typename Key, typename Value>
class Hash: public
    __gnu_cxx::hash_map<Key, Value,
                       __gnu_cxx::hash<Key>(),

```

```

equal<Key>())

{};

}
#endif

```

Até aqui o arquivo GPL `ext/hash_map`

O seguinte programa define um 'hash_map' contendo os nomes dos meses do ano e os dias que esses meses têm (usualmente). Então, usando o operador de indexação os dias em diversos meses são mostrados. O operador igualdade usa o algoritmo genérico 'equal_to<string>', que é o quarto argumento padrão do construtor do 'hash_map':

```

#include <iostream>
// O seguinte arquivo cabeçalho tem que estar no caminho do
compilador
// INCLUDE path:
#include <hashclasses.h>
using namespace std;
using namespace FBB;

int main()
{
    __gnu_cxx::hash_map<string, int, HashString > meses;
    // Alternativamente, usando as classes definidas em
hashclasses.h,
    // as seguintes definições poderiam ter sido usadas:
    //      CharPtrHash<int> meses;
    // ou:
    //      StringHash<int> meses;

    meses["janeiro"] = 31;
    meses["fevereiro"] = 28;
    meses["março"] = 31;
    meses["abril"] = 30;
    meses["maio"] = 31;
    meses["junho"] = 30;
    meses["julho"] = 31;
    meses["agosto"] = 31;
    meses["setembro"] = 30;
    meses["outubro"] = 31;
    meses["novembro"] = 30;
    meses["dezembro"] = 31;

    cout << "setembro    -> " << meses["setembro"] << endl <<
        "abril        -> " << meses["abril"] << endl <<

```

```

        "junho      -> " << meses["junho"] << endl <<
        "novembro   -> " << meses["novembro"] << endl;
    }
    /*
        Saída Gerada:
    setembro -> 30
    abril    -> 30
    junho    -> 30
    novembro -> 30
    */

```

Os recipientes 'hash_multimap', 'hash_set' e 'hash_multiset' são usados analogamente. Para estes recipientes as classes 'equal' e 'hash' também devem estar definidas. O 'hash_multimap' também requer o arquivo cabeçalho 'hash_map'.

Antes de usar os recipientes 'hash_set' e 'hash_multiset' use a seguinte diretiva ao pré-processador:

```
#include <ext/hash_set>
```

12.4: O Recipiente 'complex'

O recipiente 'complex' é um recipiente especializado, pois, define operações sobre números complexos, com tipos de dados possíveis de números reais e imaginários.

Para seu uso exige a seguinte diretiva ao pré-processador:

```
#include <complex>
```

O recipiente complexo é usado para definir números complexos, consistentes de duas partes, representando a parte real e a parte imaginária de um número complexo.

Quando iniciando (ou adjudicando) uma variável complexa, a parte imaginária pode ser ignorada, nesse caso esta parte será 0 (zero). Como padrão ambas partes são zero.

Quando os números complexos são definidos, o tipo de definição requer a especificação do tipo de dado das partes, real e imaginária. P.ex.:

```

complex<double>
complex<int>
complex<float>

```

Note que as partes real e imaginária dos números complexos possuem o mesmo tipo de dado.

Assume-se abaixo, silenciosamente, que o tipo usado é 'complex<double>'. Assim os números complexos podem ser iniciados como segue:

- 'target': Iniciação padrão: As partes real e imaginária são 0.
- 'target(1)': Parte real 1 e imaginária 0.
- 'target(0, 3.5)': Parte real 0 e imaginária 3,5.
- 'target(source)': 'target' iniciado com os valores de 'source'.

Também se pode usar valores complexos anônimos. No seguinte exemplo dois valores complexos anônimos são postos numa pilha de números complexos, para serem retirados posteriormente:

```
#include <iostream>
#include <complex>
#include <stack>

using namespace std;

int main()
{
    stack<complex<double> >
        cstack;

    cstack.push(complex<double>(3.14, 2.71));
    cstack.push(complex<double>(-3.14, -2.71));

    while (cstack.size())
    {
        cout << cstack.top().real() << ", " <<
            cstack.top().imag() << "i" << endl;
        cstack.pop();
    }
}
/*
    Saída Gerada:
-3.14, -2.71i
3.14, 2.71i
*/
```

Note que é requerido um espaço extra entre as setas adjacentes na especificação do tipo de 'cstack'.

As seguintes funções membro e operadores são definidos para os números complexos (abaixo

'value' pode ser um tipo escalar primitivo ou um objeto complexo):

Além dos operadores padrão para os recipientes, os recipientes complexos aceitam os seguintes operadores

- `'complex complex::operator+(value)'`: Retorna a soma do complexo corrente e 'value';
- `'complex complex::operator-(value)'`: Retorna a diferença entre o complexo corrente e 'value';
- `'complex complex::operator*(value)'`: Retorna o produto entre o complexo corrente e 'value';
- `'complex complex::operator/(value)'`: Retorna o quociente entre o complexo corrente e 'value';
- `'complex complex::operator+=(value)'`: Adiciona ao recipiente complexo corrente, retorna o novo valor;
- `'complex complex::operator-=(value)'`: Subtrai do complexo corrente 'value' e retorna o novo valor;
- `'complex complex::operator*=(value)'`: Multiplica o complexo corrente por 'value' e retorna o novo valor;
- `'complex complex::operator/=(value)'`: Divide o complexo corrente por 'value' e retorna o novo valor;
- `'Tipo complex::real()'`: Retorna a parte real do complexo;
- `'Tipo complex::imag()'`: Retorne a parte imaginária do complexo;

Diversas funções matemáticas estão disponíveis para os recipientes complexos, tais como `'abs()'`, `'arg()'`, `'conj()'`, `'cos()'`, `'cosh()'`, `'exp()'`, `'log()'`, `'norm()'`, `'polar()'`, `'pow()'`, `'sin()'`, `'sinh()'` e `'sqrt()'`. Estas são funções normais, não membros, que aceitam números complexos como argumento. Por exemplo:

```
abs(complex<double>(3, -5));  
pow(target, complex<int>(2, 3));
```

Os números complexos podem ser extraídos de objetos `'istream'` e inseridos em objetos `'ostream'`. A inserção resulta num par ordenado (x, y), onde x representa a parte real e y a parte imaginária do número complexo. A mesma forma pode ser usada ao extraírmos um número complexo de um objeto `'istream'`. Contudo formas mais simples também são permitidas. P.ex., 1,2345: Só a parte real, a

imaginária será posta em 0; (1.2345): é o mesmo valor.

Capítulo 13: Herança

Quando programando em C, os problemas de programação comumente são resolvidos usando-se uma estrutura do topo para a base: As funções e ações do programa são definidos em termos de sub-funções, que novamente são definidos em sub-sub-funções, etc.. Isto conduz a uma hierarquia do código: 'main()' no topo, seguida por um nível de funções chamadas por 'main()', etc..

Em C++ as dependências entre código e dados são freqüentemente definidas em termos de dependências entre classes. Isto parece com a composição (ver seção 6.4), onde os objetos de uma classe contêm objetos de outra classe como seus dados. Mas a relação descrita aqui é diferente: Uma classe pode ser definida em relação a outra anterior, pré-existente. Isto produz uma nova classe com todas as funcionalidades da classe original e adicionalmente introduzindo suas próprias funcionalidades. No lugar de composição, onde dada classe contém outra classe, aqui nos referimos a uma derivação, onde dada classe é outra classe.

Outro termo para derivação é herança: A nova classe herda a funcionalidade de uma classe existente, enquanto a classe existente não aparece como um membro de dados na definição da nova classe. Quando discutimos a herança a classe existente é dita classe base, enquanto a nova classe é chamada de classe derivada.

A derivação de classes é freqüentemente usada quando a metodologia de desenvolvimento de programas em C++ é usada totalmente. Neste capítulo veremos primeiro as possibilidades sintáticas oferecidas pela C++ para derivar classes de outras classes. Então examinaremos algumas possibilidades resultantes.

Como vimos no capítulo introdutório (ver seção 2.4), na solução orientada aos objetos dos problemas, as classes são identificadas durante a análise do problema, depois do que os objetos das classes definidas representam entidades do problema que temos entre mãos. As classes são postas em hierarquia, onde a classe do topo contém o mínimo de funcionalidade. Cada nova derivação (portanto descendo na hierarquia de classe) agrega nova funcionalidade, em comparação com as classes já existentes.

Neste capítulo utilizaremos um sistema simples de classificação de um veículo para construir uma hierarquia de classes. A primeira classe é 'Veículo', que implanta como funcionalidade a possibilidade de carregar e descarregar um veículo. O nível seguinte na hierarquia são veículos terrestre, aquático e aéreo.

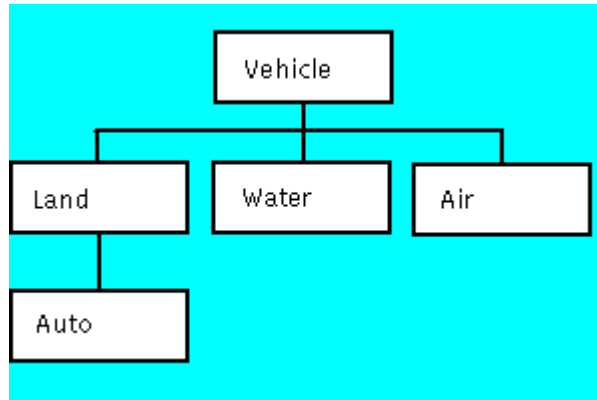


Figura 12.

Hierarquia inicial dos objetos 'Veículo'

13.1: Tipos Relativos

As relações entre as classes propostas, representando os diferentes tipos de veículos, são melhor ilustradas aqui. A figura mostra a hierarquia entre objetos: um 'Carro' é um caso especial de um veículo terrestre, que por sua vez é um caso especial de veículo.

A classe 'Veículo' é o 'maior denominador comum' no sistema de classificação. Para o objetivo do exemplo nesta classe implantamos a funcionalidade de peso e descarga dos veículos:

```
class Veículo
{
    size_t d_peso;

    public:
        Veículo();
        Veículo(size_t peso);
```



```

        size_t peso() const;
        void setPeso(size_t peso);
};

```

Usando esta classe, o peso do veículo pode ser definida logo que o objeto correspondente seja criado. Numa etapa posterior o peso pode ser redefinido ou retirado.

Para representar veículos que viagem sobre a terra uma nova classe 'Terra' deve ser definida com a funcionalidade de 'Veículo', adicionando suas especificidades informacionais e funcionais. Digamos que nos interessamos pela velocidade dos veículos terrestres e seus pesos. A relação entre 'Veículo' e 'Terra' poderia, por certo, ser representada usando-se composição, mas seria impróprio: A composição sugeriria que um veículo terrestre contém veículo, enquanto que a relação é oposta, veículo terrestre é um caso especial de veículo.

A relação em termos de composição poderia inchar o código sem necessidade. P.ex., considere o seguinte fragmento de código, onde a classe 'Terra' usando a composição (mostra só a função 'setPeso()'):

```

class Terra
{
    Veículo d_v;          // Veículo composto
public:
    void setPeso(size_t peso);
};

void Terra::setPeso(size_t peso)
{
    d_v.setPeso(peso);
}

```

Usando a composição, a função 'setPeso()' da classe 'Terra' só serve para passar seus argumentos a 'Veículo::setPeso()'. Isto só em relação à manipulação de peso, 'Terra::setPeso()' não introduz nenhuma funcionalidade extra, só código extra. Claramente esta duplicação de código é supérflua: Um objeto 'Terra' deve ser um 'Veículo', não deve conter 'Veículo'.

A relação desejada é obtida por herança: 'Terra' é derivada de 'Veículo', da qual 'Veículo' é a classe de base na derivação:

```

class Terra: public Veículo
{
    size_t d_speed;
public:
    Terra();
    Terra(size_t peso, size_t velocidade);

    void setvelocidade(size_t velocidade);
}

```

```

        size_t velocidade() const;
};

```

Pós-pondo ao nome da classe 'Terra' em sua definição: 'public Veículo' a derivação está feita: A classe 'Terra' agora toda funcionalidade de sua classe de base 'Veículo' mais suas informações e funcionalidades. A funcionalidade extra consiste de um construtor com dois argumentos e funções de interface para aceder o membro de dados 'velocidade'. No exemplo acima é usada derivação pública. A linguagem C++ também suporta derivação privada e derivação protegida. Na seção 13.6 discutimos suas diferenças. Um exemplo simples mostrando as possibilidades da classe derivada 'Terra' é:

```

Terra vei(1200, 145);

int main()
{
    cout << "Peso do Veículo " << vei.peso() << endl
          << "A Velocidade é " << vei.velocidade() << endl;
}

```

Este exemplo mostra dois aspectos da derivação:

Primeiro, 'peso()' não é mencionado como membro da interface de 'Terra'. No entanto é usada em 'vei.peso()'. Esta função membro é uma parte implícita da classe, herdada de seu pai 'Veículo'.

Segundo, apesar de que a classe 'Terra' agora contém a funcionalidade de 'Veículo', os campos privados de 'Veículo' permanecem privados: Só podem ser acessados pelas funções de 'Veículo'. Isto significa que os membros de 'Terra' têm que usar funções de interface (como 'peso()' e 'setPeso()') para acessar o campo 'peso', como qualquer código externo à classe 'Veículo'. Esta restrição é necessária para reforçar o princípio de encobrimento de dados. A classe 'Veículo' pode, p.ex., ser re-codificada e re-compilada, depois do que o programa pode ser re-linkado. A classe 'Terra' permanece inalterado.

A nota anterior não é completamente verdadeira: Se a organização interna de 'Veículo' muda, então a organização interna dos objetos 'Terra' que contêm dados de 'Veículo' também mudam. Isto significa que os objetos da classe 'Terra', depois da mudança de 'Veículo', podem requerer mais (ou menos) memória que antes da modificação. Contudo em tal situação não devemos nos preocupar sobre as funções membro da classe pai 'Veículo' na classe 'Terra'. Teremos que re-compilar as fontes de 'Terra', já que as posições relativas dos membros de dados nos objetos da classe 'Terra' estariam mudadas devido às modificações na classe 'Veículo'.

Como regra prática: As classes derivadas de outras classes precisam ser completamente re-compiladas (mas não modificadas) depois de uma mudança na organização dos dados, i.e., os membros de dados de suas classes de base. Como a inclusão de uma novas funções membro à classe não altera a organização dos dados, não é necessário re-compilar depois da adição de novas funções membro. (Um ponto sutil a notar é que ao escrever uma nova função membro e este for a primeira função membro

virtual da classe resulta num novo membro de dados: Um ponteiro oculto para uma tabela de funções virtuais. Assim, neste caso também é necessária a re-compilação, já que os membros de dados da classe foram silenciosamente modificados. Este tópico é discutido no Capítulo 14).

No seguinte exemplo assumimos que a classe 'Carro', representando automóveis contém o peso, a velocidade e o nome do carro. Esta classe é convenientemente derivada de 'Terra':

```
class Carro: public Terra
{
    char *d_nome;

public:
    Carro();
    Carro(size_t peso, size_t velocidade, char const *nome);
    Carro(Carro const &outro);

    ~Carro();

    Carro &operator=(Carro const &outro);

    char const *nome() const;
    void setNome(char const *nome);
};
```

Na definição de classe acima, 'Carro' é derivada de 'Terra', que por sua vez é derivada de 'Veículo'. A isto chamamos de derivação aninhada: 'Terra' é chamada de classe base direta de 'Carro', enquanto 'Veículo' de classe base indireta.

Note a presença de um destrutor, um construtor de cópia e um operador sobrecarregado na classe 'Carro'. Como esta classe usa um ponteiro para acessar a memória dinamicamente alocada, estes membros devem ser parte da interface de classe.

13.2: O construtor de uma classe derivada

Como mencionado anteriormente, uma classe derivado herda a funcionalidade de sua classe de base. Nesta seção descreveremos os efeitos que a herança tem sobre o construtor de uma classe derivada.

Como ficará claro da definição da classe 'Terra', existe um construtor para criar ambos 'peso' e 'velocidade' de um objeto. A implantação deste construtor é:

```
Terra::Terra (size_t peso, size_t velocidade)
{
    setPeso(peso);
```

```

        setVelocidade(velocidade);
    }

```

Esta implantação tem a seguinte desvantagem: O compilador C++ gerará o código que chama o construtor da classe de base padrão de cada construtor da classe derivada, a menos instrução explícita do contrário. Isto pode ser comparado à situação encontrada em objetos compostos (ver seção 6.4).

Conseqüentemente na implantação acima o construtor padrão de 'Veículo' é chamado, que provavelmente inicia o 'peso' do veículo, só para ser redefinida logo em seguida pela função 'setPeso()'.

Uma solução mais eficiente é, claro está, chamar o construtor de 'Veículo' com um argumento sem sinal diretamente. A sintaxe para isto é mencionar o construtor de 'Veículo' (com seu argumento) imediatamente depois da lista de argumentos do construtor da classe derivada. Tal iniciador da classe de base é mostrado no exemplo a seguir. Seguindo o cabeçalho do construtor vem um sinal de dois pontos, seguido pelo construtor da classe de base. Só então qualquer membro iniciador pode ser especificado (usando vírgulas para separar múltiplos iniciadores), seguido pelo corpo do construtor:

```

Terra::Terra(size_t peso, size_t velocidade)
:
    Veículo(peso)
{
    setVelocidade(velocidade);
}

```

13.3: O destrutor de uma classe derivada

Os destrutores das classes são chamados automaticamente quando um objeto é destruído. Isto permanece verdadeiro para objetos de classes derivadas de outras classes. Assuma que temos a seguinte situação:

```

class Base
{
    public:
        ~Base();
};

class Derived: public Base
{
    public:
        ~Derived();
};

int main()
{
    Derived
        derived;
}

```

```
}
```

No fim da função 'main()', o objeto derivado cessa de existir. Assim seu destrutor (~Derived()) é chamado. Contudo, como 'derived' é também um objeto de 'Base', o destrutor '~Base()' é chamado também. Não é necessário chamar o destrutor da classe de base explicitamente do destrutor da classe derivada.

Os construtores e destrutores são chamados de maneira semelhante a uma pilha: Quando 'derived' é construído, o construtor apropriado da classe base é chamado primeiro, então o construtor da classe derivada apropriado é chamado. Quando o objeto 'derived' é destruído seu destrutor é chamado antes, automaticamente seguido pela ativação do destrutor da classe de base. Um destrutor de uma classe derivada sempre é chamado antes que o da classe de base.

13.4: Redefinindo funções membro

A funcionalidade de todos os membros de uma classe de base (que portanto são válidas nas classes derivadas) podem ser redefinidas. Esta característica está ilustrada nesta seção.

Assumamos que um sistema de classificação de veículos pode representar caminhões, consistindo de duas partes: o cavalo mecânico, que puxa a segunda parte, o reboque. Ambos o cavalo mecânico e o reboque possuem seus próprios pesos e a função peso() retorna o peso combinado.

A definição de um caminhão começa com a definição da classe, derivada de Carro, mas então é expandida para abranger mais um campo sem sinal, representando a informação adicional de peso. Aqui escolhemos representar o peso da parte de frente de um caminhão na classe Carro e armazenar o peso do reboque num campo adicional:

```
class Caminhão: public Carro
{
    size_t d_peso_reboque;

public:
    Caminhao();
    Caminhao(size_t cavalo_ps, size_t velocidade, char const *nome,
              size_t reboque_ps);

    void setPeso(size_t cavalo_ps, size_t reboque_ps);
    size_t peso() const;
};

Caminhao::Caminhao(size_t cavalo_ps, size_t velocidade, char const *nome,
                  size_t reboque_ps)
:
```

```

    Carro(cavalo_ps, velocidade, nome)
{
    d_reboque_peso = reboque_ps;
}

```

Note que a classe 'Caminhao' agora contém duas funções já presentes na classe Carro: setPeso() e peso().

- A re-definição de 'setPeso()' não coloca problemas: Esta função simplesmente é redefinida para realizar ações específicas a um objeto 'Caminhao'.
- A re-definição de setPeso(), contudo, substituirá Carro::setPeso(): Para um Caminhao somente a função setPeso() com dois argumentos pode ser usada.
- A função de Veículo setPeso() permanece válida para um caminhão, mas agora tem que ser chamada explicitamente, como Carro::setPeso(), já que está fora de visibilidade.

Esta última função está oculta, mesmo porque Carro::setPeso() tem um só argumento. Para implantar Caminhao::setPeso(), poderíamos escrever:

```

void Caminhao::setPeso(size_t cavalo_ps, size_t reboque_ps)
{
    d_peso_reboque = reboque_ps;
    Carro::setPeso(cavalo_ps);    // note: Carro:: é requerido
}

```

- Fora da versão de setPeso() da classe Carro esta função é acessada usando o operador de resolução de escopo. Assim se um objeto Caminhao t necessita seu peso de Carro, precisa usar:

```
t.Carro::setPeso(x);
```

- Uma alternativa para não usar o operador de resolução de escopo é incluir explicitamente um membro com o mesmo protótipo do da classe de base. Este membro da classe pode ser implantado em linha para chamar o membro da classe de base. Esta pode ser uma solução elegante para soluções ocasionais. P.ex., incluímos o seguinte na interface da classe Caminhao:

```

//Na interface
void setPeso(size_t cavalo_ps)
//Abaixo da interfaces
inline void Truck::setWeight(size_t engine_wt)
{
    Auto::setWeight(engine_wt);
}

```

Agora a função de um argumento setPeso() pode ser usada por objetos Caminhao sem o operador de resolução de escopo. Como a função está definida em linha não envolve a sobrecarga de uma chamada a mais.

- A função `peso()` já está definida em `Carro`, já que é herdada de `Veículo`. No caso, a classe `Caminhao` poderia re-definir esta função membro para permitir o peso extra (reboque) nos objetos `Caminhao`:

```
size_t Caminhao::peso() const
{
    return
        (
            Carro::peso() +      // filha de:
            d_peso_reboque      // cavalo mais
        );                      // reboque
}
```

- O seguinte exemplo mostra o uso das funções membro da classe `Caminhao`, mostrando diversos pesos:

```
int main()
{
    Terra veh(1200, 145);
    Caminhao lorry(3000, 120, "Juggernaut", 2500);

    lorry.Veículo::setPeso(4000);

    cout << endl << "Peso do Caminhão " <<
        lorry.Veículo::peso() << endl <<
        "Peso do Caminhão + Reboque " << lorry.weight() << endl <<
        "A velocidade é " << lorry.speed() << endl <<
        "O Nome é " << lorry.name() << endl;
}
```

Note a chamada explícita a `Veículo::setPeso(4000)`: Assumindo que `setPeso(size_t cavalo_ps)` não é parte da interface da classe `Caminhao`, precisa ser chamada explicitamente usando-se o operador de resolução de escopo, já que a função de um só argumento `setPeso()` está oculta na classe `Caminhao`.

Com `Veículo::peso` e `Caminhao::peso()` a situação é um pouco diferente: Aqui a função `Caminhao::peso()` é uma re-definição de `Veículo::peso()`, assim para se acessar `Veículo::peso()` requer o operador de resolução de escopo `Veículo::`.

13.5: Herança Múltipla

Até aqui uma classe sempre derivava de uma só classe. A linguagem C++ também suporta derivação múltipla, onde uma classe é derivada de diversas classes de base e portanto herda a funcionalidade de múltiplas classes pais ao mesmo tempo.

No caso onde consideramos a herança múltipla é justo considerar a nova classe derivada

como instância de ambas classes de base. De outra forma a composição é mais apropriada. Em geral a derivação linear, onde há uma só classe de base, é muito mais freqüente que a derivação múltipla. A maioria dos objetos têm um propósito primário. Considere o protótipo de um objeto que a herança múltipla foi usada ao extremo: A faca do exército suíço! Este objeto é uma faca, é uma tesoura, é um abridor de latas, é uma chave de fenda, é

Como podemos construir uma 'faca do exército suíço' em C++? Primeiro precisamos (pelo menos) duas classes de base. Por exemplo, assumamos que estamos projetando uma ferramenta para construir um instrumento de painel da carlinga de um avião. Projetamos todas classes de instrumentos, como horizonte artificial e um altímetro. Um componente freqüentemente vistos em aviões é um conjunto de navegação (nav-com set): Uma combinação de um receptor de navegação (a parte 'nav') e unidade de rádio comunicação (a parte 'com'). Para definirmos o aparelho primeiro projetamos a classe NavSet. Aqui seus membros de dados estão omitidos:

```
class NavSet
{
    public:
        NavSet (Intercom &intercom, VHF_Dial &dial);

        size_t activeFrequency() const;
        size_t standByFrequency() const;

        void setStandByFrequency(size_t freq);
        size_t toggleActiveStandby();
        void setVolume(size_t level);
        void identEmphasis (bool on_off);
};
```

No construtor da classe assumimos a existência das classes Intercom, que é usada pelo piloto para ouvir as informações de navegação transmitidas e uma classe VHF_Dial que é usada para representar visualmente aquilo que NavSet recebe.

Em seguida construímos a classe ComSet. Novamente omitindo os membros de dados:

```
class ComSet
{
    public:
        ComSet (Intercom &intercom);

        size_t frequency() const;
        size_t passiveFrequency() const;

        void setPassiveFrequency( size_t freq);
        size_t toggleFrequencies();

        void setAudioLevel( size_t level);
        void powerOn (bool on_off);
};
```



```

        void testState(bool on_off);
        void transmit(Message &message);
};

```

Usando objetos desta classe podemos receber mensagens, como se transmitidas pelo Intercom, podemos também transmitir mensagens, usando um objeto Message que é passada a um objeto ComSet usando sua função membro transmit().

Agora estamos prontos a construir a classe NavComSet:

```

class NavComSet: public ComSet, public NavSet
{
    public:
        NavComSet(Intercom &intercom, VHF_Dial &dial);
};

```

Feito. Agora definimos uma classe 'NavComSet' que é ambas 'NavSet' e 'ComSet': As possibilidades de cada classe de base estão disponíveis na classe derivada, usando derivação múltipla.

Por favor, note o seguinte na derivação múltipla:

- A palavra chave 'public' está presente antes dos nomes de ambas classes de base ('NavSet' e 'ComSet'). Assim é porque a derivação padrão em C++ é privada: A palavra chave 'public' precisa ser repetida antes da especificação de cada classe de base. As classes de base podem não ter o mesmo tipo de derivação: Uma classe de base pode ter derivação pública, outra classe de base pode usar derivação protegida e outra derivação privada.
- A classe com derivação múltipla 'NavComSet' não introduz funcionalidades adicionais, mas simplesmente combina duas classes existentes numa nova classe agregada. A linguagem C++ oferece a possibilidade de simplesmente envolver classes numa classe mais complexa. Esta característica da C++ é muito usada. Usualmente é melhor desenvolver classes simples cada uma com uma funcionalidade simples e bem definida. As classes mais complexas sempre podem ser construídas destes blocos mais simples.
- Eis aqui a implantação do construtor de 'NavComSet':

```

NavComSet::NavComSet(Intercom &intercom, VHF_Dial &dial)
:
    ComSet(intercom),
    NavSet(intercom, VHF_Dial)
{}

```

O construtor não requer código extra: Seu propósito é ativar os construtores das classes de base. A ordem em que os iniciadores das classes de base são chamados não é ditada pela ordem em que de chamada no código do construtor, mas pela ordem das classes de base na

interface da classe.

- A definição da classe 'NavComSet' não necessita de membros de dados extra ou de funções membro: Aqui (e freqüentemente) as interfaces herdadas trazem toda funcionalidade e dados requeridos para a classe de derivação múltipla operar propriamente. Claro está que quando definimos as classes de base nos fazemos a vida fácil, usando nomes estritamente diferentes para as funções membro. Assim, há uma função 'setVolume()' na classe 'NavSet' e uma função 'setAudioLevel()' na classe 'ComSet'. Um pouco trapaceado, pois devemos esperar que ambas unidades, de fato, possuem um objeto composto 'Amplifier', que manipula os níveis de volume. Uma classe revista deveria ou usar uma função membro 'Amplifier &lifier() const', e deixá-la manipular a amplificação, ou funções de acesso para, p.ex., o volume ser comum às classes 'NavSet' e 'ComSet' como funções membro com o mesmo nome (p.ex., 'setVolume()'). Quando duas classes de base usam os mesmos nomes para funções membro, é necessário medidas especiais para prevenir ambigüidades:

- A classe de base intencionada pode ser explicitamente mencionada, usando-se o nome da classe de base e o operador de resolução de escopo em combinação com o nome dúbio da função membro:

```
NavComSet navcom(intercom, dial);

navcom.NavSet::setVolume(5);    // regula o volume de NavSet
navcom.ComSet::setVolume(5);    // regula o volume de ComSet
```

- A interface de classe é estendida por funções membro que explicitam ao usuário da classe. Estas funções extra normalmente são definidas em linha:

```
class NavComSet: public ComSet, public NavSet
{
    public:
        NavComSet(Intercom &intercom, VHF_Dial &dial);
        void comVolume(size_t volume)
        {
            ComSet::setVolume(volume);
        }
        void navVolume(size_t volume)
        {
            NavSet::setVolume(volume);
        }
};

inline void NavComSet::comVolume(size_t volume)
{
    ComSet::setVolume(volume);
}

inline void NavComSet::navVolume(size_t volume)
{
```

```

    NavSet::setVolume(volume);
}

```

- Se a classe 'NavComSet' é obtida de uma terceira parte e não pode ser alterada, uma classe envolvente pode ser usada, que faz a explicitação prévia em nossos programas:

```

class MyNavComSet: public NavComSet
{
    public:
        MyNavComSet(Intercom &intercom, VHF_Dial &dial)
        :
            NavComSet(intercom, dial);
        {}
        void comVolume(size_t volume)
        {
            ComSet::setVolume(volume);
        }
        void navVolume(size_t volume)
        {
            NavSet::setVolume(volume);
        }
};

inline MyNavComSet::MyNavComSet(Intercom &intercom, VHF_Dial &dial)
:
    NavComSet(intercom, dial);
{}
inline void MyNavComSet::comVolume(size_t volume)
{
    ComSet::setVolume(volume);
}
inline void MyNavComSet::navVolume(size_t volume)
{
    NavSet::setVolume(volume);
}

```

13.6: Derivação pública, protegida e privada

Como vimos, as classes podem ser derivadas de outras classes usando a herança. Em geral o tipo de derivação é público, isto implica que os direitos de acesso à interface da classe de base ficam inalterados na classe derivada.

Além da derivação pública a linguagem C++ também suporta derivação protegida e derivação privada.

Para se usar a derivação protegida a palavra chave 'protected' é especificada na lista de herança:

```
class Derived: protected Base
```

Com a derivação protegida toda a classe de base, membros públicos e protegidos, recebem direitos de acesso protegidos na classe derivada. Os membros protegidos ficam disponíveis para a classe e para todas as classes que direta ou indiretamente derivam dela.

Para usar a derivação privada a palavra chave 'private' é especificada na lista de herança:

```
class Derived: private Base
```

Com a derivação privada todos os membros da classe de base recebem acesso privado, estão disponíveis somente internamente à classe.

Podem ocorrer combinações de herança. Por exemplo, ao projetarmos uma classe 'stream' em geral é derivada da classe 'std::istream' ou 'std::ostream'. Contudo, antes de ser possível construir uma 'stream', um 'std::streambuf' precisa estar disponível. Tirando vantagem de que a ordem de herança é tomada seriamente pelo compilador, podemos usar herança múltipla (veja a seção 13.5) para derivar a classe de ambas, 'std::streambuf' e de, p.ex., 'std::ostream'. Como nossa classe encara seus clientes como uma 'std::ostream' e não como uma 'std::streambuf', usamos derivação privada para a última e pública para a classe formadora:

```
class Derived: private std::streambuf, public std::ostream
```

13.7: Conversões entre classes de base e classes derivadas

Quando a herança é usada para definir classes, pode-se dizer que um objeto de uma classe derivada é ao mesmo tempo um objeto da classe de base. Isto tem importantes conseqüências na adjudicação de objetos e onde são usados ponteiros ou referências a tais objetos. Ambas situações discutiremos a seguir.

13.7.1: Conversões na adjudicação de objetos

Continuando nossa discussão da classe 'NavComSet', introduzida na seção 13.5, começamos definindo dois objetos, um de uma classe de base e outro de uma classe derivada:

```
ComSet com(intercom);  
NavComSet navcom(intercom2, dial2);
```

O objeto 'navcom' é construído usando um objeto 'Intercom' e um 'Dial'. Contudo, um 'NavComSet' é ao mesmo tempo um 'ComSet', permitindo a adjudicação de 'navcom' (um objeto de uma classe derivada) a 'com' (um objeto de uma classe de base):

```
com = navcom;
```

O efeito desta adjudicação é que o objeto 'com' agora pode comunicar-se com 'intercom2'. Como 'ComSet' não possui objetos 'VHF_Dial', o objeto 'dial' de 'navcom' é ignorado pela adjudicação: Quando adjudicamos um objeto de uma classe de base a um objeto de uma classe derivada só os membros de dados da classe de base são adjudicados, outros membros de dados são ignorados.

A adjudicação de um objeto de uma classe de base a um objeto de uma classe derivada, contudo, é problemática. Em adjudicações como:

```
navcom = com;
```

Não fica claro como o membro de dados de 'NavComSet', 'VHF_Dial' será re-adjudicado, já que falta no objeto 'ComSet' 'com'. Esta adjudicação é portanto rejeitada pelo compilador. Se bem que objetos de classes derivadas são também objetos da classe de base, o inverso não é verdadeiro: Um objeto da classe de base não é também um objeto da classe derivada.

Se aplica a seguinte regra geral: Em adjudicações onde os objetos da classe de base e objetos da classe derivada estão envolvidos, adjudicações onde os dados são baixados é legal. Contudo, adjudicações onde os dados ficam indeterminados não são permitidas. Claro, é possível é possível redefinir um operador de adjudicação para permitir a adjudicação de um objeto de uma classe derivada de um objeto de uma classe de base. P.ex., para assegurar a compilação de:

```
navcom = com;
```

A classe 'NavComSet' precisa de um operador de adjudicação sobrecarregado que aceite um objeto 'ComSet' como argumento. É responsabilidade do programador construir um operador de adjudicação que decida o que fazer com os dados que faltam.

13.7.2: Conversões na adjudicação de ponteiros

Retornamos à nossa classe 'Veículo' e definimos os seguintes objetos e ponteiros:

```
Terra terra(1200, 130);  
Carro carro(500, 75, "Daf");  
Caminhao caminhao(2600, 120, "Mercedes", 6000);  
Veículo *vp;
```

Agora podemos adjudicar endereços dos três objetos das classes derivadas ao ponteiro de 'Veículo':

```
vp = &terra;  
vp = &carro;  
vp = &caminhao;
```

Todas essas adjudicações são aceitáveis. Contudo, uma conversão implícita da classe

derivada para a classe de base 'Veículo' é usada, já que 'vp' é definido como ponteiro a um objeto 'Veículo'. Daí, quando usamos 'vp' só as funções membro de manipulação do peso podem ser chamadas, já que é a única funcionalidade de 'Veículo'. Até aí o compilador pode garantir que o objeto 'vp' apontará.

A mesma razão é verdadeira para referências a 'Veículo'. Se, p.ex., uma função é definida com um parâmetro de referência a 'Veículo'. Na função, os membros específicos de 'Veículo' continuam acessíveis. Lembre-se que uma referência não é nada mais que um apontador disfarçado: Imita uma variável plena, mas é um ponteiro.

Esta funcionalidade restrita tem importantes conseqüências para a classe 'caminhao'. Depois da adjudicação 'vp = &caminhao', 'vp' aponta para um objeto 'caminhao'. Assim, 'vp -> peso()' retornará 2600 em lugar de 8600 (o peso combinado do cavalo e reboque: 2600 + 6000), que deveria ser retornado por 'caminhao.peso()'.

Quando uma função é chamada usando um ponteiro a um objeto, então o tipo do ponteiro (e não do objeto) determina quais funções membro estão disponíveis e executa. Em outras palavras, C++ implicitamente converte o tipo de um objeto acessado por um ponteiro ao tipo do ponteiro.

Se o tipo do objeto para o qual um ponteiro aponta é conhecido, um tipo explícito de cast pode ser usado para acessar o conjunto completo de funções membro disponíveis para o objeto:

```
Caminhao caminhao;
Veículo *vp;

vp = &caminhao;           // vp agora aponta para um objeto caminhao

Caminhao *trp;

trp = reinterpret_cast<Cminhao *>(vp);
cout << "Faça: " << trp->name() << endl;
```

Aqui a penúltima e a última adjudicações fazem um cast em 'Veículo *variável' para um 'Caminhao *'. Como é o caso em geral com os tipos de casts, este código não está livre de riscos: Só funcionará se 'vp' aponta realmente para um 'Caminhao'. De outra forma o programa pode se comportar de forma inesperada.

Capítulo 14: Polimorfismo

Como vimos no Capítulo 13, a linguagem C++ possui ferramentas para derivar classes de classes de base e usar ponteiros para endereçar objetos derivados. Também vimos que ao usar um ponteiro de uma classe de base para endereçar um objeto de uma classe derivada, o tipo do ponteiro determina qual função membro será usada. Isto significa que um 'Veículo *vp', apontando para um objeto 'Caminhao', computará incorretamente o peso combinado do caminhão numa adjudicação como 'vp -> peso()'. A razão para tal é que 'vp' chama 'Veículo::peso()' e não 'Caminhao::peso()', mesmo que 'vp' esteja apontando para 'Cminhao'.

Afortunadamente, existe um remédio. Em C++ um 'Veículo *vp' pode chamar uma função 'Caminhao::peso()' quando o ponteiro aponta para 'Caminhao'.

A terminologia para esta característica é polimorfismo: É como se o ponteiro 'vp' muda seu tipo de uma classe de base a um ponteiro para a classe à qual aponta. Assim, 'vp' se comporta como um 'Caminhao *' quando aponte para um objeto 'Caminhao' e como 'Carro *' quando aponte para um 'Carro' etc.. (num dos filmes StarTrek, o Capitão Kirk estava em apuros, como usual. Ele encontrou uma dama extremamente linda que, contudo, depois se transformou num medonho trol. Kirk estava surpreso, mas a dama lhe disse: “Você não sabia que sou polimórfica?”

O polimorfismo é realizado por uma característica chamada depois de “binding” (via de união). É assim chamada devido que a decisão de qual função chamar (uma função da classe de base ou uma função de uma classe derivada) não pode ser feita em tempo de compilação, mas é pós-posta até a execução do programa: Somente então é determinado qual função membro será, então, chamada.

14.1: Funções Virtuais

O comportamento padrão de ativação de uma função membro através de um apontador ou referência é pelo tipo de ponteiro (ou referência), este determina qual função chamar. P.ex, um 'Veículo *' ativará a função membro de 'Vaículo', mesmo quando apontando para um objeto de uma classe derivada. Isto está referenciado anteriormente ou chamado “binding” estático, já que a função é conhecida em tempo de compilação. O seguinte, conhecido como “binding” dinâmico á conseguido em C++ usando-se funções membro virtuais.

Uma função se torna uma função membro virtual quando sua declaração começa pela palavra

chave 'virtual'. Uma vez que uma função é declarada virtual na classe de base, ela permanece uma função membro virtual em todas as classes derivadas; mesmo que a palavra virtual não seja repetida na classe derivada.

No concernente ao sistema de classificação de 'Veículo' (veja seção 13.1) as duas funções membro 'peso()' e 'setPeso()' bem poderiam ser declaradas virtuais. As seções relevantes das definições de classe de 'Veículo' e 'Caminhao' são mostradas abaixo. Mostramos, também, a implantação das funções membro 'peso()' das duas classes:

```
class Veículo
{
    public:
        virtual int peso() const;
        virtual void setPeso(int ps);
};

class Caminhao: public Veículo
{
    public:
        void setPeso(int cavalo_ps, int reboque_ps);
        int peso() const;
};

int Veículo::peso() const
{
    return (peso);
}

int Caminhao::peso() const
{
    return (Carro::peso() + reboque_ps);
}
```

Note que a palavra chave 'virtual' só necessita aparecer na classe de base 'Veículo'. Não há necessidade (mas não é um erro) repeti-la nas classes derivadas: Uma vez virtual, sempre virtual. Por outro lado, uma função pode ser declarada virtual em qualquer parte na hierarquia de classe: O compilador estará feliz se 'peso()' for declarado virtual em 'Carro', antes que em 'Veículo'. As características específicas das funções membro virtuais então, para a função membro 'peso()', só aparecerão nos ponteiros ou referências de 'Carro' (e suas classes derivadas). Com um apontador de 'Veículo', “binding” estático continua em uso. Esse efeito está ilustrado abaixo:

```
Veículo v(1200);           // veículo com peso 1200
Caminhao t(6000, 115,      // caminhão com cabina pesando 6000, velocidade
115,
    "Scania", 15000);      // faz Scania, reboque com peso 15000
Vehicle *vp;               // apontador genérico de veículo

int main()
```



```

{
    vp = &v;                                     // veja (1) abaixo
    cout << vp->peso() << endl;

    vp = &t;                                     // veja (2) abaixo
    cout << vp->peso() << endl;

    cout << vp->velocidade() << endl;    // veja (3) abaixo
}

```

Como a função 'peso()' foi definida como virtual, “binding” é usado:

- Em (1) 'Veículo::peso()' é chamada;
- Em (2) 'Caminhao::peso()' é chamada;
- Em (3) É gerado um erro sintático. O membro 'velocidade()' não é membro de 'Veículo' e portanto não pode ser chamada via um 'Veículo *'.

O exemplo ilustra que quando um apontador a uma classe é usado somente funções membro dessa classe podem ser chamadas. Essas funções podem ser virtuais. Contudo, isto só influencia o tipo de “binding” (estático ou dinâmico) e não o conjunto de funções membro visível ao ponteiro.

Uma função membro virtual não pode ser uma função estática: Uma função membro virtual é ainda uma função membro ordinária no que possui um apontador 'this'. Uma função membro estática não possui apontador 'this', não pode ser declarada virtual.

14.2: Destrutores Virtuais

Quando o operador 'delete' libera memória ocupada por objetos alocados dinamicamente ou quando um objeto sai do escopo, o destrutor apropriado é chamado, para assegurar que a memória ocupada pelo objeto seja liberada. Agora, considere o seguinte fragmento de código (cf. seção 13.1):

```

Veículo *vp = new Terra(1000, 120);
delete vp;           // objeto destruído

```

Neste exemplo um objeto de uma classe derivada ('Terra') é destruído usando um ponteiro de uma classe de base ('Veículo *'). Para a definição de uma classe ‘estandarte’ isto significa que o destrutor de 'Veículo' será chamado, no lugar do de 'Terra'. Isto não só significa um vazamento de memória, já que a memória foi alocada em 'Terra', mas também impede qualquer outra tarefa, normalmente feita pelo destrutor da classe derivada de ser completada (ou iniciada). Coisa ruim.

Na linguagem C++ este problema é resolvido usando-se destrutores virtuais. Aplicando-se a

palavra chave 'virtual' à declaração de um destrutor o destrutor apropriado da classe derivada é ativado quando o argumento do operador 'delete' for um ponteiro da classe de base. Na seguinte definição parcial de classe a declaração de tal destrutor é mostrada:

```
class Veículo
{
    public:
        virtual ~Veículo();
        virtual size_t peso() const;
};
```

Declarando-se um destrutor virtual, a operação 'delete' ('delete vp') chamará o destrutor correto de 'Terra', no lugar do destrutor de 'Veículo'.

Da discussão acima podemos formular as situações onde deve-se definir um destrutor:

- Um destrutor deve ser definido quando for alocada e gerenciada memória por objetos da classe;
- Deve ser definido um destrutor virtual se a classe contém pelo menos uma função membro virtual;

No segundo caso o destrutor não tem qualquer tarefa especial a realizar. Nesses casos o destrutor virtual tem o corpo vazio. Por exemplo, a definição de 'Veículo::~~Veículo()' é tão simples como:

```
Veículo::~~Veículo()
{ }
```

Freqüentemente isto será parte da interface de classe como destrutor em linha.

14.3: Funções Virtuais Puras

Até aqui a classe de base 'Veículo' contém a implantação das funções virtuais 'peso()' e 'setPeso()'. Em C++ também é possível somente mencionar um membro virtual numa classe de base, sem defini-lo. A função é concretamente implantada numa classe derivada. Esta solução define um protocolo, que tem que ser seguido nas classes derivadas. Isto implica que as classes derivadas têm que cuidar da definição: o compilador C++ não permite a definição de um objeto de uma classe onde uma ou mais funções membro estejam indefinidas. A classe de base, assim, força um protocolo declarando uma função pelo seu nome, valores de retorno e argumentos. As classes derivadas têm que cuidar da implantação. A classe de base define, portanto, só o modelo ou molde a ser usado quando outras classes são derivadas. Tais classes de base são também chamadas de classes abstratas. As classes de base abstratas são a fundação de muitos padrões de projetos (cf. Gamma et al. (1995)), permitindo ao programador criar

softwares altamente re-usáveis. Alguns desse padrões de projeto estão cobertos pelas Anotações C++ (p.ex., Método de Modelagem na seção 20.3), mas para uma discussão completa de Padrões de Projeto o leitor deve consultar o livro Gamma et al..

As funções que só são declaradas na classe de base são chamadas de funções virtuais puras. Uma função se faz virtual pura prefixando-a com a palavra chave 'virtual' em sua declaração e pós-fixando-a com '= 0'. Um exemplo de uma função virtual pura ocorre na seguinte listagem, onde a definição da classe 'Object' requer a implantação do operador de conversão 'string()':

```
#include <string>

class Object
{
    public:
        virtual operator std::string() const = 0;
};
```

Agora, todas as classes derivadas de 'Object' têm que implantar o operador 'string()' como função membro ou seus objetos não poderão ser construídos. Assim é elegante: Todos os objetos derivados de 'Object' podem agora serem considerados objetos 'string', portanto podem, p.ex., serem inseridos em objetos 'ostream'.

Pode o destrutor virtual de uma classe de base ser uma função virtual pura? A resposta é não: Uma classe como 'Veículo' não requer classes derivadas para definir um destrutor. Em contraste, 'Object::operator string()' pode ser uma função virtual pura: Neste caso a classe de base define um protocolo ao qual se deve aderir.

Imagine que ocorreria se pudéssemos definir o destrutor de uma classe de base como virtual puro: De acordo com o compilador, o objeto da classe derivada pode ser construído: Já que seu destrutor está definido, a classe derivada não é uma classe abstrata pura. Contudo, dentro da classe derivada o destrutor da classe de base é chamado implicitamente. Este destrutor nunca foi definido e o linkador encontrará uma referência indefinida a, p.ex., 'Virtual::~Virtual()'.

Freqüentemente, mas não necessariamente sempre, funções membro virtuais puras são funções membro constantes. Isto permite a construção de objetos derivados constantes. Em outras situações isto não será necessário (ou realista) e funções membro não constantes podem ser requeridas. A regra geral para funções membro constantes se aplica também a funções virtuais puras: Se a função membro altera seu membro de dados, não pode ser uma função membro constante. Freqüentemente classes de base abstratas não possuem membros de dados. Contudo, o protótipo da função membro virtual pura deve ser usado outra vez nas classes derivadas. Se a implantação de uma função virtual pura numa classe derivada altera os dados do objeto derivado, então essa função não pode ser declarada constante. Por isso, o construtor de uma classe de base abstrata deve considerar se uma função membro

virtual pura pode ser constante ou não.

14.3.1: Implementação de Funções Virtuais Puras

Funções membro virtuais puras podem ser implementadas. Para implementar uma função membro virtual pura virtual: função membro virtual pura implementada é provida com a especificação normal `=0;`, isto a implementa e nada menos. Como `=0;` acaba em ponto e vírgula, o membro virtual puro é sempre quando muito uma declaração em sua classe, mas uma implementação pode ser feita `inline` abaixo da interface de classe ou pode ser definida como membro num arquivo fonte próprio.

Funções membro virtuais puras podem ser chamadas desde objetos de classes derivadas ou desde sua classe ou desde membros de classes derivadas especificando-se a classe de base e o operador de resolução de escopo com a função a ser chamada. O pequeno programa que segue mostra alguns exemplos:

```
#include <iostream>

class Base
{
    public:
        virtual ~Base();
        virtual void pure() = 0;
};

inline Base::~~Base()
{}

inline void Base::pure()
{
    std::cout << "Base::pure() called\n";
}

class Derived: public Base
{
    public:
        virtual void pure();
};

inline void Derived::pure()
{
    Base::pure();
    std::cout << "Derived::pure() called\n";
}

int main()
{
    Derived derived;
```

```

    derived.pure();
    derived.Base::pure();

    Derived *dp = &derived;

    dp->pure();
    dp->Base::pure();
}
// Saída:
//      Base::pure() called
//      Derived::pure() called
//      Base::pure() called
//      Base::pure() called
//      Derived::pure() called
//      Base::pure() called

```

A implementação de funções virtuais puras tem uso limitado. Podemos argumentar que a implementação de funções virtuais puras pode ser usada para realizar tarefas que já estão definidas no nível da classe de base. Contudo, não há garantia de que a função virtual da classe de base de fato será chamada de uma classe derivada desprezando a versão da função membro (como o construtor da classe de base que automaticamente é chamado do construtor uma classe derivada). Como a implementação da classe de base quando muito será chamada opcionalmente sua funcionalidade pode ser também implementada num membro separado, que então pode ser chamado sem o requerimento de mencionar a classe de base explicitamente.

14.4: Funções Virtuais em Herança Múltipla

Como foi mencionado no Capítulo 13 uma classe pode derivar de muitas classes. Tal classe derivada herda as propriedades de todas suas classes de base. Claro está que as classes de base podem ser derivadas de outras classes mais acima na hierarquia.

Considere que aconteceria se mais que uma vez uma classe de base estivesse na via de herança. Isto está ilustrado no código de exemplo abaixo: Uma classe 'Derived' é duplamente derivada de uma classe 'Base':

```

class Base
{
    int d_field;
public:
    void setfield(int val)
        { d_field = val; }
    int field() const
        { return d_field; }
};

```

```
class Derived: public Base, public Base
{
};
```

Devido à dupla derivação a funcionalidade de 'Base' ocorre duas vezes em 'Derived'. Isto conduz a ambigüidades: Quando a função 'setfield()' é chamada para um objeto 'Derived', qual função seria, já que existem duas? Em tal derivação duplicada, o compilador C++ normalmente se recusa a gerar o código e (corretamente) identifica um erro.

O código acima claramente duplica sua classe de base. Tal caso facilmente pode ser evitado. Mas a duplicação de uma classe de base pode também ocorrer através de herança aninhada, onde um objeto é derivado de, p.ex., 'Carro' e de 'Aéreo' (veja o sistema de classificação de veículo na seção 13.1). Tal classe necessitaria representar, p.ex, um carro voador (tal como o de James Bond vs. o Homem da Arma de Ouro. ...). Um 'CarroAéreo' conteria dois 'Veículo' e portanto duas vezes o campo 'peso', duas funções 'setPeso()' e duas funções 'peso()'.

14.4.1: Ambigüidade em Herança Múltipla

Investiguemos mais de perto porque um 'CarroAéreo' introduz ambigüidade, quando deriva de 'Carro' e 'Aéreo'.

- Um 'CarroAéreo' é um 'Carro', enquanto 'Terra' e enquanto 'Veículo';
- Contudo, um 'Carro Aéreo' é também 'Aéreo', enquanto 'Veículo'.

A duplicação dos dados de 'Veículo' é ilustrada na Figura 13.

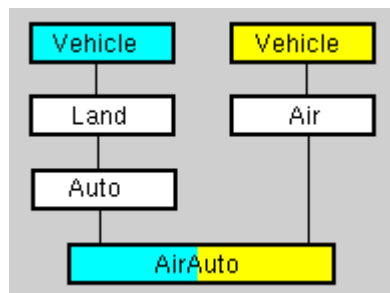


Figura 13(Duplicação de uma classe de base em derivação múltipla.)

A organização interna de um 'CarroAéreo' é mostrada na Figura 14

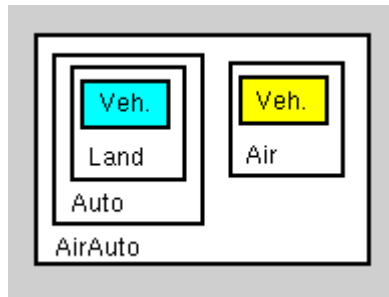


Figura 14(Organização interna de um objeto 'CarroAéreo'.)

O compilador C++ detetará a ambigüidade no objeto 'CarroAéreo' e falhará na compilação de adjudicações como:

```
CarroAéreo cool;  
  
cout << cool.peso() << endl;
```

A questão de qual função membro 'peso()' deve ser chamada não pode ser respondida pelo compilador. O programador tem duas possibilidades de resolver a ambigüidade explicitamente:

- Primeiro a chamada à função onde ocorre a ambigüidade pode ser modificada. A ambigüidade é resolvida usando-se o operador de resolução de escopo:

```
// esperamos que o peso seja mantido no Carro  
// como parte do objeto..  
cout << cool.Carro::peso() << endl;
```

Note a posição do operador de resolução de escopo e o nome da classe: Antes do nome da função membro.

- Segundo, pode-se criar uma função 'peso()' dedicada para a classe 'CarroAéreo':

```
int CarroAéreo::peso() const  
{  
    return Carro::peso();  
}
```

A segunda possibilidade das duas é preferível, já que libera o programador que use da classe 'CarroAéreo' de precauções especiais.

Contudo, aparte destas soluções explícitas, existe uma mais elegante, discutida na seção seguinte.

14.4.2: Classes de Base Virtuais (Derivação Virtual)

Como está ilustrado na Figura 14, um 'CarroAéreo' representa dois 'Veículos'. O resultado não é só ambigüidade nas funções que acessa os dados de peso, mas também a presença de dois campos de 'peso'. Isto é algo redundante, pois podemos assumir que um 'CarroAéreo' tem um só peso.

Podemos concluir que 'CarroAéreo' é um só 'Veículo', apesar de usar derivação múltipla. Isto é feito definindo a classe de base que está mencionada mais que uma vez numa classe derivada como uma classe de base virtual.

Para a classe 'CarroAéreo' isto significa que a derivação de 'Terra' e 'Aéreo' é mudada:

```
class Terra: virtual public Veículo
{
    // etc
};

class Carro: public Terra
{
    // etc
};

class Aéreo: virtual public Veículo
{
    // etc
};

class CarroAéreo: public Carro, public Aéreo
{
};
```

A derivação virtual assegura que através de 'Terra', um 'Veículo' só é adicionado a uma classe quando uma classe de base virtual não está presente. O mesmo é válido para 'Aéreo'. Isto significa que não podemos saber através de que via um 'CarroAéreo' é parte de 'Veículo'; somente podemos dizer que existe um objeto 'Veículo' incluído. A organização interna de um 'CarroAéreo' depois de uma derivação virtual é mostrada na Figura 15.

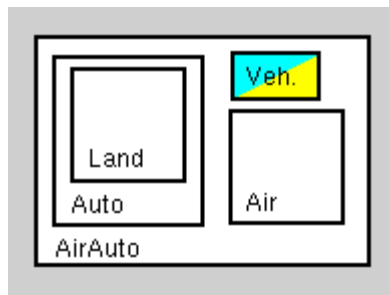


Figure 15 (Organização interna de um objeto 'CarroAéreo' quando as classes de base são virtuais.)

Note o seguinte:

- Quando as classes de base de uma classe que usa derivação múltipla são virtuais, derivadas de uma classe de base virtual (como acima), o construtor da classe de base, normalmente chamado quando o construtor da classe derivada é chamado, já não é usado: O seu iniciador da classe de base é ignorado. O construtor da classe de base será chamado independentemente dos construtores da classe derivada. Assuma que temos duas classes, 'Derived1' e 'Derived2', ambas (possivelmente virtuais) derivadas de 'Base'. Perguntamos quais construtores serão chamados quando uma 'class Final: public Derived1, public Derived2' é definida? Para distinguir os diferentes construtores envolvidos, usaremos 'Base1()' para indicar o construtor da classe 'Base' que é chamado como iniciador da classe de base para 'Derived1' (e analogamente: 'Base2()' para 'Derived2'), enquanto 'Base()' indica o construtor padrão da classe 'Base'. Além do construtor da classe 'Base', usaremos 'Derived1()' e 'Derived2()' para indicar o iniciador da classe de base para a classe 'Final: public Derived1, public Derived2':

- classes:

```
Derived1: public Base
Derived2: public Base
```

Esta é a forma normal para derivação múltipla. Existem duas classes 'Base' no objeto 'Final' e os seguintes construtores serão chamados (na ordem mencionada):

```
Base1(),
Derived1(),
Base2(),
Derived2()
```

- classes:

```
Derived1: public Base
Derived2: virtual public Base
```

Somente 'Derived2' usa derivação virtual. Para 'Derived2' o iniciador de classe de base será omitido e o construtor padrão da classe de base será chamado. Ainda mais, este construtor `separado` da classe de base será chamado primeiro:

```
Base() ,
Base1() ,
Derived1() ,
Derived2()
```

Note que 'Base()' é chamado primeiro, não 'Base1()'. Note também que só uma classe derivada usa derivação virtual, existem ainda dois objetos da classe 'Base' na eventual classe 'Final'. A fusão das classes de base somente ocorre com classes de base múltiplas virtuais.

- classes:

```
Derived1: virtual public Base
Derived2: public Base
```

Somente 'Derived1' usa derivação virtual. Para 'Derived1' o iniciador da classe de base será omitido e o construtor padrão da classe de base será chamado. Note a diferença com o primeiro caso: 'Base1()' é substituído por 'Base()'. Mesmo 'Derived1' usando o construtor padrão de 'Base', não há diferença com o primeiro caso:

```
Base() ,
Derived1() ,
Base2() ,
Derived2()
```

- classes:

```
Derived1: virtual public Base
Derived2: virtual public Base
```

Aqui ambas classes derivadas usam derivação virtual e portanto só um objeto da classe de base estará presente na classe 'Final'. Note que somente um construtor da classe de base é chamado: Para o objeto da classe 'Base' fundido:

```
Base() ,
Derived1() ,
Derived2()
```

- A derivação virtual, em contraste com as funções virtuais, é um fato puro do tempo de compilação: Se uma derivação é virtual ou não define como o compilador constroi a definição de uma classe a partir de outras classes.

Sumarizando, o uso da derivação virtual evita ambigüidade quando as funções membro de uma classe de base são chamadas. Ainda mais, evita-se a duplicação de membros de dados.

14.4.3: Quando a derivação virtual não é apropriada

Em contraste com a definição anterior de uma classe como 'CarroAéreo', podem haver situações onde a dupla presença de membros de uma classe de base seja apropriada. Para ilustrar isto, considere a definição de um 'Caminhao' da seção 13.4:

```
class Caminhao: public Carro
{
    int d_peso_reboque;

public:
    Caminhao();
    Caminhao(int cavalo_ps, int sp, char const *nm,
              int reboque_ps);

    void setPeso(int cavalo_ps, int reboque_ps);
    int peso() const;
};

Caminhao::Caminhao(int cavalo_ps, int sp, char const *nm,
                   int reboque_ps)
:
    Carro(cavalo_ps, sp, nm)
{
    d_reboque_peso = reboque_ps;
}

int Caminhao::peso() const
{
    return
        Carro::peso() +      // soma de:
        reboque_ps;         // cavalo mecânico mais
                           // reboque
}
```

Esta definição mostra como um 'Caminhao' é construído para conter dois campos de peso: Um via sua derivação de 'Carro' e um via seu membro de dados 'int d_reboque_peso'. Tal definição é, claro está, válida, mas poderia ser re-escrita. Poderíamos derivar 'Caminhao' de 'Carro' e de 'Veículo', assim, requisitando a dupla presença de 'Veículo'; Uma para o peso do cavalo mecânico e uma para o peso do reboque.

Um pequeno ponto de interesse aqui é que uma derivação como:

```
class Caminhao: public Carro, public Veículo
```

Não é aceitável pelo compilador C++: Um 'Veículo' já é parte de um Carro. Uma classe intermediária resolve este problema: Derivamos a classe 'ReboqueVei' e 'Caminhao' de 'Carro' e de 'ReboqueVei'. Todas as ambigüidades concernentes às funções membro estão resolvidas para a classe 'Caminhao':

```
class ReboqueVei: public Veículo
{
    public:
        ReboqueVei(int ps)
        :
            Veículo(ps)
        {}
};

class Caminhao: public Carro, public ReboqueVei
{
    public:
        Caminhao();
        Caminhao::Caminhao(int cavalo_ps, int sp, char const *nm,
                           int reboque_ps)
        :
            Carro(cavalo_ps, sp, nm),
            ReboqueVei(reboque_ps)
        {}
        void setPeso(int cavalo_ps, int reboque_ps);
        int peso() const;
};

int Caminhao::peso() const
{
    return
        Carro::peso() +           // soma de:
        ReboqueVei::peso();      // cavalo mecânico mais
                                // reboque
}
```

14.5: Identificação de Tipo em Tempo de Execução

A linguagem C++ oferece dois caminhos para examinar o tipo de objetos e expressões enquanto o programa está sendo executado. As possibilidades da C++ são limitadas comparadas com as de linguagens como Java. Normalmente C++ usa exame e identificação estáticos de tipo. O exame e identificação estáticos de tipo é possivelmente mais seguro e certamente mais eficiente que em tempo de execução e portanto sempre que possível. Apesar disso, a C++ oferece identificação de tipo em tempo de execução, fornecendo os operadores “cast” dinâmico e 'typeid'.

- O operador 'dynamic_cast<>()' pode ser usado para converter um ponteiro ou referência de uma classe de base a um ponteiro ou referência de uma classe derivada. Isto é chamado “down-

casting”;

- O operador 'typeid' retorna o tipo de uma expressão.

Estes operadores operam sobre tipos de objetos de classes contendo pelo menos uma função membro virtual.

14.5.1: O operador “dynamic_cast”

O operador 'dynamic_cast<>()' é usado para converter um ponteiro ou referência da uma classe de base em, respectivamente, um ponteiro ou referência de uma classe derivada.

Um “dynamic cast” é feito durante a execução. Um pré-requisito para usar o operador “dynamic cast” é a existência de pelo menos uma função virtual na classe de base.

No seguinte exemplo um ponteiro a uma classe 'Derived' é obtido do ponteiro da classe 'Base' 'bp':

```
class Base
{
    public:
        virtual ~Base();
};

class Derived: public Base
{
    public:
        char const *toString()
        {
            return "Derived object";
        }
};

int main()
{
    Base *bp;
    Derived *dp,
    Derived d;

    bp = &d;

    dp = dynamic_cast<Derived *>(bp);

    if (dp)
        cout << dp->toString() << endl;
```

```

        else
            cout << "A conversão do "dynamic cast" falhou\n";
    }

```

Note o teste: Na condição 'if' o sucesso do “dynamic cast” é examinado. Isto é feito na execução, já que o compilador não tem condições de fazê-lo. Se um ponteiro da classe de base está presente o operador “dynamic cast” retorna 0 em falha e um apontador à classe requerida em sucesso. Conseqüentemente, se há múltiplas classes derivadas, uma série de exames deve ser feito para se saber para qual classe derivada o apontador aponta (No exemplo seguinte as classes derivadas são só declaradas):

```

class Base
{
    public:
        virtual ~Base();
};
class Derived1: public Base;
class Derived2: public Base;

int main()
{
    Base *bp;
    Derived1 *d1,
    Derived1 d;
    Derived2 *d2;

    bp = &d;

    if ((d1 = dynamic_cast<Derived1 *>(bp)))
        cout << *d1 << endl;
    else if ((d2 = dynamic_cast<Derived2 *>(bp)))
        cout << *d2 << endl;
}

```

Alternativamente, uma referência a uma classe de base pode estar presente. Neste caso o operador 'dynamic_cast<>()' lança uma exceção se falha. Por exemplo:

Neste exemplo o valor 'std::bad_cast' é introduzido. A exceção 'std::bad_cast' é lançada se a referência a um objeto de uma classe derivada falha.

Note a forma da cláusula do 'catch': 'bad_cast' é o nome de um tipo. Na seção 16.4.1 a construção de tal tipo é discutida.

O operador “dynamic cast” é útil quando uma classe de base não pode ou não deve ser modificada (p.ex., quando as fontes não estão disponíveis) e uma classe derivada pode ser modificada. O código que recebe um ponteiro ou referência a uma classe de base pode então realizar um “dynamic cast” para a classe derivada para acessar a funcionalidade da classe derivada.

Os “casts” de um ponteiro ou uma referência de uma classe de base para um ponteiro ou referência a uma classe derivada são chamados “downcasts”.

Pode-se diferenciar um “dynamic cast” de um 'reinterpret_cast' bem. Claro que o “dynamic_cast” pode ser usado em referências e 'reinterpret_cast' somente em ponteiros. Mas qual é a diferença se ambos argumentos são ponteiros?

Quando o 'reinterpret_cast' é usado, dizemos ao compilador que literalmente deve re-interpretar um bloco de memória como algo diferente. Um exemplo bem conhecido é obtido ao se acessar os bytes individuais de um inteiro. Um inteiro consiste de 'sizeof(int)' bytes e estes bytes podem ser acessados re-interpretando o local do valor inteiro como 'char *'. Ao usarmos o 'reinterpret_cast' o compilador não oferece absolutamente uma salvaguarda. O compilador feliz re-interpreta um 'int *' como um 'double *', mas o resultado produz, afinal, um valor sem significado.

O 'dynamic_cast' também re-interpreta um bloco de memória como algo diferente, mas aqui uma salvaguarda na execução é oferecida. O 'dynamic_cast' falha quando o tipo requerido não combina com o tipo do objeto apontado. O propósito do 'dynamic_cast' também é muito mais reduzido que o do 'reinterpret_cast', já que só pode ser usado para ‘downcast’ de classes derivadas com membros virtuais.

14.5.2: O Operador ‘typeid’

Como com o operador 'dynamic_cast<>()', o 'typeid' é usualmente aplicado para objetos da classe de base, numa classe derivada. Similarmente, a classe de base deve conter uma ou mais funções virtuais.

Para usar o operador 'typeid', o arquivo fonte deve conter:

```
#include <typeinfo>
```

O operador 'typeid' retorna um objeto do tipo 'type_info', que, p.ex., pode ser comparado a outros objetos 'type_info'.

A classe 'type_info' pode ser implantada de diferentes maneiras em diferentes compiladores, mas, afinal, tem a seguinte interface:

```
class type_info
{
    public:
        virtual ~type_info();
        int operator==(const type_info &other) const;
        int operator!=(const type_info &other) const;
        char const *name() const;
```

```

private:
    type_info(type_info const &other);
    type_info &operator=(type_info const &other);
};

```

Note que esta classe tem um construtor de cópias privado e um operador de adjudicação sobrecarregado. Isto evita a construção e adjudicação normal de objetos 'type_info'. Os objetos 'type_info' são construídos e retornados pelo operador 'typeid'. As implantações, contudo, podem escolher em estender ou elaborar a classe 'type_info' e fornecer, p.ex., listas de funções que podem ser chamadas com certa classe.

Se é dada uma referência a 'typeid' de uma classe de base (que contenha pelo menos uma função virtual), ele indicará que o tipo de seu operando é da classe derivada. Por exemplo:

```

class Base;      // contém pelo menos uma função virtual
class Derived: public Base;

Derived d;
Base    &br = d;

cout << typeid(br).name() << endl;

```

Neste exemplo ao operador 'typeid' é dada uma referência da classe de base. Ele imprimirá o texto “Derived”, sendo o nome da classe de 'br'. Se 'Base' não contém funções virtuais, o texto “Base” seria impresso.

O operador 'typeid' pode ser usado para determinar o nome dos tipos de expressões, não só do tipo de classe de objetos. Por exemplo:

```

cout << typeid(12).name() << endl;      // imprime: int
cout << typeid(12.23).name() << endl;   // imprime: double

```

Note, contudo, que o exemplo acima é sugestivo, como imprime os tipos. Podem ser inteiros ou duplos, mas este não é necessariamente o caso. Se é requerida portabilidade, assegure-se de não haver testes em dados estáticos, como os acima e cadeias de caracteres. Examine o que seu compilador produz em caso de dúvidas.

Onde o operador 'typeid' é usado para determinar o tipo de uma classe derivada, é importante referenciar uma classe de base usada como argumento do operador 'typeid'. Considere o seguinte exemplo:

```

class Base;      // contém pelo menos uma função virtual
class Derived: public Base;

Base *bp = new Derived; // ponteiro de uma classe de base a um objeto derivado

if (typeid(bp) == typeid(Derived *))    // 1: falso
    ...

```



```

if (typeid(bp) == typeid(Base *))      // 2: verdadeiro
    ...
if (typeid(bp) == typeid(Derived))     // 3: falso
    ...
if (typeid(bp) == typeid(Base))       // 4: falso
    ...
if (typeid(*bp) == typeid(Derived))    // 5: verdadeiro
    ...
if (typeid(*bp) == typeid(Base))      // 6: falso
    ...

Base &br = *bp;

if (typeid(br) == typeid(Derived))     // 7: verdadeiro
    ...
if (typeid(br) == typeid(Base))       // 8: falso
    ...

```

Aqui, (1) retorna falso, como 'Base *' não é um 'Derived *', (2) retorna verdadeiro, os dois ponteiro são do mesmo tipo, (3) e (4) retornam falso os objetos ponteiros não são os próprios objetos.

Por outro lado, se '*bp' for usado nas expressões acima, então (1) e (2) retornam falso, já que um objeto ou sua referência não é um ponteiro, mas (5) retorna verdadeiro: '*bp' se refere a um objeto da classe 'Derived' e 'typeid (*bp)' retornará 'typeid (Derived)'. Um resultado similar é obtido se uma referência a uma classe de base é usada: 7 retorna verdadeiro e 8 falso.

Quando um apontador nulo é passado a um operador 'typeid' é lançada uma exceção 'bad_typeid'.

14.6: Derivando classes de 'streambuf'

A classe 'streambuf' (veja seção 5.7 e Figura 4) tem muitas funções membro (protegidas) virtuais (veja seção 5.7.1) usadas pelas classes 'stream' que usam objetos 'streambuf'. Derivando uma classe de 'streambuf' essas funções membro podem ser sobre passadas nas classes derivadas, implantando, assim, uma especialização da classe 'streambuf' nas quais se pode usar objetos padrão 'istream' e 'ostream'.

Basicamente um 'streambuf' interfaceia algum dispositivo. O comportamento normal dos objetos da classe 'stream' permanece inalterado. Portanto, a extração de uma 'string' de um objeto 'streambuf' retornará uma sequência consecutiva de caracteres não delimitada por um espaço em branco. Se a classe derivada é usada para operações de entrada, as seguintes funções membro são sérias candidatas a serem desprezadas. Mais adiante nesta seção daremos exemplos onde algumas destas funções são desprezadas:

- `'int streambuf::pbackfail (intc)'`: Este membro é chamado quando:

- `gptr() == 0`: sem uso de buferização;
- `gptr() == eback()`: falta lugar para devolver;
- `gptr() != c`: um caracter diferente ao próximo a ser lido tem que ser devolvido.

Se `'c == endOfFile()'` então tem que se eliminar um caracter do dispositivo de entrada, do contrário `'c'` tem que ser repostos aos caracteres a serem lidos. A função retorna EOF quando em falha. Do contrário retorna 0. A função é chamada sob falha de retorno de um caracter.

- `'streamsize streambuf::showmanyc()'`: Este membro tem que retornar um limite inferior garantido como número de caracteres que podem ser lidos do dispositivo antes de `'uflow()'` ou `'underflow()'` retornar EOF. Como padrão é retornado 0 (significa que pelo menos 0 caracteres serão retornados antes das duas funções retornarem EOF). Quando um valor positivo é retornado então a próxima chamada a `'u(nder)flow()'` não retornará EOF.
- `'int streambuf::uflow()'`: como padrão esta função chama `'underflow()'`. Se `'underflow()'` falha, EOF é retornado. Do contrário, o próximo caracter é retornado como `'*gptr()'` seguido de um `'gbump(-1)'`. O membro também move o caracter pendente, que é retornado, para a sequência de respaldo. É diferente de `'underflow()'`, que também retorna o próximo caracter, mas não altera a posição de entrada.
- `'int streambuf::underflow()'`: Este membro é chamado quando
 - não há bufer de entrada (`eback() == 0`)
 - `gptr() >= egptr()`: não há mais caracteres de entrada pendentes.

Retorna o próximo caracter de entrada, que é o caracter em `'gptr()'` ou o primeiro caracter do dispositivo de entrada.

Como este membro é eventualmente usado por outras funções membro para ler caracteres de um dispositivo, por último esta função membro deve ser desprezada em novas classes derivadas de `'streambuf'`.

- `'streamsize streambuf::xsgetn(char *buffer, streamsize n)'`: Esta função membro atua como se retorna valores de `n 'snext()'` estivesse disposto em locais consecutivos do bufer. Se for retornado

EOF a leitura para. O número de caracteres lidos é retornado. Desprezando as versões pode otimizar o processo de leitura com, p.ex., acessando diretamente o bufer de entrada.

Quando a classe derivada é usada para operações de saída, as seguintes funções membro devem ser consideradas:

- `'int streambuf::overflow(int c)'`: Esta função é chamada para escrever caracteres de seqüências pendentes no dispositivo de saída. A menos que `c` seja EOF é lógico pô-lo na seqüência pendente. Assim, se a seqüência pendente consiste dos caracteres `'h', 'e', 'l' e 'l'` e `c == 'o'`, então eventualmente será escrito `'hello'` no dispositivo de saída.

Como esta função membro é eventualmente usada por outra função membro para escrever num dispositivo, por último esta função membro deve ser desprezada para novas classes derivadas de `'streambuf'`.

- `'streamsize streambuf::xsputn(char const *buffer, streamsize n)'`: Esta função atua como se `'n'` lugares consecutivos do bufer fossem passados a `'sputc()'`. Se EOF for retornado então encerra a escritura. O número de caracteres escritos é retornado. Desprezar versões otimizaria o processo de escritura, p.ex., acessando diretamente o bufer.

Para classes derivadas que usem buffers e suportem operações de deslocamento, considerar as seguintes funções membro:

- `'streambuf *streambuf::setbuf(char *buffer, streamsize n)'`: É chamada por `'pubsetbuf()'`.
- `'pos_type streambuf::seekoff(off_type offset, ios::seekdir way, ios::openmode mode = ios::in | ios::out)'`: Chamada para anular a posição do próximo caracter a ser processado. É chamada por `'pubseekoff()'`. A nova posição ou posição inválida (p.ex., -1) é retornada.
- `'pos_type streambuf::seekpos(pos_type offset, ios::openmode mode = ios::in | ios::out)'`: atua de forma similar a `'seekoff()'`, mas opera com posições relativas.
- `'int sync()'`: Esta função membro descarrega todos os caracteres pendentes para o dispositivo, e/ou reseta um dispositivo de entrada para a posição de primeiro caracter pendente, esperando na entrada do bufer para ser consumido. Retorna 0 em sucesso e -1 em falha. Como o `'streambuf'` padrão não é buferizado, a implantação padrão também retorna 0.

Em seguida considere o seguinte problema, que será resolvido construindo-se uma classe `'CapsBuf'` derivada de `'streambuf'`. O problema é construir um `'streambuf'` que escreva sua informação na

saída padrão, de tal forma que todo espaço em branco, que limitam séries de caracteres passem a maiúscula. A classe 'CapsBuf' obviamente precisa desprezar 'overflow()' e um mínimo de sua percepção de seu estado. Seu estado muda de 'Passe a maiúscula' para 'Literal', como segue:

- O estado inicial é 'Passe a maiúscula';
- Muda para 'Passe a maiúscula' depois de processar um espaço;
- Muda para 'Literal' depois de processar um caracter não espaço.

Uma simples variável para lembrar o último caracter nos permite manter o controle do estado corrente. Como 'Passe a maiúscula' é similar a 'o último caracter processado é um espaço em branco' podemos simplesmente iniciar a variável com um espaço. Eis a definição inicial da classe 'CapsBuf':

```
#include <iostream>
#include <streambuf>
#include <ctype.h>

class CapsBuf: public std::streambuf
{
    int d_last;

public:
    CapsBuf()
    :
        d_last(' ')
    {}

protected:
    int overflow(int c)           // interface com o dispositivo.
    {
        std::cout.put(isspace(d_last) ? toupper(c) : c);
        return d_last = c;
    }
};
```

Um exemplo de um programa que usa 'CapsBuf' é:

```
#include "capsbuf1.h"
using namespace std;

int main()
{
    CapsBuf      cb;

    ostream      out(&cb);

    out << hex << "hello " << 32 << " worlds" << endl;
```

```

        return 0;
    }
    /*
        Saída gerada:

        Hello 20 Worlds
    */

```

Note o uso do operador de inserção, e note a conversão de radical (inserindo o valor hexadecimal 32 como os caracteres '2' e '0') isto é feito pelo objeto 'ostream'. O propósito real da classe 'CapsBuf' é transformar em maiúscula séries de caracteres ASCII e isso é o que faz bem.

A seguir mostramos que a inserção de caracteres numa 'stream' pode ser feita também por uma construção como:

```
cout << cin.rdbuf();
```

Ou, traduzindo à mesma coisa:

```
cin >> cout.rdbuf();
```

Tentando mostrar que o que intentamos na 'main()' acima pode-se fazer assim:

```
cin >> out.rdbuf();
```

Compilamos e linkamos o programa num executável 'caps' e:

```
echo hello world | caps
```

Infelizmente nada ocorre... Não temos nenhuma reação ao intentar a adjudicação 'cin >> cout.rdbuf()'. O que está mal aqui?

A diferença entre 'cout << cin.rdbuf()', que produz o resultado esperado e 'cin >> out.rdbuf()' é a função membro 'operator >> (streambuf *)' que realiza só uma cópia de 'streambuf' a 'streambuf' somente se os modos respectivos estão corretamente postos. Assim, o argumento do operador de extração tem que apontar a um 'streambuf' no qual a informação pode ser escrita. Como padrão, nenhum modo de 'stream' é setado para um objeto 'streambuf' pleno. Como não há construtor para 'streambuf' que aceite 'ios::openmode', forçamos o modo 'ios::out', definindo um bufer de saída usando 'setp()'. Isto é feito definindo um bufer, como não o usamos, deixamos seu tamanho em 0. Note que isto é algo diferente que usar argumento 0 com 'setp()', já que isto indica 'sem buferação', o que não altera a situação padrão. Contudo qualquer valor diferente de 0 poderia ser usado no rango '[begin, begin]', decidimos definir uma variável local 'char' no construtor e usar '[dummy, dummy]' para definir o bufer vazio. Isto define efetivamente 'CapsBuf' como um bufer de saída, ativando o membro:

```
istream::operator>>(streambuf *)
```

Como a variável vazia não é usada por 'setp()' pode ser definida como variável local. Seu único propósito é indicar a 'setp()' que não se usa buffer. Eis o construtor da classe 'CapsBuf' revisto:

```
CapsBuf::CapsBuf()  
{  
    d_last(' ');  
    char dummy;  
    setp(&dummy, &dummy);  
}
```

Now the program can use either

```
out << cin.rdbuf();
```

or:

```
cin >> out.rdbuf();
```

Agora o envólucro de 'ostream' não é necessário aqui:

```
cin >> &cb;
```

Prodizimos o mesmo resultado.

Não está claro se a solução proposta aqui com 'setp()' uma solução desageitada. Depois de tudo, o envoltório de 'ostream', 'cb' poderia informar a 'CapsBuf' para atuar como um 'streambuf' para realizar operações de saída?

14.7: Uma classe Polimórfica por Exceção

Antes, aqui nas Anotações (seção 8.3.1) aludimos à possibilidade de projetar uma classe 'Exception' cujo membro 'process()' se comportaria diferente, segundo o tipo de exceção lançada. Agora que introduzimos o polimorfismo, podemos desenvolver este exemplo.

Por agora, provavelmente, está claro que nossa classe 'Exception' deve ser uma classe de base virtual, através da qual classes especiais podem derivar para manipular exceções. Pode ser mesmo demonstrado que 'Exception' pode ser uma classe de base abstrata, com puras funções membro virtuais. Na discussão da seção 8.3.1 uma função membro 'severity()' foi mencionada que não seria uma candidata própria a função membro puramente abstrata, mas para esse membro podemos usar um operador 'dynamic_cast<>()' completamente generalizado.

```
dynamic_cast<>() operator.
```

A classe de base (abstrata) 'Exception' é como segue:

```

#ifndef _EXCEPTION_H_
#define _EXCEPTION_H_

#include <iostream>
#include <string>

class Exception
{
    friend std::ostream &operator<<(std::ostream &str, Exception const &e)
    {
        return str << e.operator std::string();
    }

    std::string d_reason;

public:
    virtual ~Exception()
    {}
    virtual void process() const = 0;
    virtual operator std::string() const
    {
        return d_reason;
    }
protected:
    Exception(char const *reason)
    :
        d_reason(reason)
    {}
};
#endif

```

O operador 'string()' substitui o membro toString() usado na seção 8.3.1. O operador amigo 'operator<<()' usa o operador 'string()' (virtual) assim somos capazes de inserir um objeto 'Exception' numa 'ostream'. Aparte disto, usa um destrutor virtual que não faz nada.

Uma classe derivada 'FatalException': pode, agora, ser definida como segue (usando 'process()' numa implantação bem básica):

```

#ifndef _FATALEXCEPTION_H_
#define _FATALEXCEPTION_H_

#include "exception.h"

class FatalException: public Exception
{
public:
    FatalException(char const *reason)
    :
        Exception(reason)

```

```

    {}
    void process() const
    {
        exit(1);
    }
};
#endif

```

O traslado do exemplo do fim da seção 8.3.1 para a situação atual pode agora ser feito facilmente (usando as classes derivadas 'WarningException' e 'MessageException'), construídas como 'FatalException':

```

#include <iostream>
#include "message.h"
#include "warning.h"
using namespace std;

void initialExceptionHandler(Exception const *e)
{
    cout << *e << endl;          // mostra o texto pleno da informação

    if
    (
        !dynamic_cast<MessageException const *>(e)
        &&
        !dynamic_cast<WarningException const *>(e)
    )
        throw;                    // Passa a outros tipos de Exceções

    e->process();                  // Processa uma mensagem ou uma chamada de atenção
    delete e;
}

```

14.8: Como o Polimorfismo é implantado

Esta seção descreve rapidamente como o polimorfismo é implantado em C++. Não é necessário entender como o polimorfismo é implantado se somente usamos esta característica. Contudo, pensamos que é bom saber como é possível o polimorfismo. Além disso a seguinte discussão explica porque existe um custo no polimorfismo em termos de uso da memória.

A idéia fundamental atrás do polimorfismo é que o compilador não sabe qual função chamar na compilação; a função apropriada será selecionada na execução. Isto significa que os endereços das funções precisam ser guardados em algum lugar, para serem vistos antes da chamada. Estes lugares devem ser acessíveis ao objeto em questão. P.ex., quando 'Veículo *vp' aponta para um objeto 'Caminhao', então 'vp->peso()' chama uma função membro de 'Caminhao'; o endereço desta função é determinado pelo objeto para o qual 'vp' aponta.

Uma implantação comum é a seguinte: Um objeto que contém uma função membro virtual tem como seu primeiro membro de dados um campo vazio, apontando a um conjunto de ponteiros com os endereços das funções membro virtuais. O membro de dados vazio usualmente é chamado de 'vpointer', o conjunto de endereços das funções membro virtuais de 'vtable'. Note que a discussão presente depende do compilador, e não faz parte da norma C++ da ANSI/ISO.

A tabela de endereços das funções virtuais é compartilhada por todos os objetos da classe. Muitas classes podem até compartilhar a mesma tabela. A sobrecarga em memória é, portanto:

- Um campo de ponteiro extra por objeto, que aponta para:
- uma tabela de ponteiros por classe (derivada) que armazena os endereços das funções virtuais das classes.

Conseqüentemente, um comando como 'vp->peso()' examina primeiro o membro de dados vazio do objeto apontado por 'vp'. No caso da classificação do sistema veículo, este membro de dados aponta para uma tabela de dois endereços: um aponta para a função 'peso()' e outro para 'setPeso()'. A função chamada é determinada por esta tabela.

A organização interna dos objetos com funções virtuais é ilustrada nas Figuras 16 e 17 (de Guillaume Caumon).

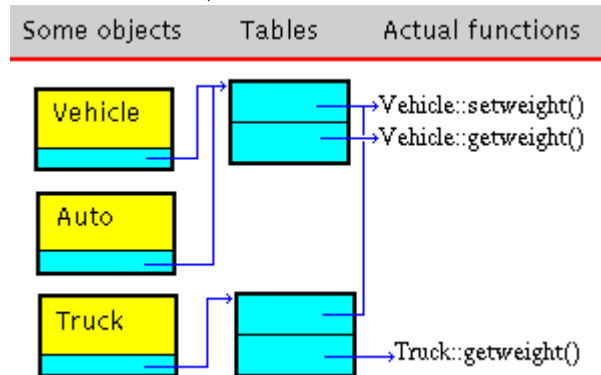


Figure 16 (Organização interna dos objetos quando são definidas funções virtuais .)

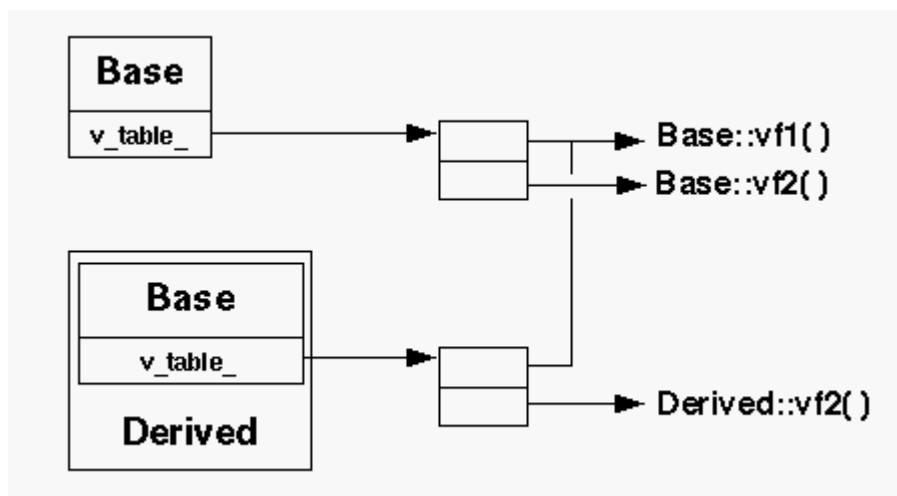


Figure 17 (Figura complementar, fornecida por Guillaume Caumon)

Como pode ser visto das figuras, todos objetos com funções virtuais precisam ter membro de dados (vazio) para endereçar uma tabela de ponteiros a funções. Os objetos das classes 'Veículos' e 'Carros' ambas endereçam a mesma tabela. A classe 'Caminhao', contudo, introduz sua própria versão de 'peso()': Portanto, esta classe, necessita sua própria tabela de apontadores a funções.

14.9: Referência a 'vtable' Indefinida

Ocasionalmente, o linkador reclamará com uma mensagem como a seguinte:

```
In function
  `Derived::Derived[in-charge]()':
: undefined reference to `vtable for Derived'
```

Este erro é causado pela ausência da implantação de uma função virtual numa classe derivada, enquanto a função é mencionada na interface da classe derivada.

Tal situação pode facilmente ser criada:

- Construa uma classe de base (completa), definindo uma função membro virtual;
- Construa uma classe derivada que mencione uma função virtual em sua interface;
- A função virtual na classe derivada desprezará a função da classe de base com mesmo nome, não é implantada. O compilador, claro está, não sabe que a função na classe derivada não está implantada e gerará, quando pedido, o código para criar a tabela, como objeto derivado.
- O linkador, contudo, é incapaz de encontrar a função membro da classe derivada. Assim, é incapaz de construir a 'vtable' da classe derivada;
- O linkador reclama com a mensagem:

```
undefined reference to `vtable for Derived'
```

Eis um exemplo que produz o erro:

```
class Base
{
    public:
        virtual void member()
        {}
};

class Derived
{
    public:
        virtual void member();    // só declarado
};

int main()
{
    Derived d;    // Compilará, já que todos os membros foram declarados.
                // A linkagem falhará, já que não tem
                // a implantação de 'Derived::member()'
}
```

Claro que é fácil corrigir o erro: Implante a função membro virtual em falta.

14.10: Construtores Virtuais

Como vimos (seção 14.2) a C++ suporta destrutores virtuais. Como muitas outras linguagens orientadas ao objeto (p.ex. Java), contudo, a noção de construtores virtuais não é suportada. A ausência de um construtor virtual se torna um problema quando só um ponteiro ou referência a uma classe de base está disponível e uma cópia de um objeto derivado é requerida. Gamma et al. (1995) desenvolveu o Projeto Padrão de um Protótipo (Prototype Design Pattern) para tais situações.

No Projeto Padrão de um Protótipo a cada classe derivada é dada a tarefa de disponibilizar uma função membro que retorna um ponteiro a uma nova cópia do objeto para o qual o membro é chamado. O nome usual desta função é 'clone()'. Uma classe de base que suporte `clonagem` só necessita para definir um destrutor e um construtor de cópias virtuais uma pura função virtual, tendo como protótipo 'virtual Base *clone() const = 0'.

Como 'clone()' é uma função virtual pura, cada classe derivada tem que implantar seu próprio `construtor virtual`.

Esta construção é suficiente em muitos casos onde temos um ponteiro ou referência a uma classe de base, mas falha, por exemplo, nos recipientes abstratos. Não podemos criar um 'vector<Base>', com as características da 'Base' e com o puro membro construtor de 'copy()' em sua interface, como

'Base()' é chamada para iniciar os novos elementos de tal vetor. Isto é impossível já que 'clone()' é uma função virtual pura, assim um objeto 'Base()' não pode ser construído.

A solução intuitiva, fornecendo 'clone()' com implantação padrão como uma função virtual ordinária, também não funciona já que as chamadas do recipiente ao construtor de cópias normal 'Base(Base const &)', que teria que chamar 'clone()' para obter uma cópia do construtor de cópias teria como argumento 'clone()'. Aqui se torna nebuloso o que fazer com essa cópia, já que o novo objeto 'Base' existe e não contém um ponteiro ou referência para 'Base' para endereçar o valor de retorno de 'clone()'

Uma alternativa e melhor solução é manter a classe 'Base' original (definida como classe abstrata) e manipular os ponteiros 'Base' retornados por 'clone()' numa classe separada 'Clonable'. No Capítulo 16 encontraremos meios de fundir 'Base' e 'Clonable' numa classe, mas para já as definiremos como classes separadas.

A classe 'Clonable' é bem padrão. Como contém um membro apontador necessita um construtor de cópias, destrutor e operador de adjudicação sobrecarregado (veja Capítulo 7). Lhe é dado pelo menos um membro não estandarte: 'Base &get() const' que retorna uma referência a um objeto derivado ao qual o membro de dados de 'Clonable' se refere 'Base *' e opcionalmente um construtor 'Clonable' ('Base const &') para permitir promoções de objetos da classe derivada de 'Base' para 'Clonable'.

Toda classe não abstrata derivada de 'Base' precisa implantar 'Base *clone()' retornando um ponteiro para uma cópia recém criada (alocada) do objeto para o qual 'clone()' foi chamada.

Uma vez definida uma classe derivada (p.ex., 'Derived1'), podemos por as facilidades de 'Clonable' e 'Base' em bom uso.

No seguinte exemplo vemos 'main()' onde um 'vector<Clonable' foi definido. Um objeto anônimo de 'Derived1' é posto no vetor. Este procedimento é como segue:

- Cria-se o objeto anônimo de 'Derived1';
- É promovido a 'Clonable', usando-se 'Clonable(Base const &)', chamando-se 'Derived1::clone()';
- O objeto 'Clonable' recém criado é inserido no vetor, usando-se 'Clonable(Clonable const &)', outra vez com 'Derived1::clone()'.

Nesta seqüência, são usados dois objetos temporários: O objeto anônimo e o objeto de 'Derived1' construído pela primeira chamada a 'Derived1::clone()'. O terceiro objeto 'Derived1' é inserido

no vetor. Depois de inserir o objeto no vetor, os dois objetos temporários são destruídos.

Em seguida o membro 'get()' é usado em combinação com 'typeid' para mostrar o tipo do objeto 'Base &': Um objeto 'Derived1'.

A parte mais interessante de 'main()' é o vetor em linha 'vector<Clonable>v2(bv)', onde uma cópia do primeiro vetor é criada. Como mostrado, a cópia conserva intactos os tipos das referências de 'Base'.

No final do programa, criamos dois objetos 'Derived1', que são corretamente eliminados pelo destrutor do vetor. Eis o programa completo, ilustrando o conceito de 'construtor virtual':

```
#include <iostream>
#include <vector>
#include <typeinfo>

class Base
{
public:
    virtual ~Base()
    {}
    virtual Base *clone() const = 0;
};

class Clonable
{
    Base *d_bp;

public:
    Clonable()
    :
        d_bp(0)
    {}
    ~Clonable()
    {
        delete d_bp;
    }
    Clonable(Clonable const &other)
    {
        copy(other);
    }
    Clonable &operator=(Clonable const &other)
    {
        if (this != &other)
        {
            delete d_bp;
            copy(other);
        }
        return *this;
    }
};
```

```

    }

    // Novas construções virtuais:
    Clonable(Base const &bp)
    {
        d_bp = bp.clone();          // permite iniciação de
    }                               // Base e objetos derivados
    Base &get() const
    {
        return *d_bp;
    }

private:
    void copy(Clonable const &other)
    {
        if ((d_bp = other.d_bp))
            d_bp = d_bp->clone();
    }
};

class Derived1: public Base
{
public:
    ~Derived1()
    {
        std::cout << "~Derived1() called\n";
    }
    virtual Base *clone() const
    {
        return new Derived1(*this);
    }
};

using namespace std;

int main()
{
    vector<Clonable> bv;

    bv.push_back(Derived1());
    cout << "=="<endl;

    cout << typeid(bv[0].get()).name() << endl;
    cout << "=="<endl;

    vector<Clonable> v2(bv);
    cout << typeid(v2[0].get()).name() << endl;
    cout << "=="<endl;
}

```


Capítulo 15: Classes com membros apontadores

No Capítulo 7 foram discutidas classes com membros de dados apontadores. Como vimos, quando ocorrem membros de dados nas classes, tais classes merecem um tratamento especial.

Agora é bem sabido como tratar membros de dados ponteiros: São usados construtores para iniciar os ponteiros, se necessita destrutores para liberar a memória apontada por esses ponteiros.

Ainda mais, nas classes com membros de dados apontadores construtores de cópia e operadores de adjudicação sobrecarregados normalmente são necessários também.

Contudo, em algumas situações não necessitamos um apontador para um objeto, mas um ponteiro a membros de um objeto. Neste capítulo estes ponteiros especiais são o tópico de discussão.

15.1: Ponteiros a Membros: Um Exemplo

Sabendo como os ponteiros a variáveis são usados não é intuitivamente que nos conduz ao conceito de apontadores a membros. Mesmo que se tenha em conta o tipo retornado e o tipo de parâmetro das funções membro podemos encontrar surpresas. Por exemplo, considere a seguinte classe:

```
class String
{
    char const *(*d_sp)() const;

    public:
        char const *get() const;
};
```

Para esta classe não é possível fazer um 'char const *(*d_sp)() const) apontar para a função membro 'get()' da classe 'String': A d_sp não pode ser dado o endereço da função 'get()'.

Uma das razões para isso é que a variável tem escopo global, enquanto a função membro 'get()' está definida na classe 'String' e tem escopo na classe. O fato da variável 'd_sp' ser parte da classe 'String' é irrelevante. De acordo com a definição de 'd_sp' ela aponta para uma função externa à classe.

Conseqüentemente, para definir um ponteiro a um membro (dados ou funções, mas usualmente uma função) de uma classe, o escopo do ponteiro tem que ser para dentro da classe. Sendo

assim, um ponteiro para um membro da classe 'String' é definido como:

```
char const *(String::*d_sp)() const;
```

Devido ao prefixo 'String::', 'd_sp' é definido como apontador só no contexto da classe 'String'. É definido como apontador a uma função na classe 'String', não espera argumentos, não modifica seus objetos de dados e retorna um apontador a caracteres constantes.

15.2: Definindo ponteiros a membros

Ponteiros a membros são definidos prefixando-se um ponteiro normal com a classe apropriada mais o operador de resolução de escopo. Por isso na seção anterior usamos 'char const (String::d_sp() const' para indicar:

- d_sp é um ponteiro (*d_sp);
- a algo na classe 'String' (String::*d_sp);
- É um apontador a uma função constante, que retorna um 'char const *':

```
char const * (String::*d_sp)() const;
```

- O protótipo da função correspondente é, portanto:

```
char const *String::somefun() const;
```

Uma função sem parâmetros da classe 'String', que retorna um 'char const *'.

Assim o procedimento normal para construir apontadores pode ser aplicado:

- Ponha parênteses no nome da função (e seu nome de classe):

```
char const * (String::somefun)() const
```

- Ponha um caracter de apontador (asterístico (*)) imediatamente antes do nome da função:

```
char const * (String:: * somefun)() const
```

- Substitua o nome da função pelo nome da variável ponteiro:

```
char const *(String::*d_sp)() const
```

Outro exemplo, esta vez definindo um apontador a um membro de dados. Assuma que a classe 'String' contém uma cadeia de caracteres como membro. Como construir um ponteiro a este membro? Novamente seguimos o procedimento básico:

- Ponha parênteses no nome da variável (e seu nome de classe):

```
string (String::text)
```

- Ponha um caracter de apontador (asterístico (*)) imediatamente antes do nome da variável:

```
string (String::*text)
```

- Substitua o nome da variável com o nome do ponteiro da variável:

```
string (String::*tp)
```

Neste caso os parênteses são supérfluos e podem ser omitidos:

```
string String::*tp
```

Alternativamente uma regra prática muito simples é:

- defina uma variável apontador normal (i.e., global)
- Ponha o prefixo com o nome da classe ao caracter de apontador, uma vez que se aponte para algo dentro da classe.

Por exemplo, o seguinte apontador a uma função global:

```
char const * (*sp) () const;
```

Se torna um ponteiro a uma função membro depois de prefixá-lo com classe e escopo:

```
char const * (String::*d_sp) () const;
```

Nada nos força, na discussão acima, a definir os apontadores a membros na classe 'String'. O ponteiro a um membro pode ser definido na classe (assim se tornando um membro de dados da classe) ou em outra classe ou como variável local ou global. Em todos esses casos ao ponteiro a um membro pode ser dado o endereço do tipo de membro ao qual aponta. O importante é que o apontador a um membro pode ser iniciado ou adjudicado sem necessidade de ser um objeto da classe correspondente.

Iniciando ou adjudicando um endereço a tal ponteiro não faz nada mais que indicar a qual membro ele deve apontar. Isto pode ser considerado um gênero de endereço relativo: Relativo ao objeto para o qual a função é chamada. Não se requer objeto quando os apontadores a membros são iniciados ou adjudicados. Por outro lado, enquanto é permitido iniciar ou adjudicar um ponteiro a membro, claro, não é possível acessar membros sem um objeto associado.

No exemplo seguinte ilustramos a iniciação e adjudicação de ponteiros a membros (para tal todos os membros da classe 'pointerDemo' são definidos públicos). No exemplo note o uso do operador '&-operator' a determinado endereço dos membros. Estes operadores também têm que possuir o escopo da classe. Mesmo quando usamos nos membros da própria classe.

```

class PointerDemo
{
    public:
        unsigned get() const
        {
            return d_value;
        }
        unsigned d_value;
};

int main()
{
    // iniciação
    unsigned (PointerDemo::*getPtr)() const = &PointerDemo::get;
    unsigned PointerDemo::*valuePtr        = &PointerDemo::d_value;

    // adjudicação
    getPtr    = &PointerDemo::get;
    valuePtr  = &PointerDemo::d_value;
}

```

Nada especial está envolvido: A diferença com ponteiros de escopo global é que aqui estamos restritos ao escopo da classe 'PointerDemo'. Devido a essa restrição na definição de todos os ponteiros e variáveis cujos endereços são usados se deve das o escopo da classe 'PointerDemo'. Os apontadores a membros também podem ser usados com funções membro virtuais. Não há mudanças se, p.ex., 'get()' for definida como uma função membro virtual.

15.3: Usando ponteiros a membros

Na seção anterior vimos como definir apontadores a funções membro. Para usar esses ponteiros, sempre é necessário um objeto. Com ponteiros que operem num escopo global, o operador de referência 'operator *' é usado para alcançar o objeto ou valor apontado. Com ponteiros a objetos o operador de seleção de campo (->) é usado no apontador ou o seletor de campo (.) opera no objeto para selecionar o membro apropriado.

Para usar um apontador a um membro em combinação com um objeto deve-se usar o seletor de campo (.**) com o ponteiro ao membro. Para se usar um ponteiro a um membro via um ponteiro a um objeto deve-se usar o 'seletor de campo do ponteiro a membro através de um ponteiro a objeto' (->*). Estes dois operadores combinam as noções de, por um lado, seleção de campo (.) ou (->) para se chegar ao campo apropriado de um objeto e, por outro lado, a noção de referência: Uma operação de referência é usada para se obter a função ou variável cujo ponteiro aponta.

Usando o exemplo da seção anterior, vejamos como podemos usar o ponteiro a função membro e ponteiro a um membro de dados:

```
#include <iostream>
```

```

class PointerDemo
{
    public:
        unsigned get() const
        {
            return d_value;
        }
        unsigned d_value;
};

using namespace std;

int main()
{
    // iniciação
    unsigned (PointerDemo::*getPtr)() = &PointerDemo::get;
    unsigned PointerDemo::*valuePtr = &PointerDemo::d_value;

    PointerDemo object;                // (1) (veja o texto)
    PointerDemo *ptr = &object;

    object.*valuePtr = 12345;           // (2)
    cout << object.*valuePtr << endl;
    cout << object.d_value << endl;

    ptr->*valuePtr = 54321;              // (3)
    cout << object.d_value << endl;

    cout << (object.*getPtr)() << endl;  // (4)
    cout << (ptr->*getPtr)() << endl;
}

```

Notamos:

- No comando (1) são definidos um objeto 'PointerDemo' e um apontador a tal objeto;
- No comando (2) especificamos um objeto e aqui o operador '.' para o membro 'valuePtr' apontá-lo. A este membro é dado um valor.
- No comando (3) ao mesmo membro é adjudicado outro valor, mas agora usando o apontador a um objeto 'PointerDemo'. Aqui usamos o operador '->'.
 - No comando (4) '.' e '->' outra vez são usados, mas desta vez para chamar uma função através de um apontador a membro. Note que a lista de argumentos da função tem prioridade mais alta que o operador de seleção de campo de um ponteiro, portanto deve ser protegido pondo-o entre parênteses.

Os apontadores a membros podem ser usados com utilidade em situações quando uma classe tem um membro que se comporta de maneira diferente em, p.ex., um estado de configuração. Considere uma vez mais uma classe 'Pessoa' da seção 7.2. Esta classe tem campos que guardam o nome de pessoas, endereço e telefone. Assumamos que desejamos construir uma base de dados de empregados. Podemos inquirir a base de dados, mas dependendo do tipo de pessoa que interroga pode desejar o nome, nome e telefone ou todos os dados guardados da pessoa. Isto implica que uma função membro como 'endereço()' tem que retornar algo como '<não disponível>' nos casos em que o interrogante não tenha autorização de ver o endereço da pessoa.

Assuma que a base de dados de empregados foi aberta com um argumento que reflita a categoria do empregado que quer fazer algumas indagações. A categoria reflete sua posição na organização, como DIRETOR, SUPERVISOR, VENDEDOR ou ESCRITURÁRIO. As duas primeiras categorias têm permissão plena sobre os empregados, um VENDEDOR tem permissão de ver os telefones, enquanto um ESCRITURÁRIO só pode verificar se a pessoa é membro da organização ou não, atualmente.

Construímos, agora, uma cadeia de caracteres 'InfoPessoa(char const *nome)' membro na classe da base de dados. Uma implantação padrão pode ser:

```
string DadosPessoa::InfoPessoa(char const *nome)
{
    Pessoa *p = lookup(nome);    // vê se `nome` existe

    if (!p)
        return "não encontrado";

    switch (d_categoria)
    {
        case DIRETOR:
        case SUPERVISOR:
            return todaInfo(p);
        case VENDEDOR:
            return foneN(p);
        case CLERK:
            return sóNome(p);
    }
}
```

Apesar de não tomar muito tempo, a 'switch' precisa ser avaliada cada vez que 'códigoPessoa()' seja chamada. No lugar de usar 'switch' podemos definir um membro 'd_infoPtr' como ponteiro a uma função membro da classe 'DadosPessoa' que retorne uma cadeia de caracteres e espere uma referência a 'Pessoa' como argumento. Note que esse ponteiro pode ser usado para apontar a 'todaInfo()', 'foneN()' ou 'sóNome()'. Ainda mais, a função para a qual a variável ponteiro aponta será conhecida na construção do objeto 'DadosPessoa'.

Depois de ter apontado o membro 'd_infoPtr' para a função membro apropriada, a função membro 'infoPessoa()' pode ser re-escrita:

```
string DadosPessoa::InfoPessoa(char const *nome)
{
    Pessoa *p = lookup(nome);          // vê se `nome' existe

    return p ? (this->*d_infoPtr)(p) : "não encontrado";
}
```

Note a construção sintática quando se usa um ponteiro a um membro da classe: 'this->*d_infoPtr'.

O membro 'd_infoPtr' é definido como segue (na classe DadosPessoa, omitindo outros membros):

```
class DadosPessoa
{
    string (DadosPessoa::*d_infoPtr)(Pessoa *p);
};
```

Finalmente o construtor precisa iniciar 'd_infoPtr' para apontar para a função membro correta. O construtor poderia ter o seguinte código (mostrando só o código pertinente):

```
DadosPessoa::DadosPessoa(DadosPessoa::CategoriaEmpregado cat)
{
    switch (cat)
    {
        case DIRETOR:
        case SUPERVISOR:
            d_infoPtr = &PersonData::allInfo;
        case VENDEDOR:
            d_infoPtr = &PersonData::noPhone;
        case ESCRITURÁRIO:
            d_infoPtr = &PersonData::nameOnly;
    }
}
```

Note como os endereços das funções membro são determinados: A classe escopo 'DadosPessoa' precisa ser especificada, mesmo dentro da função membro da classe 'DadosPessoa'.

Um exemplo usando apontadores a membros de dados é dado na seção 17.4.60, no contexto do algoritmo genérico 'stable_sort()'.

15.4: Ponteiros a membros estáticos

Um membro estático de uma classe existe sem um objeto de sua classe. Existem

separadamente de qualquer objeto de suas classes. Quando esses membros estáticos são públicos, podem ser acessados como entidades globais, não obstante, seus nomes de classe são requeridos quando usados.

Assuma que uma classe 'String' possui uma função membro estática 'int n_strings()' que retorna o número de objetos 'string' criados. Então sem usar qualquer objeto 'String' a função 'String::n_strings()' pode ser chamada:

```
void fun()
{
    cout << String::n_strings() << endl;
}
```

Os membros estáticos públicos podem ser acessados como entidades globais (mas veja a seção 10.2.1). Os membros estáticos privados, por outro lado, podem ser acessados somente dentro do contexto de suas classes: Só podem ser acessados de dentro das funções membro de sua classe.

Enquanto membros estáticos não possuem objetos, mas são comparáveis a funções e dados globais, seus endereços podem ser guardados em ponteiros normais, operando em nível global. Usando-se um apontador a membro para endereçar um membro estático de uma classe pode ser produzido um erro de compilação.

Por exemplo, o endereço de uma função membro estática 'int String::n_strings()' pode simplesmente ser guardado numa variável 'int (*pfi)()', mesmo que 'int (*pfi)()' nada tenha a ver em comum com a classe 'String'. Isto está ilustrado no seguinte exemplo:

```
void fun()
{
    int (*pfi)() = String::n_strings;
                // endereço da função membro estática

    cout << (*pfi)() << endl;
                // imprime o valor produzido por String::n_strings()
}
```

15.5: Tamanhos do Ponteiro

Uma característica peculiar de um apontador a um membro é que seus tamanhos diferem dos ponteiros 'normais'. Considere o seguinte pequeno programa:

```
#include <string>
#include <iostream>

class X
{
```

```

    public:

    void fun() { cout << "hello\n"; }
    string d_str;
};

using namespace std;

int main()
{
    cout
<< "Tamanho de um ponteiro a um membro de dados: " << sizeof(&X::d_str) << "\n"
<< "Tamanho de um ponteiro a uma função membro: " << sizeof(&X::fun) << "\n"
<< "Tamanho de um ponteiro a dados não membros: " << sizeof(char *) << "\n"
<< "Tamanho de um ponteiro a função livre: " << sizeof(&printf) << endl;
}

/*
    Saída Gerada:

    Tamanho de um ponteiro a um membro de dados: 4
    Tamanho de um ponteiro a uma função membro: 8
    Tamanho de um ponteiro a dados não membros: 4
    Tamanho de um ponteiro a função livre: 4
*/

```

Note que o tamanho de um ponteiro a uma função membro é de 8 bytes, enquanto todos os outros ponteiros possuem 4 bytes (usando o compilador Gnu g++).

Em geral, o tamanho destes apontadores não são usados explicitamente, mas seus tamanhos diferentes podem causar alguma confusão em comandos como:

```
printf("%p", &X::fun);
```

Claro está que 'printf' não é uma boa ferramenta para produzir o valor destas especificações de ponteiros C++. Os valores destes apontadores podem ser inseridos em 'streams' quando uma união, reinterprete os 8 bytes dos ponteiros como uma série de valores 'size_t char' for usada:

```

#include <string>
#include <iostream>
#include <iomanip>

class X
{
    public:
    void fun()
    {
        std::cout << "hello\n";
    }
    std::string d_str;
}

```



```

};

using namespace std;

int main()
{
    union
    {
        void (X::*f) ();
        unsigned char *cp;
    }

    u = { &X::fun };

    cout.fill('0');
    cout << hex;
    for (unsigned idx = sizeof(void (X::*))(); idx-- > 0; )
        cout << setw(2) << static_cast<unsigned>(u.cp[idx]);
    cout << endl;
}

```

Capítulo 16: Classes Aninhadas

As classes podem ser definidas dentro de outras classes. As classes definidas dentro de outras classes são chamadas classes aninhadas. As classes aninhadas são usadas em situações onde a classe aninhada tem uma relação conceitual muito próxima com a classe que a envolve. Por exemplo, com a classe 'string' um tipo 'string::iterator' está disponível o qual fornecerá todos os elementos (caracteres) guardados na 'string'. Este tipo 'string::iterator' poderia ser definido como um objeto iterador, definido como uma classe aninhada na classe 'string'.

Uma classe pode ser aninhada em qualquer parte da classe envolvente: Na seção pública, protegida ou privada. Tal classe aninhada pode ser considerada membro da classe envolvente. O acesso e as regras normais das classes se aplicam às classes aninhadas. Se uma classe for aninhada na seção pública de outra classe, é visível fora da classe envolvente. Se for aninhada na seção protegida é visível em sub-classes, derivadas da classe envolvente (veja o Capítulo 13), se for aninhada na seção privada, só é visível para os membros da classe envolvente.

A classe envolvente não tem privilégios especiais respeito à classe aninhada. Assim, a classe aninhada ainda tem o controle pleno na acessibilidade de seus membros pela classe envolvente. Por exemplo, considere a seguinte definição de classe:

```
class Surround
{
    public:
        class FirstWithin
        {
            int d_variable;

            public:
                FirstWithin();
                int var() const
                {
                    return d_variable;
                }
        };
    private:
        class SecondWithin
        {
            int d_variable;

            public:
```

```

        SecondWithin();
        int var() const
        {
            return d_variable;
        }
    };
};

```

Nesta definição o acesso aos membros é definido como segue:

- A classe 'FirstWithin' é visível dentro e fora de 'Surround'. A classe 'FirstWithin', portanto, tem escopo global;
- O construtor de 'FirstWithin' e sua função membro 'var()' também são globais;
- O membro de dados 'int d_variable' só é visível para os membros da classe 'FirstWithin'. Nem os membros de 'Surround' nem os membros de 'SecondWithin' podem acessar a 'd_variable' da classe 'FirstWithin' diretamente;
- A classe 'SecondWithin' só é visível dentro de 'Surround'. Os membros públicos de 'SecondWithin' também podem ser usados pelos membros da classe 'firstWithin', como classe aninhada pode ser considerada membro de sua classe envolvente;
- O construtor 'SecondWithin()' e a função membro 'var()' da classe 'SecondWithin' também só podem ser acessados pelos membros de 'Surround' (e membros das classes aninhadas);
- O membro de dados 'd_variable' da classe 'SecondWithin' só é visível pelos membros da classe 'SecondWithin'. Nem os membros de 'Surround' nem os de 'FirstWithin' podem acessar 'd_variable' da classe 'SecondWithin' diretamente;
- Como sempre, um objeto do tipo da classe é requerido antes que seus membros possam ser chamados. Isto também é assim para as classes aninhadas.

Se a classe envolvente tivesse direitos de acesso aos membros privados de sua classe aninhada ou se a classe aninhada tivesse acesso aos membros privados da classe envolvente, as classes seriam definidas como 'friend classes' (veja seção 16.3).

As classes aninhadas podem ser consideradas membros da classe envolvente. Assim, um membro da classe 'Surround' não pode acessar 'FirstWithin::var()' diretamente. Isto é compreensível, considerando o fato de que um objeto 'Surround' não é também um objeto 'FirstWithin' ou 'SecondWithin'. De fato, classes aninhadas são só tipos de nomes. Não implica que os objetos de tal

classe existam automaticamente na classe envolvente. Se um membro da classe envolvente necessita usar um membro (não estático) da classe aninhada então a classe envolvente tem que definir um objeto da classe aninhada, que pode então ser usado pelos membros da classe envolvente, bem como seus membros.

Por exemplo, na seguinte definição de classe existe uma envolvente 'Outer' e uma aninhada 'Inner'. A classe 'Outer' contém uma função membro 'caller()' que usa um objeto de 'Inner' que é composto em 'Outer' para chamar a função membro de 'inner', 'infunction()':

```
class Outer
{
    public:
        void caller()
        {
            d_inner.infunction();
        }
    private:
        class Inner
        {
            public:
                void infunction();
        };
        Inner d_inner;      // a classe Inner precisa ser conhecida
};
```

A mencionada função 'Inner::infunction()' pode ser chamada como parte da definição, na linha, de 'Outer::coller()', mesmo que a definição da classe 'Inner' ainda não tenha sido examinada pelo compilador. Por outro lado, o compilador precisa ter examinado a definição da classe 'Inner' antes que um membro de dados da classe seja definido.

16.1: Definindo membros de classes aninhadas

As funções membro das classes aninhadas podem ser definidas como funções 'in line'. Funções membro em linha podem ser definidas como se fossem definidas fora da definição da classe: Se a função 'Outer::caller()' tivesse sido definida fora da classe 'Outer', a definição plena da classe (incluindo a definição da classe 'Inner') teriam sido efetivas ao compilador. Nessa situação a função é perfeitamente compatível. As funções em linha podem ser compiladas igualmente: podem ser definidas e usar qualquer classe aninhada. Mesmo que apareçam depois na interface de classe.

As funções membros das classes aninhadas também podem ser definidas fora de sua classe envolvente. Considere o construtor da classe 'FirstWithin' no exemplo da seção anterior. O construtor de 'FirstWithin' está definido na classe 'FirstWithin' que é, por seu turno, definida dentro da classe

'Surround'. Conseqüentemente, o escopo da classe, das duas classes, precisa ser usado para definir o construtor. P.ex.:

```
Surround::FirstWithin::FirstWithin()  
{  
    variable = 0;  
}
```

Membros estáticos de dados podem ser definidos em concordância. Se a classe 'FirstWithin' tivesse um membro de dados estático `size_t`, 'epoch', poderia ser iniciado como segue:

```
size_t Surround::FirstWithin::epoch = 1970;
```

Ainda mais, são necessários múltiplos operadores de resolução de escopo para referenciar membros públicos estáticos com código exterior da classe envolvente:

```
void showEpoch()  
{  
    cout << Surround::FirstWithin::epoch = 1970;  
}
```

Dentro da classe 'Surround' somente 'FirstWithin::scope' deve ser usado; dentro da classe 'FirstWithin' não há necessidade de referência explícita ao escopo.

E a respeito à classe 'SecondWithin'? As classes 'FirstWithin' e 'SecondWinthin' estão aninhadas em 'Surround' e podem ser consideradas membros da classe envolvente. Como membros da mesma classe podem referenciar diretamente uma à outra, os membros da classe 'SecondWithin' podem se referir aos membros (públicos) da classe 'FirstWithin'. Conseqüentemente, os membros da classe 'SecondWithin' podem referenciar o membro de 'FirstWithin' como 'FirstWithin::epoch'.

16.2: Declarando classes aninhadas

As classes aninhadas podem ser declaradas antes de serem definidas na classe envolvente. Tal declaração avançada é requerida se uma classe contém muitas classes aninhadas e as classes aninhadas contêm ponteiros, referências, parâmetros ou valores de retorno a objetos de outra classe aninhada.

Por exemplo, a seguinte classe 'Outer' contém duas classes aninhadas 'Inner1' e 'Inner2'. A classe 'Inner1' contém um ponteiro a objetos de 'Inner2' e 'Inner2' um ponteiro a objetos de 'Inner1'. Tal referência cruzada requerem declarações avançadas. Estas declarações avançadas precisam estar na mesma categoria de acesso que suas definições. No exemplo seguinte a declaração avançada de 'Inner2' precisa ser dada na seção privada, como sua definição é também parte da interface privada da classe 'Outer':

```
class Outer
```

```

{
    private:
        class Inner2;          // declaração avançada

        class Inner1
        {
            Inner2 *pi2;      // aponta para objetos de Inner2
        };
        class Inner2
        {
            Inner1 *pil;      // aponta para objetos de Inner1
        };
};

```

16.3: Acessando membros privados em classes aninhadas

Para permitir a classes aninhadas acessarem a membros privados de suas classes envoltoras; para acessar membros privados de outras classes aninhadas; ou para permitir a classe envolvente acessar a membros privados de suas classes aninhadas, a palavra chave 'friend' tem que ser usada. Considere a seguinte situação, onde uma classe 'Surround' tem duas classes aninhadas 'FirstWithin' e 'SecondWithin', e cada classe possui membro de dados estáticos 'int s_variable':

```

class Surround
{
    static int s_variable;
public:
    class FirstWithin
    {
        static int s_variable;
        public:
            int value();
    };
    int value();
private:
    class SecondWithin
    {
        static int s_variable;
        public:
            int value();
    };
};

```

Para que a classe 'Surround' seja capaz de acessar os membros privados de 'FirstWithin' e 'SecondWithin', estas duas últimas classes têm que declarar 'Surround' como 'friend'. A função 'Surround::value()' pode acessar os membros privados de suas classes aninhadas. Por exemplo (note a declaração 'friend' nas duas classes aninhadas):

```

class Surround

```

```

{
    static int s_variable;
public:
    class FirstWithin
    {
        friend class Surround;
        static int s_variable;
        public:
            int value();
    };
    int value()
    {
        FirstWithin::s_variable = SecondWithin::s_variable;
        return (s_variable);
    }
private:
    class SecondWithin
    {
        friend class Surround;
        static int s_variable;
        public:
            int value();
    };
};

```

Agora, para permitir que as classes aninhadas acessem aos membros privados de suas classes envoltoras, a classe envolvente tem que declarar suas classes aninhadas como 'friend'. A palavra chave 'friend' só pode ser usada quando a classe que se tornará 'friend' já é conhecida como classe pelo compilador, assim, ou requer uma declaração avançada das classes aninhadas, seguida pela declaração 'friend', ou a declaração 'friend' segue a definição das classes aninhadas. A declaração seguida pela declaração de 'friend' aparece como:

```

class Surround
{
    class FirstWithin;
    class SecondWithin;
    friend class FirstWithin;
    friend class SecondWithin;

    public:
        class FirstWithin;
        ...

```

Alternativamente a declaração 'friend' pode seguir a definição das classes. Note que uma classe pode ser declarada 'friend' após sua definição, enquanto o código em linha na definição já usa o fato que ela será declarada 'friend' da classe envolvente. Note também que o código em linha destas classes aninhadas usa membros da classe envolvente, ainda não conhecida pelo compilador. Finalmente note que 's_variable' que é definida na classe 'Surround' é acessada nas classes aninhadas como 'Surround::s_variable':

```

class Surround
{
    static int s_variable;
public:
    class FirstWithin
    {
        friend class Surround;
        static int s_variable;
        public:
            int value()
            {
                Surround::s_variable = 4;
                return s_variable;
            }
    };
    friend class FirstWithin;
    int value()
    {
        FirstWithin::s_variable = SecondWithin::s_variable;
        return s_variable;
    }
private:
    class SecondWithin
    {
        friend class Surround;
        static int s_variable;
        public:
            int value()
            {
                Surround::s_variable = 40;
                return s_variable;
            }
    };
    friend class SecondWithin;
};

```

Finalmente queremos permitir às classes aninhadas acessarem os membros privados umas das outras. Novamente isto requer algumas declarações 'friend'. Para permitir 'FirstWithin' acessar os membros privados de 'SecondWithin' requer nada mais que uma declaração de 'friend' em 'SecondWithin'. Contudo para permitir que 'SecondWithin' acesse os membros privados de 'FirstWithin' a declaração não pode ser feita plenamente na classe 'FirstWithin', já que a definição de 'SecondWithin' ainda é desconhecida. Requer uma declaração avançada de 'SecondWithin' e esta declaração deve ser feita em 'Surround' antes que em 'FirstWithin'.

Claramente a declaração avançada da classe 'SecondWithin' na classe 'FirstWithin' não faz sentido, já que esta se referiria a uma classe 'SecondWithin' externa (global). É impossível fornecer a declaração avançada da classe aninhada 'SecondWithin' dentro de 'FirstWithin' como classe 'Surround::SecondWithin', pois o compilador iria responder que “Surround does not have a nested type

named 'SecondWithin'.

O procedimento próprio aqui é declarar a classe 'SecondWithin' na classe 'Surround', antes que a classe 'FirstWithin' seja definida. Usando este procedimento a declaração 'friend' de 'SecondWithin' é aceita dentro da definição da classe 'FirstWithin'. A seguinte definição de classe permite completo acesso aos membros privados de todas as classes por todas as outras classes:

```
class Surround
{
    class SecondWithin;
    static int s_variable;
public:
    class FirstWithin
    {
        friend class Surround;
        friend class SecondWithin;
        static int s_variable;
        public:
            int value()
            {
                Surround::s_variable = SecondWithin::s_variable;
                return s_variable;
            }
    };
    friend class FirstWithin;
    int value()
    {
        FirstWithin::s_variable = SecondWithin::s_variable;
        return s_variable;
    }
private:
    class SecondWithin
    {
        friend class Surround;
        friend class FirstWithin;
        static int s_variable;
        public:
            int value()
            {
                Surround::s_variable = FirstWithin::s_variable;
                return s_variable;
            }
    };
    friend class SecondWithin;
};
```

16.4: Aninhando Enumerações

As enumerações também podem ser aninhadas nas classes. O aninhamento de enumerações é uma boa forma de mostrar a conexão estreita entre a enumeração e sua classe. Na classe 'ios' vimos valores como 'ios::beg' e 'ios::cur'. Na implantação da C++ Gnu estes valores são definidos como valores na enumeração 'seek_dir':

```
class ios: public _ios_fields
{
    public:
        enum seek_dir
        {
            beg,
            cur,
            end
        };
};
```

Com propósito de ilustração assumamos que a classe 'DataStructure' pode ser percorrida para diante e para trás. Tal classe define uma enumeração 'Traversal' com valores 'forward' e 'backward'. Ainda uma função membro 'setTraversal()' é definido requerendo um dos dois valores da enumeração. A classe pode ser definida como segue:

```
class DataStructure
{
    public:
        enum Traversal
        {
            forward,
            backward
        };
        setTraversal(Traversal mode);
    private:
        Traversal
            d_mode;
};
```

Na classe 'DataStructure' os valores da enumeração 'Traversal' podem ser usados diretamente. Por exemplo:

```
void DataStructure::setTraversal(Traversal mode)
{
    d_mode = mode;
    switch (d_mode)
    {
        forward:
            break;

        backward:
```

```

        break;
    }
}

```

Fora da classe 'DataStructure' o nome do tipo de enumeração não é usado para se referir aos valores da enumeração. Aqui o nome da classe é suficiente. Só se uma variável do tipo enumeração é requerida o nome do tipo de enumeração é necessário, como ilustrado pelo seguinte pedaço de código:

```

void fun()
{
    DataStructure::Traversal          // enum typename required
        localMode = DataStructure::forward; // enum typename not required

    DataStructure ds;

    ds.setTraversal(DataStructure::backward); // enum typename not required
}

```

Novamente só se 'DataStructure' define uma classe aninhada 'Nested', em seu turno definindo a enumeração 'Traversal', os escopos para as duas classes são requeridos. Nesse caso o exemplo anterior deveria ser:

```

void fun()
{
    DataStructure::Nested::Traversal
        localMode = DataStructure::Nested::forward;

    DataStructure ds;

    ds.setTraversal(DataStructure::Nested::backward);
}

```

16.4.1: Enumerações Vazias

Os tipos de enumeração habitualmente têm valores. Contudo estes não são obrigatórios. Na seção 14.5.1 foi introduzido o tipo 'std::bad_cast'. Um 'std::bad_cast' é lançado pelo operador 'dynamic_cast<>()' quando uma referência a um objeto de uma classe de base não pode ser convertida a uma referência a uma classe derivada. O 'std::bad_cast' pode ser apanhado como tipo, independentemente de qualquer valor que represente.

Não é necessário a um tipo conter valores. É possível definir uma enumeração vazia, sem valores, cujo nome pode ser usado como o nome válido de um tipo, p.ex., uma cláusula 'catch' que define um manipulador de exceções.

Uma enumeração vazia é definida como segue (em geral, mas não necessariamente com uma classe):

```
enum EmptyEnum
{
};
```

Assim uma enumeração vazia pode ser lançada (e apanhada) como uma exceção:

```
#include <iostream>

enum EmptyEnum
{
};

using namespace std;

int main()
try
{
    throw EmptyEnum();
}
catch (EmptyEnum)
{
    cout << "Apanhou a enumeração vazia\n";
}
/*
Saída Gerada:

Apanhou a enumeração vazia
*/
```

16.5: Revendo construtores virtuais

Na seção 14.10 a noção de construtores virtuais foi introduzida. Nessa seção uma classe 'Base' foi usada como uma classe de base abstrata. Uma classe 'Clonable' foi definida para gerenciar os apontadores da classe 'Base' em recipientes como vetores.

Como a classe 'Base' é muito pequena, dificilmente requer qualquer implantação, ela pode muito bem ser definida como uma classe aninhada em 'Clonable'. Isto enfatizará a estreita relação que existe entre 'Clonable' e 'Base' como mostra o modo como as classes derivam de 'Base'. Se escreve:

```
class Derived: public Base

Mas melhor seria:

class Derived: public Clonable::Base
```

Definindo 'Base' como classe aninhada e derivando de 'Clonable::Base', antes que de 'Base', nada precisa ser modificado. Eis o programa anterior da seção 14.10, mas agora usando classes aninhadas:

```
#include <iostream>
```

```

#include <vector>
#include <typeinfo>

class Clonable
{
public:
    class Base
    {
    public:
        virtual ~Base()
        {}
        virtual Base *clone() const = 0;
    };

private:
    Base *d_bp;

public:
    Clonable()
    :
        d_bp(0)
    {}
    ~Clonable()
    {
        delete d_bp;
    }
    Clonable(Clonable const &other)
    {
        copy(other);
    }
    Clonable &operator=(Clonable const &other)
    {
        if (this != &other)
        {
            delete d_bp;
            copy(other);
        }
        return *this;
    }

    // New for virtual constructions:
    Clonable(Base const &bp)
    {
        d_bp = bp.clone(); // permite a iniciação de
    } // Base e objetos derivados
    Base &get() const
    {
        return *d_bp;
    }

private:

```

```

        void copy(Clonable const &other)
        {
            if ((d_bp = other.d_bp))
                d_bp = d_bp->clone();
        }
};

class Derived1: public Clonable::Base
{
public:
    ~Derived1()
    {
        std::cout << "~Derived1() called\n";
    }
    virtual Clonable::Base *clone() const
    {
        return new Derived1(*this);
    }
};

using namespace std;

int main()
{
    vector<Clonable> bv;

    bv.push_back(Derived1());
    cout << "=="<endl;

    cout << typeid(bv[0].get()).name() << endl;
    cout << "=="<endl;

    vector<Clonable> v2(bv);
    cout << typeid(v2[0].get()).name() << endl;
    cout << "=="<endl;
}

```

Capítulo 17: A Biblioteca de Modelos Padrão: Algoritmos Genéricos

A Biblioteca de Modelos Padrão (The Standard Template Library (STL)) consiste de recipientes, algoritmos genéricos, iteradores, objetos funções, alocadores e adaptadores. A STL é uma biblioteca de propósitos gerais consistindo de algoritmos e estruturas de dados. As estruturas de dados usadas nos algoritmos são abstratas no sentido de que os algoritmos podem ser usados com (praticamente) todos os tipos de dados.

Os algoritmos funcionam com os tipos de dados abstratos devido a que algoritmos baseados em modelos. Neste capítulo a construção de modelos não é discutida (veja o capítulo 18 para isso). Este capítulo enfoca o uso desses algoritmos modelo.

Diversas partes da STL já foram discutidas nas Anotações C++. No capítulo 12 os recipientes abstratos foram discutidos e na seção 9.10 os objetos funções foram introduzidos. Mencionamos também iteradores em diversos lugares neste documento.

Os componentes restantes da STL serão cobertos neste capítulo. Serão discutidos iteradores, adaptadores e algoritmos genéricos nas próximas seções. Os alocadores cuidam da alocação de memória na STL. A classe padrão dos alocadores é suficiente para a maioria das aplicações e não é mais discutida nas Anotações C++.

O esquecimento de eliminar memória alocada é uma fonte comum de erros ou vazamento de memória nos programas. A classe modelo 'auto_ptr' pode ser usada para prevenir estes tipos de problemas. A classe 'auto_ptr' é discutida na seção 17.3.

Todos os elementos da STL estão definidos no espaço nomeado padrão. Por isso a diretiva 'using namespace std' ou comparável é requerida a menos que se prefira especificar o espaço nomeado explicitamente. Isto é verdade em pelo menos numa situação: Nos arquivos cabeçalho onde não se deve usar a diretiva 'using', assim aí 'std::', especificação de escopo deve sempre ser especificada ao referirmos aos elementos da STL.

17.1: Objetos Função Predefinidos

Os objetos função desempenham importante papel em combinação com algoritmos genéricos. Por exemplo, existe um algoritmo genérico 'sort()' que espera dois iteradores que definem a amplitude dos objetos a serem ordenados, bem como um objeto função que chama o operador de comparação entre dois objetos apropriado. Passemos rapidamente esta situação. Assuma que num vetor estão guardadas 'strings' e queremos ordenar o vetor em ordem descendente. No caso ordenar o vetor é tão simples como:

```
sort(stringVec.begin(), stringVec.end(), greater<std::string>());
```

O último argumento é reconhecido como um construtor: É uma instância da classe modelo 'greater<>()' aplicada a 'strings'. Este objeto é chamado como um objeto função pelo algoritmo genérico 'sort()'. Ele chamará 'operator>()' do tipo de dados (aqui 'std::string') sempre que 'operator()()' for chamado. Eventualmente quando 'sort()' retorna, o primeiro elemento do vetor será o maior elemento.

O operador 'operator()()' (operador de chamada de funções) não é visível neste ponto: Não confundir os parênteses em 'greater<string>()' com o operador de chamada 'operator()()'. Quando esse operador é usado em 'sort()', recebe dois argumentos: Duas 'strings' para comparar. Internamente o 'operator>()' do tipo de dados para os quais o iterador aponta (i.e., 'string') é chamado pelo operador função ('operator()()') de 'greater<string>' para comparar os dois objetos. Enquanto a chamada à função operador 'greater<>' está definida na linha, o operador de chamada não está presente no código. Mais exatamente 'sort()' chama 'string::operator>()', pensando que está chamando 'greater<>::operator()()'.

Agora que sabemos que um construtor é passado como argumento a (muitos) algoritmos genéricos, podemos projetar nossos próprios objetos função. Assuma que queremos ordenar nosso vetos insensivelmente às maiúsculas e minúsculas. Como proceder? Primeiro notamos que o 'string::operator<()' padrão (para ordenação incremental) não é apropriado, já que faz comparação sensível a maiúsculas e minúsculas. Assim, fornecemos nossa própria classe 'case_less', com a qual duas 'strings' são comparadas insensivelmente ao caso maiúscula ou minúscula. Usando a função estandarte C 'strcasecmp()', o seguinte programa realiza a tarefa. Ordena os argumento da linha de comando em ordem alfabética ascendente:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

using namespace std;

class case_less
{
public:
```



```

        bool operator()(string const &left, string const &right) const
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
    };

    int main(int argc, char **argv)
    {
        sort(argv, argv + argc, case_less());
        for (int idx = 0; idx < argc; ++idx)
            cout << argv[idx] << " ";
        cout << endl;
    }

```

O construtor padrão da classe 'case_less' é usada com 'sort()' como último argumento. Por isso a única função membro que deve ser definida na classe 'case_less' é o objeto operador função 'operator()()'. Como sabemos que será chamado com argumentos 'string', o definimos para esperar dois argumentos 'string', que são usados na função 'strcasecmp()'. Ainda mais, o 'operator()()' está composta na linha, assim, não produz sobrecarga ao ser chamada pela função 'sort()'. As chamadas da função 'sort()' ao objeto função com várias combinações da 'strings', i.e., em coisas que faz. Contudo, de fato chama 'strcasecmp()', devida à sua definição em linha de 'case_less::operator()()'.

O objeto função de comparação frequentemente é um objeto função predefinido, já que estão disponível para muitas operações comuns. Nas próximas seções os objetos função pré-definidos disponíveis estão presente junto com alguns exemplos mostrando seu uso. No fim da seção sobre objetos função os adaptadores são introduzidos. Antes do uso de objetos função pré-definidos deve-se especificar a seguinte diretiva ao pré-processador:

```
#include <functional>
```

Os objetos função pré-definidos são usados predominantemente com algoritmos genéricos. Os objetos função pré-definidos existem para operações aritméticas, relacionais e lógicas. Na seção 20.4 objetos funções pré-definidos são desenvolvidos para operações bit a bit.

17.1.1: Objetos Funções Aritméticos

Os objetos função aritméticos suportam as operações aritméticas padrão: Adição, subtração, multiplicação, divisão, módulo e negação. Estes objetos função aritméticos invocam o operador correspondente ao tipo de dado associado. Por exemplo, para adição o objeto 'plus<type>' está disponível. Se o tipo for size_t então o operador '+' para os valores size_t, se o tipo for 'string' então o operador + para 'string' é usado. Por exemplo:

```
#include <iostream>
```

```

#include <string>
#include <functional>
using namespace std;

int main(int argc, char **argv)
{
    plus<size_t> uAdd;          // objeto função para somar sem sinal

    cout << "3 + 5 = " << uAdd(3, 5) << endl;

    plus<string> sAdd;          // objeto função para somar strings

    cout << "argv[0] + argv[1] = " << sAdd(argv[0], argv[1]) << endl;
}
/*
    Saída Gerada com chamada a: a.out going

    3 + 5 = 8
    argv[0] + argv[1] = a.outgoing
*/

```

Porque isto é útil? Note que o objeto função pode ser usado com todos os tipos de dados (não só com tipos de dados predefinidos), onde o operador particular foi sobrecarregado. Assuma que queremos realizar uma operação sobre uma variável comum por um lado e por outro sobre todos os elementos de um conjunto. P.ex., queremos computar a soma dos elementos de um conjunto ou concatenar todas as 'strings' num texto. Nestas situações os objetos função são úteis. Como visto antes, os objetos função são largamente usados no contexto dos algoritmos genéricos, assim vejamos um deles.

Um dos algoritmos genéricos chama-se 'accumulate()'. Visita todos os elementos implicados por um iterador de extensão e realiza uma operação binária requerida num elemento comum e em cada elemento da extensão, retornando o resultado acumulado. Por exemplo, o seguinte programa acumula todos os argumentos do comandos em linha e imprime a 'string' final:

```

#include <iostream>
#include <string>
#include <functional>
#include <numeric>
using namespace std;

int main(int argc, char **argv)
{
    string result =
        accumulate(argv, argv + argc, string(), plus<string>());

    cout << "Todos os argumentos concatenados: " << result << endl;
}

```

Os dois primeiros argumentos definem a extensão (iterador) dos elementos a visitar, o

terceiro argumento é 'string()'. O objeto 'string' anônima fornece um valor inicial. Poderia muito bem ser iniciada como 'string("Todos os argumentos concatenados")', em tal caso o comando 'cout' seria um simples 'cout << result << endl'.

Então, o operador a ser aplicado é 'plus<string>()'. Note que aqui é chamado um construtor: Não é 'plus<string>', mas 'plus<string>()'. É retornada a 'string' final concatenada.

Agora definimos nossa própria classe 'Time', onde o operador 'operator+()' foi sobrecarregado. Outra vez aplicamos o objeto função pré-definido 'plus', amoldado para nosso recém definido tipo de dado, para somar tempos:

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <functional>
#include <numeric>

using namespace std;

class Time
{
    friend ostream &operator<<(ostream &str, Time const &time)
    {
        return cout << time.d_days << " dias, " << time.d_hours <<
            " horas, " <<
            time.d_minutes << " minutos e " <<
            time.d_seconds << " segundos.";
    }

    size_t d_days;
    size_t d_hours;
    size_t d_minutes;
    size_t d_seconds;

public:
    Time(size_t hours, size_t minutes, size_t seconds)
    :
        d_days(0),
        d_hours(hours),
        d_minutes(minutes),
        d_seconds(seconds)
    {}
    Time &operator+=(Time const &rValue)
    {
        d_seconds += rValue.d_seconds;
        d_minutes += rValue.d_minutes + d_seconds / 60;
        d_hours += rValue.d_hours + d_minutes / 60;
        d_days += rValue.d_days + d_hours / 24;
    }
};
```

```

        d_seconds    %= 60;
        d_minutes    %= 60;
        d_hours      %= 24;

        return *this;
    }
};

Time const operator+(Time const &lValue, Time const &rValue)
{
    return Time(lValue) += rValue;
}

int main(int argc, char **argv)
{
    vector<Time> tvector;

    tvector.push_back(Time( 1, 10, 20));
    tvector.push_back(Time(10, 30, 40));
    tvector.push_back(Time(20, 50,  0));
    tvector.push_back(Time(30, 20, 30));

    cout <<
        accumulate
        (
            tvector.begin(), tvector.end(), Time(0, 0, 0), plus<Time>()
        ) <<
        endl;
}
/*
    Saída Produzida:

    2 dias, 14 horas, 51 minutos e 30 segundos.
*/

```

Note que todas as funções membro de 'Time' na fonte acima são funções em linha. Esta solução foi seguida para manter o exemplo relativamente pequeno e mostrar, explicitamente, que 'operator+=()' pode ser uma função em linha. Por outro lado, na vida real, o 'operator+=()' de 'Time', provavelmente, não seria feita em linha devido ao seu tamanho.

Considerando a discussão prévia, o exemplo é bem adequado. A classe 'Time' define um construtor, este define um operador de inserção e este seu próprio 'operator+()', adicionando dois objetos de tempo.

Em 'main()' quatro objetos estão guardados num objeto 'vector<Time>'. Então o algoritmo genérico 'accumulate()' é chamado para computar o tempo acumulado. Este retorna um objeto 'Time' que é inserido no objeto 'ostream', 'cout'.

Enquanto o primeiro exemplo mostrou o uso de um objeto função nomeado, os últimos dois

exemplos mostraram o uso de um objeto anônimo que é passado à função ('accumulate()).

Os seguintes objetos aritméticos são os objetos pré-definidos:

- 'plus<>()': Como já visto este objeto membro chama 'operator+()' como operador binário, passando seus dois parâmetros e retornando o valor de retorno de 'operator<>()';
- 'minus<>()': Este objeto 'operator()' chama a 'operator-()' como operador binário, passando seus dois parâmetros e retornando o valor de retorno de 'operator-()';
- 'multiplies<>()': Este objeto 'operator()' chama a 'operator*()' como operador binário, passando seus dois parâmetros e retornando o valor de retorno de 'operator*()';
- 'divides<>()': Este objeto membro 'operator()' chama a 'operator/()', passando seus dois parâmetros e retornando o valor de retorno de 'operator/()';
- 'modulus<>()': Este objeto membro 'operator()' chama 'operator%()', passando seus dois parâmetros e retornando o valor de retorno de 'operator%()';
- 'negate<>()': Este objeto membro de 'operator()' chama 'operator-()' como operador unário, passando seu parâmetro e retornando o valor de retorno de 'operator-()'.

Segue um exemplo usando o operador unário 'operator-()', onde o algoritmo genérico 'transform()' é usado para mudar o sinal de todos os membros de um conjunto. O algoritmo genérico 'transform()' espera dois iteradores que definem a extensão dos objetos a serem transformados, um iterador definindo o início do destino (que pode ser o mesmo que define o primeiro argumento) e um objeto função que define uma operação unária para o tipo de dados indicado:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    int iArr[] = { 1, -2, 3, -4, 5, -6 };

    transform(iArr, iArr + 6, iArr, negate<int>());

    for (int idx = 0; idx < 6; ++idx)
        cout << iArr[idx] << ", ";

    cout << endl;
```

```

}
/*
    Saída Gerada:

    -1, 2, -3, 4, -5, 6,
*/

```

17.1.2: Objetos Funções Relacionais

Os operadores relacionais são chamados são chamados pelos objetos função relacionais. Todos os operadores relacionais padrão são suportados: ==, !=, >, >=, < e <=. Os seguintes objetos existem:

- 'equal_to<>()': Este objeto membro do 'operator()' chama 'operator==()' como operador binário, passando seus dois parâmetros e retornando o valor retornado por 'operator==()';
- 'not_equal_to<>()': Este objeto membro de 'operator()' chama 'operator!==' como operador binário, passando seus dois parâmetros e retornando o valor retornado por 'operator!==';
- 'greater<>()': Este objeto membro de 'operator()' chama 'operator>()' como operador binário, passando seus dois parâmetros e retornando o valor retornado por 'operator>()';
- 'greater_equal<>()': Este objeto membro de 'operator()' chama 'operator>=()' como operador binário, passando seus dois parâmetros e retornando o valor retornado por 'operator>=()';
- 'less<>()': Este objeto membro de 'operator()' chama 'operator<()' como operador binário, passando seus dois parâmetros e retornando o valor retornado por 'operator<()';
- 'less_equal<>()': Este objeto membro de 'operator()' chama 'operator<=()' como operador binário, passando seus dois parâmetros e retornando o valor retornado por 'operator<=()';

Como os objetos função aritméticos, estes objetos função podem ser usados como objetos nomeados ou anônimos. Um exemplo usando objetos função relacionais e o algoritmo genérico 'sort()' é:

```

#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{

```

```

    sort(argv, argv + argc, greater_equal<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;

    sort(argv, argv + argc, less<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;
}

```

O algoritmo genérico 'sort()' espera um iterador de extensão e um comparador do tipo de dados ao qual os iteradores apontam. O exemplo mostra a ordenação alfabética de 'strings' e ordenação reversa de 'strings'. Passando 'greater_equal<string>()' as 'strings' são ordenadas de forma decrescente (a primeira palavra será 'a maior'), passando 'less<string>()' as 'strings' serão ordenadas de forma crescente (a primeira palavra será 'a menor').

Note que o tipo dos elementos de 'argv' é 'char *' e a função relacional espera uma 'string'. O objeto relacional 'greater_equal<string>()' usará, portanto, o operador '>=' para 'strings', mas será chamado por 'char * variables'. A promoção de 'char const *' a 'string' é feita silenciosamente.

17.1.3: Objetos Função Lógicos

Os operadores lógicos são chamados pelos objetos função lógicos. Os operadores lógicos padrão são suportados: &&, ||, e !. Os seguintes objetos estão disponíveis:

- 'logical_and<>()': Este objeto membro de 'operator()' chama 'operator&&()' como operador binário, passando seus dois parâmetros e retornando o valor de retorno do 'operator&&()'.
- 'logical_or<>()': Este objeto membro de 'operator()' chama 'operator||()' como operador binário, passando seus dois parâmetros e retornando o valor de retorno do 'operator||()'.
- 'logical_not<>()': Este objeto membro de 'operator()' chama 'operator_not()' como operador binário, passando seus dois parâmetros e retornando o valor de retorno do 'operator_not()'.

Um exemplo usando 'operator!()' no seguinte programa trivial, onde o algoritmo genérico é usado para transformar os valores lógicos do conjunto:

```

#include <iostream>
#include <string>

```

```

#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    bool bArr[] = {true, true, true, false, false, false};
    size_t const bArrSize = sizeof(bArr) / sizeof(bool);

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;

    transform(bArr, bArr + bArrSize, bArr, logical_not<bool>());

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;
}
/*
    Saída Gerada:

    1 1 1 0 0 0
    0 0 0 1 1 1
*/

```

17.1.4: Adaptadores de Funções

Os adaptadores de funções modificam um objeto função existente. Existem dois tipos de adaptadores de funções:

- **Enlaces** (“binders”): São adaptadores de funções que convertem objetos função binários em objetos função unários. Fazem isto enlaçando um objeto a um objeto função constante. Por exemplo, com o objeto função 'minus<int>()', que é um objeto função binário, o primeiro argumento pode ser posto em 100, significando que o resultado será 100 menos o valor do segundo objeto. Um dos dois argumentos deve ser posto num valor específico. Para por o primeiro argumento num valor específico, o objeto função 'bind1st()' é usado. Para por o segundo argumento num valor específico se usa 'bind2nd()'. Como exemplo, assumamos que queremos contar todos os elementos de um vetor 'Pessoa' que excedam (de acordo com algum critério) alguma referência aos objetos 'Pessoa'. Nesta situação passamos o seguinte enlace e objeto função relacional ao algoritmo genérico 'count_if()':

```
bind2nd(greater<Pessoa>(), referencePessoa)
```


O que faz semelhante enlace? Primeiro de tudo é um objeto função, assim precisa de 'operator()()'. Depois, espera dois argumentos: Uma referência a outro objeto função e um operando fixo. Contudo os enlaces são definidos como modelos, é ilustrativo dar uma olhada em suas implantações, assumindo que sejam funções puras. Eis tal pseudo-implantação de um enlace:

```
class bind2nd
{
    FunctionObject const &d_object;
    Operand const &d_rvalue;
public:
    bind2nd(FunctionObject const &object, Operand const &operand)
        :
            d_object(object),
            d_operand(operand)
    {}
    ReturnType operator()(Operand const &lvalue)
    {
        return d_object(lvalue, d_rvalue);
    }
};
```

Quando membro 'operator()()' é chamado o enlace somente passa a chamado ao objeto do 'operator()()', fornecendo-lhe dois argumentos: 'lvalue' que recebeu e o operando fixo que recebeu através de seu construtor. Note a simplicidade destes tipos de classes: Todos seus membros usualmente se pode implantar em linha.

O algoritmo genérico 'count_if()' visita todos os elementos na extensão do iterador, retornando o número de vezes que o predicado especificado como seu argumento final retorna verdadeiro. Cada elemento dentro da extensão do iterador é dado ao predicado, que é, portanto, uma função unária. Usando o enlace, a função binária 'greater()' é adaptada a uma função unária, comparando cada elemento da extensão à pessoa de referência. Eis aqui, para completar, a chamada à função 'count_if()':

```
count_if(pVector.begin(), pVector.end(),
        bind2nd(greater<Person>(), referencePerson))
```

- Negadores são adaptadores de funções que convertem o valor verdadeiro da função predicado. Como há funções predicado binárias e unárias, existem dois adaptadores de funções negadores: 'not1()' usado com objetos função unários e 'not2()' usado com objetos função binários.

Se queremos contar o número de pessoas num 'vector< Pessoa >', que não exceda uma certa pessoa de referência, pode-se, entre outras soluções, usar uma das alternativas:

- Usar um predicado binário que ofereça diretamente a comparação requerida:

```
count_if(pVector.begin(), pVector.end(),
        bind2nd(less_equal<Person>(), referencePerson))
```

- Usar 'not2' combinado com o predicado 'greater()':

```
count_if(pVector.begin(), pVector.end(),
        bind2nd(not2(greater<Person>()), referencePerson))
```

Note que 'not2()' é um negador que nega o valor verdadeiro do membro 'operator()()' um binário: Precisa ser usado para encobrir o predicado binário 'grater<Pessoa>()', negando seu valor verdadeiro.

- Usar 'not1()' combinado com o predicado 'bind2nd()':

```
count_if(pVector.begin(), pVector.end(),
        not1(bind2nd(greater<Person>()), referencePerson))
```

Note que 'not1()' está negando o valor verdadeiro de um membro 'operator()()': É usado para encobrir o predicado unário 'bind2nd()', negando seu valor verdadeiro.

O pequeno exemplo seguinte ilustra o uso de um adaptador de função negador, completando a seção de objetos função:

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;

int main(int argc, char **argv)
{
    int iArr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    cout << count_if(iArr, iArr + 10, bind2nd(less_equal<int>(), 6)) <<
        endl;
    cout << count_if(iArr, iArr + 10, bind2nd(not2(greater<int>()), 6)) <<
        endl;
    cout << count_if(iArr, iArr + 10, not1(bind2nd(greater<int>()), 6)) <<
        endl;
}
/*
```

```
Saída Produzida:  
  
6  
6  
6  
*/
```

Poderíamos nos perguntar qual das alternativas de solução é a mais rápida. Usando a primeira solução, onde se usa diretamente um objeto função disponível, duas ações são realizadas para cada iteração de 'count_if()':

- Operador de enlace 'operator()()' é chamado;
- A operação '<=' é realizada para valores inteiros.

Com a segunda solução, onde o negador 'not2' é usado para negar o valor verdadeiro do complementar do adaptador da função lógica, três ações são realizadas para cada iteração de 'count_if()'

- O operador de enlace 'operator()()' é chamado;
- O negador de 'operator()()' é chamado;
- A operação '>' é realizada para valores inteiros.

Usando a terceira solução, onde um negador 'not1' é usado para negar o valor verdadeiro do enlace, três ações são realizadas para cada iteração de 'count_if()':

- O negador de 'operator()()' é chamado;
- O enlace de 'operator()()' é chamado;
- A operação '>' é realizada para valores inteiros.

Daqui podemos deduzir que a primeira solução é a mais rápida. Sem dúvida, usando um compilador Gnu g++ sobre um velho pentium 166 Mhz, que realiza 3.000.000 de chamadas a 'count_if()' mostra que a primeira solução requer cerca de 70% do tempo necessário para as outras soluções.

Contudo estas diferenças desaparecem se o compilador for instruído para otimizar em velocidade (usando a flag -O6 do compilador). Interpretando estes resultados devemos ter presente que funções aninhadas múltiplas se fundem numa só chamada à função se a implantação destas funções está dada em linha. Se isto ocorre, as três soluções ficam reduzidas a uma simples operação: A comparação entre dois valores inteiros. Parece que o compilador faz isto quando requisitado a otimizar em velocidade.

17.2: Iteradores

Os iteradores são objetos que atuam como ponteiros. Os iteradores têm as seguintes características gerais:

- Dois iteradores podem ser comparados em igualdade usando os operadores '==' e '!='. Note que os operadores de ordem (p.ex, '>', '>') não são normalmente usados.
- Dado um iterador 'iter', '*iter' representa o objeto para o qual o iterador aponta (alternativamente pode ser usado 'iter->' para acessar os membros do objeto apontado por 'iter').
- '++iter' ou 'iter++' avança o iterador para o próximo elemento. A noção de avançar um iterador para o próximo elemento é consequentemente aplicada: muitos recipientes possuem um tipo 'reversed_iterator', que a operação 'iter++' acessa o elemento anterior em sequência.
- A aritmética dos ponteiros pode ser usada em recipientes com seus elementos armazenados consecutivamente em memória. Isto inclui vetores e 'deque'. Para estes recipientes 'iter + 2' aponta para o segundo elemento além daquele apontado por 'iter'.
- Um iterador que foi meramente definido é comparável a um ponteiro zero, como mostra o pequeno exemplo:

```
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<int>::iterator vi;

    cout << &*vi << endl;    // imprime 0
}
```

Os recipientes da STL usualmente definem membros que produzem iteradores (i.e., tipo iterador) usando as funções membro 'begin()' e 'end()' e, no caso de iteradores reversos (tipo 'reversed_iterator'), 'rbegin()' e 'rend()'. A prática padrão requer que a extensão do iterador seja a esquerda inclusive: A notação [left, right) indica que “left” é um iterador que aponta para o primeiro elemento a ser considerado, enquanto “right” é um iterador que aponta justo para o primeiro elemento além do último elemento a ser usado. A extensão do iterador se diz vazia quando 'left == right'. Note que com recipientes vazios os iteradores de começo e fim são iguais.

O seguinte exemplo mostra uma situação onde todos os elementos de um vetor de 'strings' são escritos em 'cout' usando o iterador de extensão '[begin(), end())', e o iterador '[rbegin(), rend())'. Note que o laço 'for' de ambas as extensões são idênticos:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char **argv)
{
    vector<string> args(argv, argv + argc);

    for
    (
        vector<string>::iterator iter = args.begin();
        iter != args.end();
        ++iter
    )
        cout << *iter << " ";
    cout << endl;

    for
    (
        vector<string>::reverse_iterator iter = args.rbegin();
        iter != args.rend();
        ++iter
    )
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

Ainda mais, a STL define tipos de 'const_iterator' capazes de visitar uma série de elementos num recipiente constante. Visto que os elementos do vetor no exemplo seguinte são imutáveis são requeridos 'const_iterators':

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char **argv)
{
    vector<string> const args(argv, argv + argc);

    for
    (
        vector<string>::const_iterator iter = args.begin();
        iter != args.end();
    )
```

```

        ++iter
    )
    cout << *iter << " ";
    cout << endl;

    for
    (
        vector<string>::const_reverse_iterator iter = args.rbegin();
        iter != args.rend();
        ++iter
    )
        cout << *iter << " ";
    cout << endl;

    return 0;
}

```

O exemplo também ilustra que se pode usar ponteiros plenos no lugar de iteradores. A iniciação do 'vector<string> args(argv, argv + argc)' fornece os argumentos do vetor com um par de iteradores baseados em ponteiros: 'argv' aponta para o primeiro elemento para iniciar com 'sarg', 'argv + argc' aponta para justo depois do último elemento a ser usado, 'argv++' acessa a próxima 'string'. Esta é uma característica geral dos ponteiros, que justifica porque pode-se usar ponteiros em situações onde se esperaria um iterador.

A STL define cinco tipos de iteradores. Estes tipos se valem dos algoritmos genéricos e para estarmos capacitados de criar tipos particulares de iteradores é importante conhecer suas características. Em geral os iteradores devem definir:

- 'operator==()': Examina a igualdade entre dois iteradores;
- 'operator++()': Incrementa o iterador, como operador prefixado;
- 'operator*()': Acessa o elemento ao que se refere o iterador.

Os seguintes tipos de iteradores são usados na descrição de algoritmos genéricos mais adiante neste capítulo:

- 'InputIterators': Lem de um recipiente. O operador de 'dereference' funciona como 'rvalue' nas expressões. No lugar de um 'InputIterator' também é possível usar (ver abaixo) um 'Forward-', 'Bidirectional-' ou 'RandomAccessIterator' com os algoritmos genéricos apresentados no capítulo. As notações como 'InputIterator1' e 'InputIterator2' também aparecem. Nestes casos são usados números para indicar quais iteradores se completam. P.ex., a função genérica 'inner_product()' possui o seguinte protótipo:

```
Type inner_product(InputIterator1 first1, InputIterator1 last1,  
                  InputIterator2 first2, Type init);
```

Aqui 'InputIterator1 first1' e 'InputIterator1 last1' definem uma extensão de entrada, enquanto 'InputIterator2 first2' definem o início da segunda extensão. Uma notação análoga a esta é observada com outros tipos de iteradores.

- 'OutputIterators': São usados para escrever em recipientes. O operador de 'dereferense' funciona como um 'lvalue' em expressões, mas não necessariamente como um 'rvalue'. No lugar de um 'OutputIterator' é possível usar (veja abaixo) um 'Forward-', 'Bidirectional-' ou 'RandomAccessIterator'.
- 'ForwardIterators': Combina 'InputIterators' e 'OutputIterators'. São usados para percorrer recipientes numa direção, para leitura e/ou escrita. No lugar de 'ForwardIterators' é possível usar 'Bidirectional-' ou 'RandomAccessIterator'.
- 'BidirectionalIterators': Usados para percorrer recipientes em ambas direções, para leitura e escrita. No lugar de 'BidirectionalIterators' pode-se usar um 'RandomAccessIterator'. Por exemplo, para percorrer uma lista ou 'deque' um 'BidirectionalIterator' pode ser útil.
- 'RandomAccessIterators': Fornece acesso aleatório aos elementos de recipientes. Um algoritmos como 'sort()' requer um 'RandomAccessIterators' e não pode ser usado em listas e mapas, que só prevêm 'BidirectionalIterators'.

O exemplo dado para 'RandomAccessIterator' ilustra como solucionar a direção dos iteradores: Observa-se qual iterador é requerido pelo algoritmo (genérico) e então se examina se a estrutura de dados suporta o iterador requerido. Se não, o algoritmo não pode ser usado com o tipo particular de estrutura de dados.

17.2.1: Iteradores de Inserção

Os algoritmos genéricos freqüentemente requerem um recipiente alvo onde o resultado do algoritmo é depositado. Por exemplo, o algoritmo 'copy()' tem três parâmetros, os dois primeiros é a extensão dos elementos visitados e o terceiro define a primeira posição onde os resultados da operação de cópia vão ser armazenados. Com o algoritmo 'copy()' o número de elementos que são copiados normalmente é conhecido com antecipação, já que o número é freqüentemente determinado pela aritmética de apontadores. Contudo, existem situações onde não se pode empregar a aritmética de ponteiros. Analogamente o número de elementos resultantes difere do número inicial. o algoritmo

genérico 'unique_copy()' é o caso: O número de elementos copiados ao destino não é conhecido com anterioridade.

Nestas ocasiões uma função adaptadora de inserção pode ser usada para criar os elementos no recipiente de destino quando necessário. Existem três tipos de adaptadores de inserção:

- 'back_inserter()': Chama o membro do recipiente 'push_back()' para agregar um novo elemento no fim do recipiente. P.ex., para copiar todos os elementos da fonte em ordem inversa no destino:

```
copy(source.rbegin(), source.rend(), back_inserter(destination));
```

- 'front_inserter()': Chama o membro do recipiente 'push_front()' para agregar um novo elemento no início do recipiente. P.ex., para copiar todos os elementos da fonte no início do destino (assim também revertendo a ordem):

```
copy(source.begin(), source.end(), front_inserter(destination));
```

- 'inserter()': Chama o membro do recipiente 'insert()' para agregar um novo elemento começando num ponto determinado. P.ex., para copiar todos os elementos da fonte para o recipiente de destino, começando no início do destino, deslocando os elementos existentes para além dos elementos recém inseridos:

```
copy(source.begin(), source.end(), inserter(destination,
destination.begin()));
```

Concentrando-nos no 'back_inserter()', este iterador espera o nome de um recipiente com um membro 'push_back()'. Este membro é chamado pelo membro de inserção do operador(). Quando uma classe (que não seja um recipiente abstrato) suporta um recipiente com 'push_back()', seu objeto também pode ser usado como argumentos de 'back_inserter()', se a classe define, na sua interface, um:

```
typedef DataType const &const_reference;
```

Onde 'DataType const &' é o tipo de parâmetro da função membro da classe push_back(). Por exemplo, o seguinte programa define um esqueleto (compilável) de uma classe 'IntStore', cujos objetos podem ser usados como argumentos do iterador 'back_inserter':

```
#include <algorithm>
#include <iterator>
using namespace std;

class Y
{
public:
    typedef int const &const_reference;

    void push_back(int const &)
    {}
};
```



```

int main()
{
    int arr[] = {1};
    Y y;

    copy(arr, arr + 1, back_inserter(y));
}

```

17.2.2: Iteradores para objetos 'istream'

O 'istream_iterator<Type>()' pode ser usado para definir um (par) iterador para objetos 'istream'. A forma geral do iterador é:

```
istream_iterator<Type> identifier(istream &inStream)
```

Aqui 'Type' é o tipo do elemento de dados lido da 'istream'. O tipo pode ser qualquer tipo para o qual 'operator>>()' seja definido nos objetos 'istream'.

O construtor padrão define o fim do par de iteradores, correspondendo ao fim da 'stream'. Por exemplo:

```
istream_iterator<string> endOfStream;
```

Note que o objeto 'stream' que foi especificado para 'begin_iterator' não está mencionado aqui.

Usando um 'back_inserter()' e um conjunto de adaptadores 'istream_iterator<>()' todas as 'strings' são lidas de 'cin' como segue:

```

#include <algorithm>
#include <iterator>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> vs;

    copy(istream_iterator<string>(cin), istream_iterator<string>(),
        back_inserter(vs));

    for
    (
        vector<string>::iterator from = vs.begin();
        from != vs.end();
        ++from
    )

```

```

    )
    cout << *from << " ";
    cout << endl;

    return 0;
}

```

No exemplo acima, note o uso de versões anônimas dos adaptadores 'istream_iterator'. Especialmente, note o uso do construtor padrão anônimo. Em vez disso, usando 'istream_iterator<string>()' a seguinte construção (não anônima) poderia ter sido usada:

```

istream_iterator<string> eos;

copy(istream_iterator<string>(cin), eos, back_inserter(vs));

```

Antes de ser possível o uso dos 'istream_iterators' a seguinte diretiva ao pré-processador deve ser especificada:

```
#include <iterator>
```

Isto está implicado quando 'iostream' está incluído.

17.2.3: Iteradores para objetos 'istreambuf'

Os iteradores de entrada também estão disponíveis para objetos 'istreambuf'. Antes do uso deles deve-se especificar a seguinte diretiva ao pré-processador:

```
#include <iterator>
```

O 'istreambuf_iterator' é para leitura de um objeto 'istreambuf' que suporte operações de entrada. As operações padrão disponíveis para objetos 'istream_iterator' são também para 'istreambuf_iterators'. Há três construtores:

- o 'istreambuf_iterator<Type>()': Este construtor representa o iterador de fim da 'stream', quando extraíndo valores do tipo 'Type' do 'istreambuf';
- 'istreambuf_iterator<Type>(istream)': Este constrói um 'istreambuf_iterator' que acessa o 'istreambuf' do objeto 'istream', usado como construtor do argumento;
- 'istreambuf_iterator<Type>(istreambuf *)': Este constrói um 'istreambuf_iterator' que acessa o 'istreambuf' cujo endereço é usado como argumento do construtor.

17.2.4: Iteradores para objetos 'ostream'

O 'ostream_iterator<Type>()' pode ser usado para definir um iterador de destino para um objeto 'ostream'. As formas gerais do 'ostream_iterator<Type>()' são:

```
ostream_iterator<Type> identifier(ostream &outStream), // e:
ostream_iterator<Type> identifier(ostream &outStream, char const *delim);
```

'Type' é o tipo dos elementos de dados a serem escritos em 'ostream'. O tipo pode ser qualquer tipo definido para 'operator<<()' em combinações com objetos 'ostream'. A última forma dos iteradores 'ostream' separa os tipos de dados individuais com 'strings' como delimitadoras. A definição de formação não usa delimitadores.

O exemplo seguinte mostra como 'istream_iterator' e 'ostream_iterator' podem ser para copiar informações de um arquivo a outro. Uma sutileza é o comando 'in.unsetf(ios::skipws)': zera a flag 'ios::skipws'. A consequência disto é que o comportamento padrão de 'operator>>()' de saltar espaços em branco se modificará. Os espaços em brancos serão retornados pelo operador e o arquivo é copiado irrestritamente. Eis o programa:

```
#include <algorithm>
#include <fstream>
#include <iomanip>
using namespace std;

int main(int argc, char **argv)
{
    ifstream in(argv[1]);
    ofstream out(argv[2]);

    in.unsetf(ios::skipws);
    copy(istream_iterator<char>(in), istream_iterator<char>(),
        ostream_iterator<char>(out));

    return 0;
}
```

Antes de se usar 'ostream_iterator' a seguinte diretiva ao pré-processador deve ser especificada:

```
#include <iterator>
```

17.2.4.1: Iteradores para objetos 'ostreambuf'

Antes de se usar 'ostreambuf_iterator' a seguinte diretiva ao pré-processador deve ser especificada:

```
#include <iterator>
```

O 'ostreambuf_iterator' é para escrever em objetos 'streambuf' que suportem operações de saída. As operações padrão válidas para objetos 'ostream_iterator' também são válidas para 'ostreambuf_iterator'. Possui dois construtores:

- 'ostreambuf_iterator<Type>(ostream)': Este constrói um 'ostreambuf_iterator' que acessa o 'streambuf' do objeto 'ostream', usado como argumento do construtor para inserir valores de tipo 'Type';
- 'ostreambuf_iterator<Type>(streambuf *)': Constrói um 'ostreambuf_iterator' que acessa o 'streambuf' cujo endereço é usado como argumento do construtor.

Eis um exemplo usando ambos, 'istream_iterator' e 'ostream_iterator', mostrando mais um modo de copiar uma 'stream':

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    istreambuf_iterator<char> in(cin.rdbuf());
    istreambuf_iterator<char> eof;
    ostreambuf_iterator<char> out(cout.rdbuf());

    copy(in, eof, out);

    return 0;
}
```

17.3: A Classe 'auto_ptr'

Um dos problemas no uso de ponteiros é a estrita contabilidade requerida sobre suas memórias e tempo de vida. Quando uma variável apontadora cai fora de escopo a memória apontada fica de repente inacessível e o programa sofre uma fuga de memória. Por exemplo, a seguinte função 'fun()' cria uma fuga de memória a cada chamada a ela: o valor inteiro alocado permanece inacessível e alocado:

```
void fun()
{
    new int;
}
```

Para prevenir perda de memória se requer uma contabilidade estrita: o programador deve

assegurar-se que a memória apontada por um ponteiro seja liberada justo antes do ponteiro sair do escopo. No exemplo acima a reparação poderia ser:

```
void fun()
{
    delete new int;
}
```

Agora 'fun()' só gasta um instante de tempo.

Quando um apontador aponta para um simples valor ou objeto, os requerimentos de controle podem ser relaxados quando a variável apontadora é definida como um objeto 'std::auto_ptr'. Os 'auto_ptr' são objetos mascarados de ponteiros. Como são objetos, seus destrutores são chamados quando saem de escopo e, por isso, seus destrutores tomam a responsabilidade de liberar dinamicamente a memória alocada.

Antes de se usar 'auto_ptr' deve-se especificar a seguinte diretiva ao pré-processador:

```
#include <memory>
```

Normalmente um objeto 'auto_ptr' é iniciado usando valores ou objetos dinamicamente.

As seguintes restrições se aplicam aos 'auto_ptr':

- O objeto 'auto_ptr' não pode ser usado para apontar para objetos conjuntos;
- Um objeto 'auto_ptr' só pode apontar para memória dinâmica, pois somente esta pode ser liberada dinamicamente.
- Muitos objetos 'auto_ptr' não devem apontar para o mesmo bloco de memória alocado dinamicamente. A interface de 'auto_ptr' foi projetada para evitar isto. Uma vez que um objeto 'auto_ptr' sai do escopo, sua memória é liberada, imediatamente mudando qualquer outro objeto que também aponte para a memória alocada num ponteiro amplo.

A classe 'auto_ptr' define diversas funções para acessar o ponteiro ou para apontar a outro bloco de memória. Estas funções membro e meios para construir objetos 'auto_ptr' são discutidos na próxima seção.

17.3.1: Definindo variáveis 'auto_ptr'

Existem três maneiras de definir objetos 'auto_ptr'. Cada definição tem a especificação de tipo habitual entre parênteses angulares. São dados exemplos concretos nas seções seguintes, mas uma

visão geral está aqui:

- A forma básica inicia um objeto 'auto_ptr' para apontar um bloco de memória alocada pelo operador 'new':

```
auto_ptr<type> identifier (new-expression); Esta forma está em 17.3.2;
```

- Outra forma inicia um objeto 'auto_ptr' usando um construtor de cópias:

```
auto_ptr<type> identifier(outro auto_ptr como tipo); seção 17.3.3;
```

- A terceira forma simplesmente cria um objeto auto_ptr que não aponta para um bloco particular de memória:

```
auto_ptr<type> identifier; seção 17.3.4
```

17.3.2: Apontando para um objeto recém alocado

A forma básica de iniciar um objeto 'auto_ptr' é fornecer seu construtor com o bloco de memória alocada através do operador 'new':

```
auto_ptr<type> identifier (new-expression);
```

Por exemplo, para iniciar um 'auto_ptr' para apontar a uma 'string' se usa o seguinte construtor:

```
auto_ptr<string> strPtr(new string("Hello world"));
```

Para iniciar um 'auto_ptr' para apontar um valor duplo:

```
auto_ptr<double> dPtr(new double(123.456));
```

Note o uso do operado 'new' nas expressões acima. Usando 'new' asseguramos a natureza dinâmica da memória apontada pelo objeto 'auto_ptr' e permite a liberação da memória uma vez o objeto 'auto_ptr' fora de escopo. Note também que o tipo não contém o ponteiro: O tipo usado na construção de 'auto_ptr' é igual ao usado na expressão 'new'.

No exemplo que aloca um inteiro da seção 17.3, a fuga de memória pode ser evitada usando um objeto 'auto_ptr':

```
#include <memory>
using namespace std;

void fun()
{
    auto_ptr<int> ip(new int);
}
```

Todas as funções membro disponíveis para objetos alocados pela expressão 'new' podem ser

acessadas via 'auto_ptr' como se fosse um ponteiro pleno para o objeto alocado dinamicamente. Por exemplo, no seguinte programa o texto 'C++' é inserido atrás da palavra 'hello':

```
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    auto_ptr<string> sp(new string("Hello world"));

    cout << *sp << endl;

    sp->insert(strlen("Hello "), "C++ ");
    cout << *sp << endl;
}
/*
Saída Produzida:

Hello world
Hello C++ world
*/
```

17.3.3: Apontando para outro 'auto_ptr'

Um 'auto_ptr' pode também ser iniciado por outro objeto 'auto_ptr' para o mesmo tipo. A forma genérica é:

```
auto_ptr<type> identifier(outro auto_ptr object);
```

Por exemplo, para iniciar um 'auto_ptr<string>', dada a variável 'sp' definida na seção anterior, pode-se usar a seguinte construção:

```
auto_ptr<string> strPtr(sp);
```

Analogamente, o operador de adjudicação pode ser usado. Um objeto 'auto_ptr' pode ser adjudicado a outro objeto 'auto_ptr' do mesmo tipo. Por exemplo:

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;

int main()
{
    auto_ptr<string> hello1(new string("Hello world"));
    auto_ptr<string> hello2(hello1);
    auto_ptr<string> hello3;
```

```

        hello3 = hello2;
        cout << *hello1 << endl <<
                *hello2 << endl <<
                *hello3 << endl;
    }
    /*
        Saída Produzida:

        Segmentation fault
    */

```

Observando o exemplo acima vemos que:

- 'hello1' é iniciada como descrito na seção anterior;
- A seguinte 'hello2' é definida e recebe seu valor de 'hello1', usando o construtor de cópia. Isto efetivamente muda 'hello1' para um ponteiro nulo;
- Então 'hello3' é definida como um 'auto_ptr<string>' padrão e recebe seu valor através da adjudicação da 'hello2', que também se torna um ponteiro nulo.

O programa gera uma falha de segmento. A razão é clara: é causada por referência a ponteiros nulos. No fim, somente 'hello3' aponta para uma 'string'.

17.3.4: Criando um 'auto_ptr' pleno

Já vimos a terceira forma de criação de um objeto 'auto_ptr': Sem argumento, um objeto 'auto_ptr' vazio é construído sem apontar para um bloco de memória determinado:

```
auto_ptr<type> identifier
```

Neste caso o ponteiro é posto em zero. Como o 'auto_ptr' não é um ponteiro, seu valor não pode ser comparado a '0' para ver se foi iniciado. P.ex., código como:

```

auto_ptr<int> ip;

if (!ip)
    cout << "Ponteiro nulo como um objeto 'auto_ptr'?" << endl;

```

Não produzirá nenhuma saída (tão pouco compilará...). Assim, como inspecionar o valor do ponteiro mantido pelo objeto 'auto_ptr'? Para isto o membro 'get()' está disponível. Esta função membro, bem como as outras funções membro da classe 'auto_ptr' estão descritas na seção seguinte.

17.3.5: Operadores e membros

Os seguintes operadores são definidos na classe 'auto_ptr':

- `auto_ptr &auto_ptr<Type>operator=(auto_ptr<Type> &other):` Este operador transfere apontada a memória apontada pelo 'rvalue' do objeto 'auto_ptr' para o 'rvalue' do objeto 'auto_ptr'. Assim, o 'rvalue' perde a memória apontada e se torna um ponteiro vazio (zero).
- `Type &auto_ptr<Type>operator*():` Retorna uma referência à informação guardada no objeto 'auto_ptr'. Atua como um operador normal de um apontador de referência.
- `Type *auto_ptr<Type>operator->():` Retorna um apontador à informação guardada no objeto 'auto_ptr'. Através deste operador membros do objeto guardado podem ser selecionados. Por exemplo:

```
auto_ptr<string> sp(new string("hello"));  
  
cout << sp->c_str() << endl;
```

As seguintes funções membro são definidas para objetos 'auto_ptr':

- `Type *auto_ptr<Type>::get():` Este operador faz o mesmo que 'operator->()': Retorna um apontador para a informação guardada no objeto 'auto_ptr'. Este ponteiro pode ser inspecionado: Se zero o objeto 'auto_ptr' não aponta a qualquer memória. Este membro não pode ser usado para fazer o objeto 'auto_ptr' apontar para (outro) bloco de memória;
- `Type *auto_ptr<Type>::release():` Retorna um ponteiro à informação guardada no objeto 'auto_ptr', que perde a memória para a que apontava (e fica um apontador nulo). O membro pode ser usado para transferir a informação guardada num objeto 'auto_ptr' para um ponteiro pleno. É responsabilidade do programador liberar a memória retornada por esta função;
- `void auto_ptr<Type>::reset(Type *):` Este operador também pode ser chamado sem argumento para liberar a memória guardada no objeto 'auto_ptr' ou com um ponteiro a um bloco de memória dinamicamente alocada, que será a memória acessada pelo objeto 'auto_ptr'. Esta função membro pode ser usada para adjudicar um novo bloco de memória (novo conteúdo) a um objeto 'auto_ptr'.

17.3.6: Construtores e membros de dados ponteiros

Agora que as características principais dos objetos 'auto_ptr' foram descritas, considere a seguinte classe simples:

```
// required #includes

class Map
{
    std::map<string, Data> *d_map;
public:
    Map(char const *filename) throw(std::exception);
};
```

O construtor da classe 'Map()' realiza as seguintes tarefas:

- Aloca um objeto 'std::map';
- Abre o arquivo cujo nome é dado como argumento do construtor;
- Lê o arquivo, preenchendo o mapa.

É claro que pode não ser possível abrir o arquivo. Nesse caso é lançada uma exceção apropriada. Assim, a implantação do construtor examinará algo assim:

```
Map::Map(char const *fname)
:
    d_map(new std::map<std::string, Data>) throw(std::exception)
{
    ifstream istr(fname);
    if (!istr)
        throw std::exception("Não pode abrir o arquivo");
    fillMap(istr);
}
```

O que há de errado com esta implantação? Sua maior debilidade é que abriga uma perda potencial de memória. A perda de memória ocorre quando lança a exceção. Em todos os outros casos a função opera perfeitamente bem. Quando a exceção é lançada, o mapa acaba de ser alocado dinamicamente. Contudo, há dúvidas se o destrutor da classe chamará 'delete d_map', o destrutor, aqui, não é chamado para destruir objetos que foram completamente construídos. Como o construtor termina em exceção, seu objeto associado não foi completamente construído e, portanto, o destrutor desse objeto nunca é chamado.

Objetos 'auto_ptr' podem ser usados para prevenir esta classe de problemas. Definindo 'd_map' assim:

```
std::auto_ptr<std::map<std::string, Data> >
```

Imediatamente se transforma num objeto. Agora o construtor de 'Map' pode com segurança lançar a exceção. Como 'd_map' é um objeto, seu destrutor será chamado no momento (apesar de completamente construído) em que o objeto 'Map' sair do escopo.

Como regra prática: As classes devem usar objetos 'auto_ptr' antes que ponteiros plenos aos seus membros de dados se há qualquer chance de seu construtor terminar prematuramente numa exceção.

17.4: O Algoritmo Genérico

Esta seção descreve os algoritmos genéricos em ordem alfabética. Para cada algoritmo a seguinte informação é fornecida:

- O arquivo cabeçalho requerido;
- O protótipo da função;
- Uma descrição curta;
- Um pequeno exemplo.

Nos protótipos dos algoritmos 'Type' é usado para especificar um tipo genérico. Também o tipo particular de iterador (veja a seção 17.2) requerido é mencionado, bem como outros tipos genéricos que possam ser requeridos (p.ex., que realizem operações binárias como 'plus<Type>()').

Quase todos os algoritmos genéricos esperam um iterador de extensão dos elementos '[first, last)' sobre os quais o algoritmo opera. Os iteradores apontam para objetos ou valores. Quando um iterador aponta para um tipo de valor ou objeto, a função objeto usada pelo algoritmo usualmente recebe 'Type const &(objetos ou valores)': as funções objetos podem portanto não modificar os objetos que recebem como argumentos. Isto não é verdadeiro para algoritmos genéricos modificadores, que são (claro está) capazes de modificar os objetos sobre os quais operam.

Os algoritmos genéricos podem ser categorizados. Nas Anotações C++ distinguimos as seguintes categorias:

- Comparadores: Comparam (extensões de) elementos:

```
Requer: #include <algorithm>

equal();

includes();
lexicographical_compare();
max();
min();
mismatch();
```

- Copiadores: realizam operações de cópia:

Requer: `#include <algorithm>`

```
copy();
    copy_backward();
    partial_sort_copy();
    remove_copy();
    remove_copy_if();
    replace_copy();
    replace_copy_if();
    unique_copy();
```

Contadores: realizam operações de contagem:

Requer: `#include <algorithm>`

```
count();
    count_if();
```

- Empilhadores: manipulam pilhas:

Requer: `#include <algorithm>`

```
make_heap();
    pop_heap();
    push_heap();
    sort_heap();
```

Iniciadores: iniciam dados:

Requer: `#include <algorithm>`

```
fill();
    fill_n();
    generate();
    generate_n();
```

- Operadores: realizam operações aritméticas de algum tipo:

Requer: `#include <numeric>`

```
adjacent_difference();
    accumulate();
    inner_product();
    partial_sum();
```

- Buscadores: realizam operações de busca (e encontro):

Requer: `#include <algorithm>`

```

adjacent_find();
    binary_search();
    equal_range();
    find();
    find_if();
    find_end();
    find_first_of();
    lower_bound();
    max_element();
    min_element();
    search();
    search_n();
    set_difference();
    set_intersection();
    set_symmetric_difference();
    set_union();
    upper_bound();

```

- **Ordenadores:** realizam operações de re-ordenação (ordenação, fusão, permutação, embaralhamento, troca):

Requer: #include <algorithm>

```

inplace_merge();
    iter_swap();
    merge();
    next_permutation();
    nth_element();
    partial_sort();
    partial_sort_copy();
    partition();
    prev_permutation();
    random_shuffle();
    remove();
    remove_copy();
    remove_if();
    remove_copy_if();
    reverse();
    reverse_copy();
    rotate();
    rotate_copy();
    sort();
    stable_partition();
    stable_sort();
    swap();
    unique();

```

- **Visitantes:** visitam elementos dentro de uma extensão:

Requer: #include <algorithm>

```

for_each();
    replace();
    replace_copy();
    replace_if();
    replace_copy_if();
    transform();
    unique_copy();

```

17.4.1: accumulate()

- Arquivo cabeçalho:

```
#include <numeric>
```

- Protótipos da função:

- `Type accumulate(InputIterator first, InputIterator last, Type init);`
- `Type accumulate(InputIterator first, InputIterator last, Type init, BinaryOperation op);`

- Descrição:

- Primeiro protótipo: 'operator+()' aplicado a todos os elementos implicados pelo iterador de extensão e pelo valor inicial 'init'. O valor resultante é retornado.
- Segundo protótipo: O operador binário 'op()' é aplicado a todos os elementos implicados pelo iterador de extensão e pelo valor inicial 'init' e retorna o resultado.

- Exemplo:

```

#include<numeric>
#include<vector>
#include<iostream>
using namespace std;

int main()
{
    int        ia[] = {1, 2, 3, 4};
    vector<int> iv(ia, ia + 4);

    cout <<
        "Soma de valores: " << accumulate(iv.begin(), iv.end(), int()) <<
        endl <<
        "Produto de valores: " << accumulate(iv.begin(), iv.end(), int(1),
                                                multiplies<int>()) << endl;

    return 0;
}

```

```

}
/*
    Saída Gerada:

    Soma de valores: 10
    Produto de valores: 24
*/

```

17.4.2: adjacent_difference()

- Arquivo cabeçalho:

```
#include <numeric>
```

- Protótipos da função:

- `OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);`
- `OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);`

- Descrição:

Todas as operações são realizadas sobre os valores originais, todos os valores computados são retornados.

- Primeiro protótipo: O primeiro elemento retornado é igual ao primeiro elemento da extensão de entrada. Os elementos remanescentes são iguais à diferença do elemento correspondente na entrada e seu elemento anterior.
- Segundo protótipo: O primeiro elemento retornado é igual ao primeiro elemento da extensão de entrada. Os elementos restantes são iguais ao resultado do operador binário 'op' aplicado ao elemento correspondente da entrada (operando da esquerda) e seu elemento anterior (operando da direita).
- Exemplo:

```

#include<numeric>
#include<vector>
#include<iostream>
using namespace std;

int main()
{
    int                ia[] = {1, 2, 5, 10};

```

```

vector<int>      iv(ia, ia + 4);
vector<int>      ov(iv.size());

adjacent_difference(iv.begin(), iv.end(), ov.begin());

copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
cout << endl;

adjacent_difference(iv.begin(), iv.end(), ov.begin(), minus<int>());

copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
cout << endl;

return 0;
}
/*
Saída Gerada:

1 1 3 5
1 1 3 5
*/

```

17.4.3: adjacent_find()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- `ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);`
- `OutputIterator adjacent_find(ForwardIterator first, ForwardIterator last, Predicate pred);`

- Descrição:

- Primeiro protótipo: O iterador que aponta para o primeiro elemento do primeiro par de dois elementos adjacentes iguais é retornado. Se não existir tal elemento o último é retornado.
- Segundo protótipo: O iterador que aponta para o primeiro elemento do primeiro par de dois elementos adjacentes para o qual o predicado binário 'pred' é verdadeiro é retornado. Se tal elemento não existe, o último é retornado.

- Exemplo:

```
#include<algorithm>
#include<string>
#include<iostream>

class SquaresDiff
{
    size_t d_minimum;

public:
    SquaresDiff(size_t minimum)
    :
        d_minimum(minimum)
    {}
    bool operator()(size_t first, size_t second)
    {
        return second * second - first * first >= d_minimum;
    }
};

using namespace std;

int main()
{
    string sarr[] =
    {
        "Alpha", "bravo", "charley", "delta", "echo", "echo",
        "foxtrot", "golf"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    string *result = adjacent_find(sarr, last);

    cout << *result << endl;
    result = adjacent_find(++result, last);

    cout << "Segunda vez, começa da posição seguinte:\n" <<
        (
            result == last ?
                "*** Não há mais elementos adjacentes iguais ***"
            :
                "*result"
        ) << endl;

    size_t iv[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    size_t *ilast = iv + sizeof(iv) / sizeof(size_t);
    size_t *ires = adjacent_find(iv, ilast, SquaresDiff(10));

    cout <<
        "Os primeiros números com diferença dos quadrados pelo menos 10: "
        << *ires << " e " << *(ires + 1) << endl;
```

```

    return 0;
}
/*
    Saída Gerada:

    echo
    Segunda vez, começa da posição seguinte:
    ** Não há mais elementos adjacentes iguais **
    Os primeiros números com diferença dos quadrados pelo menos 10: 5 e 6
*/

```

17.4.4: binary_search()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value);`
- `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value, Comparator comp);`

- Descrição:

- Primeiro protótipo: O valor é buscado usando busca binária na extensão dos elementos implicados pelo iterador '[first, last)'. Os elementos dentro da extensão devem estar ordenados pela função 'Type::operator<()'. Será retornado verdadeiro se o elemento for encontrado e falso se não.
- Segundo protótipo: O valor é buscado usando a busca binária dentro da extensão do iterador '[first, last)'. Os elementos dentro da extensão devem estar ordenados pela função objeto de comparação. Retorna verdadeiro se achar o elemento, do contrário retorna falso.

- Exemplo:

```

#include <algorithm>
#include <string>
#include <iostream>
#include <functional>
using namespace std;

int main()

```

```

{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    bool result = binary_search(sarr, last, "foxtrot");

    cout << (result ? "encontrou " : "não encontrou ") << "foxtrot" << endl;

    reverse(sarr, last);                // reverse the order of elements
                                        // binary search now fails:
    result = binary_search(sarr, last, "foxtrot");
    cout << (result ? "encontrou " : "não encontrou ") << "foxtrot" << endl;
                                        // ok when using appropriate
                                        // comparator:
    result = binary_search(sarr, last, "foxtrot", greater<string>());
    cout << (result ? "encontrou " : "não encontrou ") << "foxtrot" << endl;

    return 0;
}
/*
Saída Gerada:

encontrou foxtrot
não encontrou foxtrot
encontrou foxtrot
*/

```

17.4.5: copy()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
OutputIterator copy(InputIterator first, InputIterator last,
OutputIterator destination);
```

- Descrição:

- A extensão dos elementos implicados pelo iterador '[first, last)' é copiada a uma extensão de saída, começando em 'destination', usando o operador de adjudicação do tipo de dados. O valor de retorno é o 'OutputIterator' que aponta justo além do último elemento que foi copiado para o destino (o 'last' (último) na extensão de saída é retornado).

- Exemplo:

Note a segunda chamada a 'copy()'. Usa um 'ostream_iterator' para objetos 'string'. Este iterador escreve os valores da 'string' na 'ostream' especificada (i.e., 'cout'), separando os valores com a 'string' especificada (i.e., “ ”).

```
#include <algorithm>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy(sarr + 2, last, sarr); // move todos os elementos duas posições à
                                // esquerda

                                // copia para cout usando um
                                // ostream_iterator
para strings
    copy(sarr, last, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

charley delta echo foxtrot golf hotel golf hotel
*/
```

- Veja também:

```
unique_copy()
```

17.4.6: copy_backward()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- **Protótipos da função:**

- `BidirectionalIterator copy_backward(InputIterator first, InputIterator last, BidirectionalIterator last2);`

- **Descrição:**

- A extensão de elementos implicados pelo iterador '[first, last)' é copiada do elemento 'last - 1' até (e incluso) o elemento 'first', na posição 'first' até 'last2 - 1', usando o operador de adjudicação do tipo de dados requerido. A extensão do destino, portanto, é '[last2 - (last - first), last2)'. O valor de retorno é o 'BidirectionalIterator' que aponta para o último elemento copiado ao destino (assim, 'first' no destino, apontado por 'last2 - (last - first)').

- **Exemplo:**

```
#include <algorithm>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        copy_backward(sarr + 3, last, last - 3),
        last,
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
Saída Gerada:

golf hotel foxtrot golf hotel foxtrot golf hotel
*/
```

17.4.7: count()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
• size_t count(InputIterator first, InputIterator last, Type const  
  &value);
```

- Descrição:

- O número de vezes que ocorre valor 'value' na extensão do iterador 'first, last' é retornado. Para determinar se o valor é igual a um elemento na extensão do iterador, é usado 'Type::operator==()'.

- Exemplo:

```
#include<algorithm>
#include<iostream>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "Número de vezes que o valor 3 aparece: " <<
        count(ia, ia + sizeof(ia) / sizeof(int), 3) <<
        endl;

    return 0;
}
/*
Saída Gerada:

Número de vezes que o valor 3 aparece: 3
*/
```

17.4.8: count_if()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
• size_t count_if(InputIterator first, InputIterator last, Predicate  
  predicate);
```

- Descrição:

- O número de vezes que o predicado unário 'predicate' retorna verdadeiro quando aplicado aos elementos implicados pelo iterador 'first, last', é retornado.

- Exemplo:

```
#include<algorithm>
#include<iostream>

class Odd
{
public:
    bool operator()(int value)
    {
        return value & 1;
    }
};

using namespace std;

int main()
{
    int    ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "O número de valores ímpares no conjunto é: " <<
        count_if(ia, ia + sizeof(ia) / sizeof(int), Odd()) << endl;

    return 0;
}
/*
Saída Gerada:

O número de valores ímpares no conjunto é: 5
*/
```

17.4.9: equal()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- `bool equal(InputIterator first, InputIterator last, InputIterator otherFirst);`
- `bool equal(InputIterator first, InputIterator last, InputIterator otherFirst, BinaryPredicate pred);`

- Descrição:

- Primeiro protótipo: Os elementos na extensão '[first, last)' são comparados com uma extensão igual começando em 'otherFirst'. A função retorna verdadeira se os elementos visitados são iguais em pares. As extensões não necessitam serem iguais, somente os elementos na extensão indicada são considerados (e precisam existir).
- Segundo protótipo: Os elementos na extensão '[first, last)' são comparados a uma extensão igual começando em 'otherFirst'. A função retorna verdadeira se o predicado binário, aplicado a todos os elementos de ambas extensões retorna verdadeiro para todos os pares correspondentes de elementos. As extensões não necessitam serem iguais, somente os elementos indicados pela extensão são considerados (e precisam existir).

- Exemplo:

```
#include <algorithm>
#include <string>
#include <iostream>

class CaseString
{
public:
    bool operator()(std::string const &first,
                    std::string const &second) const
    {
        return !strcasecmp(first.c_str(), second.c_str());
    }
};

using namespace std;

int main()
{
    string first[] =
    {
        "Alpha", "bravo", "Charley", "delta", "Echo",
        "foxtrot", "Golf", "hotel"
    };
    string second[] =
```



```

        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel"
    };
    string *last = first + sizeof(first) / sizeof(string);

    cout << "Os elementos de 'first' e 'second' são pares " <<
        (equal(first, last, second) ? "iguais" : "desiguais") <<
        endl <<
        "comparados caso maiúscula/minúscula iguais, são " <<
        (
            equal(first, last, second, CaseString()) ?
                "iguais" : "desiguais"
        ) << endl;

    return 0;
}
/*
Saída Gerada:

Os elementos de 'first' e 'second' são pares  desiguais
comparados caso maiúscula/minúscula iguais, são  iguais
*/

```

17.4.10: equal_range()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- `pair<ForwardIterator, ForwardIterator> equal_range (ForwardIterator first, ForwardIterator last, Type const &value);`
- `pair<ForwardIterator, ForwardIterator> equal_range (ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);`

- Descrição:

- (Veja também funções membro com nomes idênticos de, p.ex., o mapa (seção 12.3.6) e multimapa (seção 12.3.7)):
- Primeiro protótipo: Começando de uma seqüência ordenada (onde o operador 'operator<()' do tipo de dados para o qual os iteradores apontam foi usado para ordenar os elementos na extensão fornecida), um par de iteradores é retornado que representa o valor retornado de, respectivamente, 'lower_bound()' (limite inferior) (que retorna o primeiro

elemento que não é menor que a referência, veja seção 17.4.25) e 'upper_bound()' (que retorna o primeiro elemento além do valor de referência, veja seção 17.4.66).

- Segundo protótipo: Começando de uma sequência ordenada (onde a função objeto de comparação foi usada para ordenar os elementos), um par de iteradores é retornado representando o valor de retorno de, respectivamente, 'lower_bound()' (limite inferior) (seção 17.4.25) e 'upper_bound()' (limite superior) (seção 17.4.66).
- Exemplo:

```
#include <algorithm>
#include <functional>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    int                range[] = {1, 3, 5, 7, 7, 9, 9, 9};
    size_t const       size = sizeof(range) / sizeof(int);

    pair<int *, int *> pi;

    pi = equal_range(range, range + size, 6);

    cout << "Limite inferior para 6: " << *pi.first << endl;
    cout << "Limite superior para 6: " << *pi.second << endl;

    pi = equal_range(range, range + size, 7);

    cout << "Limite inferior para 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Limite superior para 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    sort(range, range + size, greater<int>());

    cout << "Ordenada em ordem decrescente\n";

    copy(range, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    pi = equal_range(range, range + size, 7, greater<int>());

    cout << "Limite inferior para 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
```

```

    cout << endl;

    cout << "Limite superior para 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

    Limite inferior para 6: 7
    Limite superior para 6: 7
    Limite inferior para 7: 7 7 9 9 9
    Limite superior para 7: 9 9 9
    Ordenada em ordem decrescente
    9 9 9 7 7 5 3 1
    Limite inferior para 7: 7 7 5 3 1
    Limite superior para 7: 5 3 1
*/

```

17.4.11: fill()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
• void fill(ForwardIterator first, ForwardIterator last, Type const
&value);
```

- Descrição:

- Todos os elementos implicados pela extensão do iterador '[first, last)' são iniciados com o valor, sobrepondo-se aos valores anteriormente armazenados.

- Exemplo:

```

#include<algorithm>
#include<vector>
#include<iterator>
#include<iostream>
using namespace std;

int main()
{

```

```

vector<int>      iv(8);

fill(iv.begin(), iv.end(), 8);

copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
cout << endl;

return 0;
}
/*
Saída Gerada:

8 8 8 8 8 8 8 8
*/

```

17.4.12: fill_n()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
• void fill_n(ForwardIterator first, Size n, Type const &value);
```

- Descrição:

- Os n elementos começando pelo elemento apontado por 'first' são iniciados com o valor dado, sobrescrevendo o antigo valor guardado.

- Exemplo:

```

#include<algorithm>
#include<vector>
#include<iterator>
#include<iostream>
using namespace std;

int main()
{
    vector<int>      iv(8);

    fill_n(iv.begin() + 2, 4, 8);

    copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}

```

```

        return 0;
    }
    /*
        Saída Gerada:

        0 0 8 8 8 8 0 0
    */

```

17.4.13: find()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

    InputIterator find(InputIterator first, InputIterator last, Type const
    &value);

```

- Descrição:

- O elemento com valor 'value' é buscado na extensão de elementos dada pelo iterador '[first, last)'. É retornado um iterador que aponta para o primeiro elemento encontrado. Se não for encontrado é retornado o último. O operador 'operator==()' do tipo de dados requerido é usado para comparar os elementos.

- Exemplo:

```

#include<algorithm>
#include<string>
#include<iterator>
#include<iostream>
using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find(sarr, last, "delta"), last, ostream_iterator<string>(cout, " ")

```

```

);
cout << endl;

if (find(sarr, last, "india") == last)
{
    cout << "'india' não foi encontrada na extensão\n";
    copy(sarr, last, ostream_iterator<string>(cout, " "));
    cout << endl;
}

return 0;

}
/*
Saída Gerada:

delta echo
'india' não foi encontrada na extensão
alpha bravo charley delta echo
*/

```

17.4.14: find_end()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)`
- `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)`

- Descrição:

- Primeiro protótipo: Na sequência de elementos dada por '[first1, last1)' é buscada como última ocorrência a sequência de elementos '[first2, last2)'. Se a sequência '[first2, last2)' não for encontrada, 'last1' é retornado, de outra forma um iterador apontando para o primeiro elemento da sequência coincidente é retornado. O operador 'operator==()' do tipo de dados respectivo é usado para comparar os elementos das duas sequências.
- Segundo protótipo: Na sequência de elementos dada por '[first1, last1)' é buscada como última ocorrência a sequência de elementos '[first2, last2)'. Se a sequência '[first2, last2)'

não for encontrada, 'last1' é retornado, de outra forma um iterador apontando para o primeiro elemento da sequência coincidente é retornado. O predicado binário fornecido é usado para comparar os elementos das duas sequências.

- Exemplo:

```
#include<algorithm>
#include<string>
#include<iterator>
#include<iostream>

class Twice
{
public:
    bool operator()(size_t first, size_t second) const
    {
        return first == (second << 1);
    }
};

using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo", "foxtrot", "golf",
        "hotel", "foxtrot", "golf", "hotel", "india", "juliet", "kilo"
    };
    string search[] =
    {
        "foxtrot",
        "golf",
        "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_end(sarr, last, search, search + 3),    // sequência de início
        last, ostream_iterator<string>(cout, " ")    // em 2º 'foxtrot'
    );
    cout << endl;

    size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
    size_t nrs[]    = {2, 3, 4};

    copy                // sequência de valores começando pela última sequência
    (                    // de extensão[] que são o dobro dos valores em nrs[]
        find_end(range, range + 9, nrs, nrs + 3, Twice()),
```

```

        range + 9, ostream_iterator<size_t>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
Saída Gerada:

foxtrot golf hotel india juliet kilo
4 6 8 10
*/

```

17.4.15: find_first_of()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
ForwardIterator1 find_first_of(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)
```

```
ForwardIterator1 find_first_of(ForwardIterator1 first1,
ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred)
```

- Descrição:

- Primeiro protótipo: A sequência de elementos implicada por '[first1, last1)' é varrida em busca da primeira ocorrência de um elemento da sequência de elementos implicada por '[first2, last2)'. Se não se encontra nenhum elemento de '[first2, last2)' 'last1' é retornado, de outra forma um iterador apontando para o primeiro elementos em '[first1, last1)' que seja igual a um elemento em '[first2, last2)' é retornado. O 'operator==()' do tipo de dados apropriado é usado para comparar os elementos das duas sequências.
- Segundo protótipo: A sequência de elementos implicada por '[first1, last1)' é varrida em busca da primeira ocorrência de um elemento da sequência de elementos implicada por '[first2, last2)'. Cada elemento da extensão '[first1, last1)' é comparada a cada elemento de '[first2, last2)' e é retornado um iterador que aponta para o primeiro em '[first1, last1)' para o qual o predicado 'pred' (recebendo um elemento fora da extensão '[first1, last1)' e um elemento de '[first2, last2)') retorne verdadeiro. Do contrário 'last1' é retornado.

- Exemplo:


```

#include<algorithm>
#include<string>
#include<iterator>
#include<iostream>

class Twice
{
public:
    bool operator()(size_t first, size_t second) const
    {
        return first == (second << 1);
    }
};

using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel",
        "foxtrot", "golf", "hotel",
        "india", "juliet", "kilo"
    };
    string search[] =
    {
        "foxtrot",
        "golf",
        "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        // sequência de início
        find_first_of(sarr, last, search, search + 3), // no 1º 'foxtrot'
        last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
    size_t nrs[] = {2, 3, 4};

    copy
    (
        // sequência de valores iniciando na 1ª sequência
        // de range[] que tem valores o dobro de nrs[]
        find_first_of(range, range + 9, nrs, nrs + 3, Twice()),
        range + 9, ostream_iterator<size_t>(cout, " ")
    );
    cout << endl;

    return 0;
}

```

```

}
/*
    Saída Gerada:

    foxtrot golf hotel foxtrot golf hotel india juliet kilo
    4 6 8 10 4 6 8 10
*/

```

17.4.16: find_if()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

• InputIterator find_if(InputIterator first, InputIterator last,
    Predicate pred);

```

- Descrição:

- É retornado um iterador apontando para o primeiro elemento na extensão do iterador '[first, last)' para o qual o predicado (unário) retorna verdadeiro. Se não for encontrado o elemento, o último é retornado.

- Exemplo:

```

#include<algorithm>
#include<string>
#include<iterator>
#include<iostream>

class CaseName
{
    std::string d_string;

public:
    CaseName(char const *str): d_string(str)
    {}
    bool operator()(std::string const &element)
    {
        return !strcasecmp(element.c_str(), d_string.c_str());
    }
};

using namespace std;

```

```

int main()
{
    string sarr[] =
    {
        "Alpha", "Bravo", "Charley", "Delta", "Echo",
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_if(sarr, last, CaseName("charley")),
        last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    if (find_if(sarr, last, CaseName("india")) == last)
    {
        cout << "'india' não foi encontrado\n";
        copy(sarr, last, ostream_iterator<string>(cout, " "));
        cout << endl;
    }

    return 0;
}
/*
Saída Gerada:

Charley Delta Echo
'india' não foi encontrado
Alpha Bravo Charley Delta Echo
*/

```

17.4.17: for_each()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

• Function for_each(ForwardIterator first, ForwardIterator last,
    Function func);

```

- Descrição:

- Cada um dos elementos da extensão '[first, last)' passa em turno como referência para a

função (ou função objeto) 'func'. A função pode modificar os elementos que recebe (como o iterador usado é um iterador avançado). Alternativamente, se o elemento pode ser transformado, 'transform()' (veja seção 17.4.63) pode ser usado. A função ou uma cópia da função objeto fornecida é retornada: Ver exemplo abaixo, na qual uma lista extra de argumentos é agregada na chamada de 'for_each()', cujo argumento é eventualmente passado à função dado a 'for_each()'. Na 'for_each()' o valor retornado da função que é passado é ignorado.

Exemplo:

```
#include<algorithm>
#include<string>
#include<iostream>
#include<cctype>

void lowerCase(char &c)                                // 'c' *é* modificado
{
    c = static_cast<char>(tolower(c));
}

                                                                    // 'str' *não* é modificada
void capitalizedOutput(std::string const &str)
{
    char    *tmp = strcpy(new char[str.size() + 1], str.c_str());

    std::for_each(tmp + 1, tmp + str.size(), lowerCase);

    tmp[0] = toupper(*tmp);
    std::cout << tmp << " ";
    delete tmp;
};

using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "BRAVO", "charley", "DELTA", "echo",
        "FOXTROT", "golf", "HOTEL",
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    for_each(sarr, last, capitalizedOutput)("that's all, folks");
    cout << endl;

    return 0;
}
/*
Saída Gerada:
```

```
Alpha Bravo Charley Delta Echo Foxtrot Golf Hotel That's all, folks
*/
```

- Outro exemplo usando uma função objeto:

```
#include<algorithm>
#include<string>
#include<iostream>
#include<cctype>

void lowerCase(char &c)
{
    c = tolower(c);
}

class Show
{
    int d_count;

public:
    Show()
    :
        d_count(0)
    {}

    void operator()(std::string &str)
    {
        std::for_each(str.begin(), str.end(), lowerCase);
        str[0] = toupper(str[0]); // here assuming str.length()
        std::cout << ++d_count << " " << str << "; ";
    }

    int count() const
    {
        return d_count;
    }
};

using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "BRAVO", "charley", "DELTA", "echo",
        "FOXTROT", "golf", "HOTEL",
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    cout << for_each(sarr, last, Show()).count() << endl;
```

```

    return 0;
}
/*
Saída Gerada (tudo numa linha):

1 Alpha; 2 Bravo; 3 Charley; 4 Delta; 5 Echo; 6 Foxtrot;
                                           7 Golf; 8 Hotel; 8
*/

```

O exemplo mostra também que o algoritmo 'for_each()' pode ser usado com funções que definem parâmetros constantes e não constantes. Veja também a seção 17.4.63 para diferenças entre os algoritmos genéricos 'for_each()' e 'transform()'.

O algoritmo 'for_each()' não pode ser usado diretamente (i.e., passando '*this' como função objeto argumento) numa função membro para modificar seu próprio objeto já que o algoritmo 'for_each()' primeiro cria sua própria cópia da função objeto passada. Uma classe envolvente cujo construtor aceite um apontador ou referência ao objeto em questão e possivelmente a uma de suas funções membro resolve este problema. Na seção 20.7 a construção de tal classe envolvente é descrita.

17.4.18: generate()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
void generate(ForwardIterator first, ForwardIterator last, Generator
generator);
```

- Descrição:

- Todos os elementos implicados pelo iterador de extensão '[first, last)' são iniciados com o valor de retorno de 'generator', que pode ser uma função ou função objeto. Note que a 'Generator::operator()()' não é dado nenhum argumento. O exemplo usa um fato bem conhecido da álgebra: para se obter o quadrado de $n + 1$ se usa $n*n+(2*n)+1$.

- Exemplo:

```

#include<algorithm>
#include<vector>
#include<iterator>
#include<iostream>

```

```

class NaturalSquares
{
    size_t d_newsqr;
    size_t d_last;

public:
    NaturalSquares(): d_newsqr(0), d_last(0)
    {}
    size_t operator() ()
    {
        // usa: (a + 1)^2 == a^2 + 2*a + 1
        return d_newsqr += (d_last++ << 1) + 1;
    }
};

using namespace std;

int main()
{
    vector<size_t>    uv(10);

    generate(uv.begin(), uv.end(), NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Saída Gerada:

    1 4 9 16 25 36 49 64 81 100
*/

```

17.4.19: generate_n()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- `void generate_n(ForwardIterator first, Size n, Generator generator);`

- Descrição:

- Os n elementos começando pelo elemento apontado pelo iterador 'first' são iniciados com o valor retornado por 'generator', que pode ser uma função ou uma função objeto.

- Exemplo:

```
#include<algorithm>
#include<vector>
#include<iterator>
#include<iostream>

class NaturalSquares
{
    size_t d_newsqr;
    size_t d_last;

    public:
        NaturalSquares(): d_newsqr(0), d_last(0)
        {}
        size_t operator() ()
        {
            // usa: (a + 1)^2 == a^2 + 2*a + 1
            return d_newsqr += (d_last++ << 1) + 1;
        }
};

using namespace std;

int main()
{
    vector<size_t>    uv(10);

    generate_n(uv.begin(), 5, NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Saída Gerada:

    1 4 9 16 25 0 0 0 0 0
*/
```

17.4.20: includes()

- Arquivo cabeçalho:

```
#include <algorithm>
```


- **Protótipos da função:**

```

- bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);

- bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2, Compare
comp);

```

- **Descrição:**

- **Primeiro Protótipo:** Ambas seqüências de elementos implicadas pelas extensões '[first1, last1)' e '[first2, last2)' devem ser ordenadas, usando 'operator<()' do tipo de dados para os quais os iteradores apontam. A função retorna verdadeira se cada elemento da segunda seqüência está contido na primeira seqüência (o segundo é um subconjunto da primeira).
- **Segundo protótipo:** Ambas seqüências de elementos implicadas pelas extensões '[first1, last1)' e '[first2, last2)' devem ser ordenadas, usando-se a função objeto 'comp'. A função retorna verdadeira se cada elemento da segunda seqüência está contido na primeira (a segunda seqüência é um subconjunto da primeira).

- **Exemplo:**

```

#include <algorithm>
#include <string>
#include <iostream>

class CaseString
{
public:
    bool operator() (std::string const &first,
                    std::string const &second) const
    {
        return (!strcasecmp(first.c_str(), second.c_str()));
    }
};

using namespace std;

int main()
{
    string first1[] =
    {
        "alpha", "bravo", "charley", "delta", "echo",

```

```

        "foxtrot", "golf", "hotel"
    };
    string first2[] =
    {
        "Alpha", "bravo", "Charley", "delta", "Echo",
        "foxtrot", "Golf", "hotel"
    };
    string second[] =
    {
        "charley", "foxtrot", "hotel"
    };
    size_t n = sizeof(first1) / sizeof(string);

    cout << "Os elementos da 'segunda' " <<
        (includes(first1, first1 + n, second, second + 3) ? "" : "não")
        << " estão contidos na primeira seqüência:\n"
        << "a segunda é um subconjunto da 'first1'\n";

    cout << "Os elementos de 'first1' " <<
        (includes(second, second + 3, first1, first1 + n) ? "" : "não")
        << " estão contidos na 'segunda' seqüência\n";

    cout << "Os elementos da 'second' " <<
        (includes(first2, first2 + n, second, second + 3) ? "" : "não")
        << " estão contidos na first2 seqüência\n";

    cout << "Usando comparação com insensibilidade de caso,\n"
        << "os elementos de 'second' "
        <<
        (includes(first2, first2 + n, second, second + 3, CaseString()) ?
            "" : "não")
        << " estão contidos na seqüência first2\n";

    return 0;
}
/*
Saída Gerada:

Os elementos da 'segunda' estão contidos na primeira seqüência:
a segunda é um subconjunto da 'first1'
Os elementos de 'first1' não estão contidos na 'segunda' seqüência
Os elementos da 'second' não estão contidos na first2 seqüência
Usando comparação com insensibilidade de caso,
os elementos de 'second' estão contidos na seqüência first2
*/

```

17.4.21: inner_product()

- Arquivo cabeçalho:

```
#include <numeric>
```

- **Protótipos da função:**

```
    - Type inner_product(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, Type init);
```

```
    - Type inner_product(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, Type init, BinaryOperator1 op1, BinaryOperator2 op2);
```

- **Descrição:**

- **Primeiro protótipo:** A soma de todos os produtos de pares de elementos dentro da extensão '[first1, last1)' e o mesmo número de elementos começando em 'first2' é adicionada a 'init' e esta soma é retornada. A função usa 'operator+()' e 'operator*()' para o tipo de dados para o que os iteradores apontam.
- **Segundo protótipo:** No lugar do operador padrão soma é usado o operador binário 'op2' e o operador binário 'op1' no lugar da multiplicação e aplicado a todos os pares de elementos dentro da extensão '[first1, last1)' e o mesmo número de elementos começando em 'last2'. O resultado final é retornado.

- **Exemplo:**

```
#include <numeric>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

class Cat
{
    std::string d_sep;
public:
    Cat(std::string const &sep)
    :
        d_sep(sep)
    {}
    std::string operator()
        (std::string const &s1, std::string const &s2) const
    {
        return s1 + d_sep + s2;
    }
};
```

```

using namespace std;

int main()
{
    size_t ial[] = {1, 2, 3, 4, 5, 6, 7};
    size_t ia2[] = {7, 6, 5, 4, 3, 2, 1};
    size_t init = 0;

    cout << "A soma de todos os quadrados em ";
    copy(ial, ial + 7, ostream_iterator<size_t>(cout, " "));
    cout << "é " <<
        inner_product(ial, ial + 7, ial, init) << endl;

    cout << "A soma de todos os produtos cruzados em ";
    copy(ial, ial + 7, ostream_iterator<size_t>(cout, " "));
    cout << " e ";
    copy(ia2, ia2 + 7, ostream_iterator<size_t>(cout, " "));
    cout << "é " <<
        inner_product(ial, ial + 7, ia2, init) << endl;

    string names1[] = {"Frank", "Karel", "Piet"};
    string names2[] = {"Brokken", "Kubat", "Plomp"};

    cout << "A lista de todos os nomes combinados em ";
    copy(names1, names1 + 3, ostream_iterator<string>(cout, " "));
    cout << "e\n";
    copy(names2, names2 + 3, ostream_iterator<string>(cout, " "));
    cout << "é:" <<
        inner_product(names1, names1 + 3, names2, string("\t"),
            Cat("\n\t"), Cat(" ")) <<
        endl;

    return 0;
}
/*
Generated output:

A soma de todos os quadrados em 1 2 3 4 5 6 7 é 140
A soma de todos os produtos cruzados em 1 2 3 4 5 6 7 e 7 6 5 4 3 2 1 é 84
A lista de todos os nomes combinados em Frank Karel Piet e
Brokken Kubat Plomp é:
    Frank Brokken
    Karel Kubat
    Piet Plomp
*/

```

17.4.22: inplace_merge()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
    - void inplace_merge(BidirectionalIterator first, BidirectionalIterator  
middle, BidirectionalIterator last);
```

```
    - void inplace_merge(BidirectionalIterator first, BidirectionalIterator  
middle, BidirectionalIterator last, Compare comp);
```

- Descrição:

- Primeiro protótipo: As duas extensões (ordenadas) '[first, midle)' e '[midle, last)' são unidas, mantendo a ordem (usando o operador 'operator<()' do tipo de dados para o qual o iterador aponta). A série final é guardada na extensão '[first, last)'.

• Segundo protótipo: Primeiro protótipo: As duas extensões (ordenadas) '[first, midle)' e '[midle, last)' são unidas, mantendo a ordem (Usando o resultado booleano do operador binário de comparação 'comp'). A série final é guardada na extensão '[first, last)'.

- Exemplo:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class CaseString
{
public:
    bool operator()(std::string const &first,
                    std::string const &second) const
    {
        return strcasecmp(first.c_str(), second.c_str()) < 0;
    }
};

using namespace std;

int main()
{
    string range[] =
```

```

    {
        "alpha", "charley", "echo", "golf",
        "bravo", "delta", "foxtrot",
    };

    inplace_merge(range, range + 4, range + 7);
    copy(range, range + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    string range2[] =
    {
        "ALFA", "CHARLEY", "DELTA", "foxtrot", "hotel",
        "bravo", "ECHO", "GOLF"
    };

    inplace_merge(range2, range2 + 5, range2 + 8, CaseString());
    copy(range2, range2 + 8, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Saída Gerada:

    alpha bravo charley delta echo foxtrot golf
    ALFA bravo CHARLEY DELTA ECHO foxtrot GOLF hotel
*/

```

17.4.23: iter_swap()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- void iter_swap(ForwardIterator1 iter1, ForwardIterator2 iter2);
```

- Descrição:

- Os elementos apontados por 'iter1' e 'iter2' são intercambiados.

- Exemplo:

```

#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

```

```

using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Antes:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (size_t idx = 0; idx < n; ++idx)
        iter_swap(first + idx, second + idx);

    cout << "Depois:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

Antes:
alpha bravo charley
echo foxtrot golf
Depois:
echo foxtrot golf
alpha bravo charley
*/

```

17.4.24: lexicographical_compare()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- bool lexicographical_compare(InputIterator1 first1, InputIterator1
                               last1, InputIterator2 first2, InputIterator2 last2);

- bool lexicographical_compare(InputIterator1 first1, InputIterator1
                               last1, InputIterator2 first2, InputIterator2 last2, Compare

```

comp);

- Descrição:

- Primeiro protótipo: O par correspondente de elementos dentro da extensão apontada por '[first1, last1)' e '[first2, last2)' são comparados. A função retorna verdadeira se:

- O primeiro elemento da primeira extensão é menor que o elemento correspondente da segunda extensão (usando 'operator<()' do tipo de dados próprio);
- 'last1' está acessível, mas 'last2' não o está ainda.
- É retornado falso nos outros casos que indicam que a primeira sequência não é lexicograficamente menor que a segunda. Assim, retorna falso se:
- O primeiro elemento da primeira sequência é maior que o primeiro elemento da segunda sequência (usando o 'operator<()' para o tipo de dados das sequências;
- 'last2' é acessado antes de 'last1';
- 'last1' e 'last2' são acessados.

- Segundo protótipo: Com esta função a operação de comparação binária é feita por 'comp' no lugar de 'operator<()'.

- Exemplo:

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

class CaseString
{
public:
    bool operator()(std::string const &first,
                    std::string const &second) const
    {
        return strcasecmp(first.c_str(), second.c_str()) < 0;
    }
};
```



```

using namespace std;

int main()
{
    string word1 = "hello";
    string word2 = "help";

    cout << word1 << " está " <<
        (
            lexicographical_compare(word1.begin(), word1.end(),
                                    word2.begin(), word2.end()) ?
                "antes de "
            :
                "além ou junto de "
        ) <<
        word2 << " no alfabeto\n";

    cout << word1 << " está " <<
        (
            lexicographical_compare(word1.begin(), word1.end(),
                                    word1.begin(), word1.end()) ?
                "antes de "
            :
                "além ou junto de "
        ) <<
        word1 << " no alfabeto\n";

    cout << word2 << " está " <<
        (
            lexicographical_compare(word2.begin(), word2.end(),
                                    word1.begin(), word1.end()) ?
                "antes de "
            :
                "além ou junto de "
        ) <<
        word1 << " no alfabeto\n";

    string one[] = {"alpha", "bravo", "charley"};
    string two[] = {"ALPHA", "BRAVO", "DELTA"};

    copy(one, one + 3, ostream_iterator<string>(cout, " "));
    cout << " está ordenada " <<
        (
            lexicographical_compare(one, one + 3,
                                    two, two + 3, CaseString()) ?
                "antes de "
            :
                "além ou junto de "
        );
    copy(two, two + 3, ostream_iterator<string>(cout, " "));
}

```

```

cout << endl <<
    "usando comparação com insensibilidade de caso maiúsculo/minúsculo.\n";

return 0;
}
/*
Saída Gerada:

hello está antes de help no alfabeto
hello está além ou junto de hello no alfabeto
help está além ou junto de hello no alfabeto
alpha bravo charley  está ordenada antes de ALPHA BRAVO DELTA
usando comparação com insensibilidade de caso maiúsculo/minúsculo.
*/

```

17.4.25: lower_bound()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- ForwardIterator lower_bound(ForwardIterator first, ForwardIterator
last, const Type &value);

```

```

- ForwardIterator lower_bound(ForwardIterator first, ForwardIterator
last, const Type &value, Compare comp);

```

- Descrição:

- Primeiro protótipo: Os elementos ordenados indicados pelo iterador '[first, last)' são recorridos para se buscar o primeiro elemento que não seja menor que (i.e., maior ou igual a) 'value'. O iterador retornado marca o local na sequência onde 'value' pode ser inserido sem quebrar a ordem dos elementos. O 'operator<()' do tipo de dados que o iterador aponta é usado. Se não existe tal elemento é retornado o último.
- Segundo protótipo: Os elementos indicados pelo iterador '[first, last)' devem estar ordenados usando-se a função (objeto) 'comp'. Cada elemento na extensão é comparado a 'value' usando-se a função 'comp'. É retornado um iterador que aponta o primeiro elemento que o predicado binário 'comp' retorne falso. Se não existir tal elemento, o último é retornado.

- Exemplo:

```

#include <algorithm>
#include <iostream>

```

```

#include <iterator>
#include <functional>
using namespace std;

int main()
{
    int    ia[] = {10, 20, 30};

    cout << "Seqüência: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 pode ser inserido antes de " <<
        *lower_bound(ia, ia + 3, 15) << endl;
    cout << "35 pode ser inserida depois do " <<
        (lower_bound(ia, ia + 3, 35) == ia + 3 ?
         "último elemento" : "???)") << endl;

    iter_swap(ia, ia + 2);

    cout << "Seqüência: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 pode ser inserido antes de " <<
        *lower_bound(ia, ia + 3, 15, greater<int>()) << endl;
    cout << "35 pode ser inserido antes do " <<
        (lower_bound(ia, ia + 3, 35, greater<int>()) == ia ?
         "primeiro elemento " : "???)") << endl;

    return 0;
}
/*
Saída Gerada:

Seqüência: 10 20 30
15 pode ser inserido antes de 20
35 pode ser inserida depois do último elemento
Seqüência: 30 20 10
15 pode ser inserido antes de 10
35 pode ser inserido antes do primeiro elemento
*/

```

17.4.26: max()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- `Type const &max(Type const &one, Type const &two);`
- `Type const &max(Type const &one, Type const &two, Comparator comp);`

- **Descrição:**

- Primeiro protótipo: O maior dos dois elementos 'one' e 'two' é retornado, usando-se 'operator>()' do tipo de dados para o qual o iterador aponta.
- Segundo protótipo: 'one' é retornado se o predicado binário 'comp(one, two)' retorna verdadeiro, do contrário é retornado 'two'.

- **Exemplo:**

```
#include <algorithm>
#include <iostream>
#include <string>

class CaseString
{
public:
    bool operator()(std::string const &first,
                    std::string const &second) const
    {
        return (strcasecmp(second.c_str(), first.c_str()) > 0);
    }
};

using namespace std;

int main()
{
    cout << "Word '" << max(string("first"), string("second")) <<
          "' é lexicograficamente última\n";

    cout << "Word '" << max(string("first"), string("SECOND")) <<
          "' é lexicograficamente última\n";

    cout << "Word '" << max(string("first"), string("SECOND"),
                           CaseString()) << "' é lexicograficamente última\n";

    return 0;
}
/*
Saída Gerada:

Word 'second' é lexicograficamente última
Word 'first' é lexicograficamente última
Word 'SECOND' é lexicograficamente última
*/
```

17.4.27: max_element()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
    - ForwardIterator max_element(ForwardIterator first, ForwardIterator  
last);  
    - ForwardIterator max_element(ForwardIterator first, ForwardIterator  
last, Comparator comp);
```

- Descrição:

- Primeiro protótipo: É retornado um iterador que aponta para o maior elemento dentro de '[first, last)'. O 'operator<()' do tipo de dados apontado pelo iterador é usado.
- Segundo protótipo: No lugar de usar 'operator<()', é usado o predicado binário 'comp' para realizar a comparação entre os elementos. O elemento que for retornado mais freqüentemente como verdadeiro, em comparação com os outros, é retornado.

- Exemplo:

```
#include <algorithm>  
#include <iostream>  
  
class AbsValue  
{  
    public:  
        bool operator()(int first, int second) const  
        {  
            return abs(first) < abs(second);  
        }  
};  
  
using namespace std;  
  
int main()  
{  
    int    ia[] = {-4, 7, -2, 10, -12};  
  
    cout << "O max. valor int. é: " << *max_element(ia, ia + 5) << endl;  
    cout << "O max. valor int. absoluto é: " <<  
        *max_element(ia, ia + 5, AbsValue()) << endl;  
  
    return 0;  
}
```

```

}
/*
    Saída Gerada:

    O max. valor int. é: 10
    O max. valor int. absoluto é: -12
*/

```

17.4.28: merge()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result);

```

```

- OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);

```

- Descrição:

- Primeiro protótipo: As duas extensões (ordenadas) '[first1, last1)' e '[first2, last2)' são unidas, mantendo uma lista ordenada (usando 'operator<()' do tipo de dados apontado pelos iteradores). A série final é armazenada na extensão começando em 'result' e terminando justo antes de 'OutputIterator' retornado pela função.
- Segundo protótipo: As duas extensões (ordenadas) '[first1, last1)' e '[first2, last2)' são unidas, mantendo uma lista ordenada (usando o resultado booleano da comparação binária 'comp'). A série final é armazenada na extensão começando em 'result' e terminando justo antes de 'OutputIterator' retornado pela função.

- Exemplo:

```

#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class CaseString
{
public:
    bool operator()(std::string const &first,
                   std::string const &second) const
    {

```

```

        return (strcasecmp(first.c_str(), second.c_str()) < 0);
    }
};

using namespace std;

int main()
{
    string range1[] =
        {
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        };
    string range2[] =
        {
            "echo", "delta", "golf", "romeo"
        };
    string result[5 + 4];

    copy(result,
        merge(range1, range1 + 5, range2, range2 + 4, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string range3[] =
        {
            "ALPHA", "bravo", "foxtrot", "HOTEL", "ZULU"
        };
    string range4[] =
        {
            "delta", "ECHO", "GOLF", "romeo"
        };

    copy(result,
        merge(range3, range3 + 5, range4, range4 + 4, result,
            CaseString()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

alpha bravo echo delta foxtrot golf hotel romeo zulu
ALPHA bravo delta ECHO foxtrot GOLF HOTEL romeo ZULU
*/

```

17.4.29: min()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- Type const &min(Type const &one, Type const &two);
```

```
- Type const &min(Type const &one, Type const &two, Comparator comp);
```

- Descrição:

- Primeiro protótipo: O menor dos dois elementos 'one' e 'two' é retornado, usando 'operator<()' do tipo de dados apontado pelos iteradores.
- Segundo protótipo: É retornado 'one' se o predicado binário 'comp(one, two)' retornar falso, do contrário retorna 'two'.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
```

```
class CaseString
{
public:
    bool operator() (std::string const &first,
                    std::string const &second) const
    {
        return (strcasecmp(second.c_str(), first.c_str()) > 0);
    }
};
```

```
using namespace std;
```

```
int main()
{
    cout << "A palavra '" << min(string("first"), string("second")) <<
        "' é lexicograficamente primeira\n";

    cout << " A palavra '" << min(string("first"), string("SECOND")) <<
        "' é lexicograficamente primeira\n";

    cout << " A palavra '" << min(string("first"), string("SECOND"),
        CaseString()) << "' é lexicograficamente primeira\n";
}
```



```

        return 0;
    }
    /*
    Generated output:

    A palavra 'first' é lexicograficamente primeira
    A palavra 'SECOND' é lexicograficamente primeira
    A palavra 'first' é lexicograficamente primeira
    */

```

17.4.30: min_element()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
```

```
- ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
Comparator comp);
```

- Descrição:

- Primeiro protótipo: É retornado um iterador que aponta para o menor elemento na extensão dentro de '[first, last)'. O 'operator<()' do tipo de dados apontado pelos iteradores.

- Segundo protótipo: no lugar do uso de 'operator<()', se usa o predicado binário 'comp' para comparar os elementos implicados. O elemento que retorne mais freqüentemente falso é retornado.

- Exemplo:

```

#include <algorithm>
#include <iostream>

class AbsValue
{
public:
    bool operator()(int first, int second) const
    {
        return abs(first) < abs(second);
    }
};

using namespace std;

```

```

int main()
{
    int    ia[] = {-4, 7, -2, 10, -12};

    cout << "O mínimo valor inteiro é: " << *min_element(ia, ia + 5) <<
        endl;
    cout << "O mínimo valor absoluto inteiro é: " <<
        *min_element(ia, ia + 5, AbsValue()) << endl;

    return 0;
}
/*
Generated output:

O mínimo valor inteiro é: -12
O mínimo valor absoluto inteiro é: -2
*/

```

17.4.31: mismatch()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
    InputIterator1 last1, InputIterator2 first2);

- pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
    InputIterator1 last1, InputIterator2 first2, Compare comp);

```

- Descrição:

- Primeiro protótipo: As duas seqüências de elementos começando por 'first1' e 'first2' são comparadas usando o operador de igualdade do tipo de dados apontado pelos iteradores. A comparação termina se os elementos comparados diferem (i.e., 'operator==()' retorna falso) ou 'last1' é acessado. É retornado um par de iteradores apontando para as posições finais. A segunda seqüência pode conter mais elementos que a primeira. O comportamento do algoritmo é indefinido se a segunda seqüência contém menos elementos que a primeira.
- Segundo protótipo: As duas seqüências de elementos começando por 'first1' e 'first2' são comparadas usando um operador binário de comparação 'comp'. A comparação termina se 'comp' retorna falso ou 'last1' é acessado. É retornado um par de iteradores

apontando para as posições finais. A segunda sequência pode conter mais elementos que a primeira. O comportamento do algoritmo é indefinido se a segunda sequência contém menos elementos que a primeira.

- Exemplo:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <utility>

class CaseString
{
public:
    bool operator()(std::string const &first,
                    std::string const &second) const
    {
        return strcasecmp(first.c_str(), second.c_str()) == 0;
    }
};

using namespace std;

int main()
{
    string range1[] =
    {
        "alpha", "bravo", "foxtrot", "hotel", "zulu"
    };
    string range2[] =
    {
        "alpha", "bravo", "foxtrot", "Hotel", "zulu"
    };
    pair<string *, string *> pss = mismatch(range1, range1 + 5, range2);

    cout << "Os elementos " << *pss.first << " e " << *pss.second <<
        " no deslocamento " << (pss.first - range1) << " diferem\n";
    if
    (
        mismatch(range1, range1 + 5, range2, CaseString()).first
        ==
        range1 + 5
    )
        cout << "Quando comparados com insensibilidade maiúscula/minúscula
concordam\n";

    return 0;
}
/*
```

```

Saída Gerada:

Os elementos hotel e Hotel no deslocamento 3 diferem
Quando comparados com insensibilidade maiúscula/minúscula concordam
*/

```

17.4.32: next_permutation()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

- bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Comp comp);

```

- Descrição:

- Primeiro protótipo: Dada a seqüência de elementos '[first, last)', a próxima permutação é determinada. Por exemplo, se os elementos 1, 2 e 3 é a série para a qual 'next_permutation()' é chamada, então chamadas subseqüentes a 'next_permutation()' reordena nas seguintes séries:

```

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

```

Este exemplo mostra que os elementos são reordenados de tal forma que a cada permutação representa um valor maior (132 é maior que 123, 213 é maior que 132, etc., usando 'operator<()' para o tipo de dados apontados pelos iteradores). É retornado verdadeiro se uma reordenação tem lugar, falso se a seqüência representa o maior valor possível. Neste caso a seqüência é ordenada com 'operator<()'.

- Segundo protótipo: Dada a seqüência de elementos '[first, last)', a próxima permutação é determinada. Os elementos da série são reordenados. Se a reordenação tem lugar é retornado verdadeiro, caso contrário é retornado falso.

Para a reordenação é usado o predicado 'comp' para comparar os elementos.

- Exemplo:

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

class CaseString
{
public:
    bool operator()(std::string const &first,
                    std::string const &second) const
    {
        return strcasecmp(first.c_str(), second.c_str()) < 0;
    }
};

using namespace std;

int main()
{
    string saints[] = {"Oh", "when", "the", "saints"};

    cout << "All permutations of 'Oh when the saints':\n";

    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
    while (next_permutation(saints, saints + 4, CaseString()));

    cout << "After first sorting the sequence:\n";

    sort(saints, saints + 4, CaseString());

    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
    while (next_permutation(saints, saints + 4, CaseString()));

    return 0;
}
```

```

/*
Saída Gerada (parcial):

All permutations of 'Oh when the saints':
Sequences:
Oh when the saints
saints Oh the when
saints Oh when the
saints the Oh when
...
After first sorting the sequence:
Sequences:
Oh saints the when
Oh saints when the
Oh the saints when
Oh the when saints
...
*/

```

17.4.33: nth_element()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- void nth_element(RandomAccessIterator first, RandomAccessIterator
nth, RandomAccessIterator last);

- void nth_element(RandomAccessIterator first, RandomAccessIterator
nth, RandomAccessIterator last, Compare comp);

```

- Descrição:

- Primeiro protótipo: Todos os elementos na extensão '[first, last)' são ordenados relativamente ao elemento apontado por 'nth': todos os elementos da série '[left, nth)' são menores que o elemento apontado por 'nth' e todos os elementos além, da série '[nth+1, last)', são maiores que o elemento apontado por 'nth'. Os dois subconjuntos não são ordenados. O 'operator<()' do tipo de dados que os iteradores apontam.

- Segundo protótipo: Como o primeiro, à exceção da função de comparação que será 'comp'.

- Exemplo:

```

#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    nth_element(ia, ia + 3, ia + 10);

    cout << "ordenados respeito a: " << ia[3] << endl;
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    nth_element(ia, ia + 5, ia + 10, greater<int>());

    cout << " ordenados respeito a: " << ia[5] << endl;
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

ordenados respeito a: 4
1 2 3 4 9 7 5 6 8 10
ordenados respeito a: 5
10 8 7 9 6 5 3 4 2 1
*/

```

17.4.34: partial_sort()

- - Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- void partial_sort(RandomAccessIterator first, RandomAccessIterator
middle, RandomAccessIterator last);
```

```
- void partial_sort(RandomAccessIterator first, RandomAccessIterator
middle, RandomAccessIterator last, Compare comp);
```

- Descrição:

- Primeiro protótipo: Os elementos da sequência '[midle, first)' são ordenados, usando-se 'operator<()' do tipo de dados apontados pelos iteradores' e armazenados na mesma extensão. Os elementos restantes da série não são ordenados e permanecem na extensão '[midle, last)'.
- Segundo protótipo: Os elementos da sequência '[midle, first)' são ordenados, de acordo com o predicado binário 'comp' e armazenados na mesma extensão. Os elementos restantes da série não são ordenados e permanecem na extensão '[midle, last)'.
- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    partial_sort(ia, ia + 3, ia + 10);

    cout << "encontrar os 3 elementos menores:\n";
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "encontrar os 5 elementos maiores:\n";
    partial_sort(ia, ia + 5, ia + 10, greater<int>());
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

encontrar os 3 elementos menores:
1 2 3 7 9 5 4 6 8 10
encontrar os 5 elementos maiores:
10 9 8 7 6 1 2 3 4 5
*/
```

17.4.35: partial_sort_copy()

- Arquivo cabeçalho:


```
#include <algorithm>
```

- **Protótipos da função:**

- `void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last);`
 - `void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last, Compare comp);`

- **Descrição:**

- **Primeiro protótipo:** Os menores elementos dentre '[first, last)' são copiados para a série '[dest_first, dest_last)', usando 'operator<()' do tipo de dados apontado. Só o número de elementos da extensão menor são copiados para a segunda série.
 - **Segundo protótipo:** Os elementos dentro de '[first, last)' são ordenados pelo predicado binário 'comp'. Os elementos que o predicado retorna mais frequentemente verdadeiro são copiados para '[dest_first, dest_last)'. Só os menores são copiados para a segunda série.

Exemplo:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 10, 3, 8, 5, 6, 7, 4, 9, 2};
    int ia2[6];

    partial_sort_copy(ia, ia + 10, ia2, ia2 + 6);

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;
    cout << "Os 6 elementos menores: ";
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Os 4 elementos menores numa extensão maior:\n";
    partial_sort_copy(ia, ia + 4, ia2, ia2 + 6);
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Os 4 elementos maiores numa extensão maior:\n";
    partial_sort_copy(ia, ia + 4, ia2, ia2 + 6, greater<int>());
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
```

```

    cout << endl;

    return 0;
}
/*
Saída Gerada:

1 10 3 8 5 6 7 4 9 2
Os 6 elementos menores: 1 2 3 4 5 6
Os 4 elementos menores numa extensão maior:
1 3 8 10 5 6
Os 4 elementos maiores numa extensão maior:
10 8 3 1 5 6
*/

```

17.4.36: partial_sum()

- Arquivo cabeçalho:

```
#include <numeric>
```

- Protótipos da função:

```

- OutputIterator partial_sum(InputIterator first, InputIterator
                             last, OutputIterator result);

- OutputIterator partial_sum(InputIterator first, InputIterator
                             last, OutputIterator result, BinaryOperation

```

```
op);
```

- Descrição:

- Primeiro protótipo: O valor de cada elemento na extensão '[result, <returned OutputIterator>)' é obtido adicionando-se os elementos correspondentes da extensão '[first, last)'. O primeiro elemento na extensão resultante será igual ao elemento apontado por 'first'.
- Segundo protótipo: O valor de cada elemento na extensão '[result, <returned OutputIterator>)' é obtido aplicando-se o operador binário 'op' ao elemento anterior na extensão resultante e o elemento correspondente dentro de '[first, last)'. O primeiro elemento na extensão resultante será igual ao elemento apontado por 'first'.

- Exemplo:

```

#include <numeric>
#include <algorithm>

```

```

#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 5};
    int ia2[5];

    copy(ia2,
        partial_sum(ia, ia + 5, ia2),
        ostream_iterator<int>(cout, " "));
    cout << endl;

    copy(ia2,
        partial_sum(ia, ia + 5, ia2, multiplies<int>()),
        ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Saída Gerada:

    1 3 6 10 15
    1 2 6 24 120
*/

```

17.4.37: partition()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```

- BidirectionalIterator partition(BidirectionalIterator first,
    BidirectionalIterator last, UnaryPredicate pred);

```

- Descrição:

- Todos os elementos na extensão '[first, last)' para os quais o predicado unário 'pred' avalie como verdadeiro são postos antes dos elementos avaliados como falso. O valor de retorno aponta justo além do último elemento da extensão particionada para o qual 'pred' avaliou como verdadeiro.

- Exemplo:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class LessThan
{
    int d_x;
public:
    LessThan(int x)
    :
        d_x(x)
    {}
    bool operator()(int value)
    {
        return value <= d_x;
    }
};

using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};
    int *split;

    split = partition(ia, ia + 10, LessThan(ia[9]));
    cout << "Último elemento <= 4 é ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

Último elemento <= 4 é ia[3]
1 3 4 2 9 10 7 8 6 5
*/

```

17.4.38: prev_permutation()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- bool prev_permutation(BidirectionalIterator first,
```

```
BidirectionalIterator last);
```

```
- bool prev_permutation(BidirectionalIterator first,  
    BidirectionalIterator last, Comp comp);
```

- **Descrição:**

- **Primeiro protótipo:** A permutação anterior que dá a sequência de elementos na extensão '[first, last)' é determinada. Os elementos dentro da extensão são reordenados de tal forma que a primeira ordenação é obtida, representando um valor `menor` (veja `next_permutation()`, seção 17.4.32, para um exemplo que envolve ordenações opostas). O valor verdadeiro é retornado se a reordenação teve lugar, o valor falso se não houve reordenação, que é o caso se a sequência fornecida estivesse já ordenada de acordo com '`operator<()`' para o tipo de dados que o iterador aponta.
- **Segundo protótipo:** A permutação anterior que dá a sequência de elementos na extensão '[first, last)' é determinada. Os elementos dentro da extensão são reordenados. O valor verdadeiro é retornado se a reordenação teve lugar, o valor falso se não houve reordenação, que é o caso se a sequência fornecida estivesse já ordenada usando o predicado binário '`comp`' para comparar os elementos.

- **Exemplo:**

```
#include <algorithm>  
#include <iostream>  
#include <string>  
#include <iterator>  
  
class CaseString  
{  
public:  
    bool operator()(std::string const &first,  
                    std::string const &second) const  
    {  
        return strcasecmp(first.c_str(), second.c_str()) < 0;  
    }  
};  
  
using namespace std;  
  
int main()  
{  
    string  saints[] = {"Oh", "when", "the", "saints"};  
  
    cout << "Todas as permutações anteriores a 'Oh when the saints':\n";  
  
    cout << "Sequências:\n";
```

```

do
{
    copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
    cout << endl;
}
while (prev_permutation(saints, saints + 4, CaseString()));

cout << "Depois da primeira ordenação a seqüência:\n";
sort(saints, saints + 4, CaseString());

cout << "Seqüências:\n";
while (prev_permutation(saints, saints + 4, CaseString()))
{
    copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
    cout << endl;
}
cout << "Não há (mais) permutações prévias \n";

return 0;
}
/*
Saída Gerada:

Todas as permutações anteriores a 'Oh when the saints':
Seqüências:
Oh when the saints
Oh when saints the
Oh the when saints
Oh the saints when
Oh saints when the
Oh saints the when
Depois da primeira ordenação a seqüência:
Seqüências:
Não há (mais) permutações prévias
*/

```

17.4.39: random_shuffle()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last);
```

```
- void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
```

```
RandomNumberGenerator rand);
```

- Descrição:

- Primeiro protótipo: Os elementos dentro da extensão '[first, last)' são reordenados aleatoriamente.
- Segundo protótipo: Os elementos dentro da extensão '[first, last)' são reordenados, usando o gerador de números aleatórios 'rand', que retorna um inteiro dentro da série '[0, resto)', onde resto é passado como argumento a 'operator()()' da função objeto 'rand'. Alternativamente, o gerador de números aleatórios pode ser uma função, que espera um resto inteiro e retorna um valor inteiro aleatório dentro do intervalo '[0, resto)'. Note que quando uma função objeto é usada, não pode ser um objeto anônimo. Por isso a função no exemplo usa o procedimento destacado em Press et al. (1992) “Numerical Recipes in C: The Art of Scientific Computing” (New York: Cambridge University Press, (2nd ed., p. 277)).

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <time.h>
#include <iterator>

int randomValue(int remaining)
{
    return static_cast<int>
        ( ((0.0 + remaining) * rand()) / (RAND_MAX + 1.0) );
}

class RandomGenerator
{
public:
    RandomGenerator()
    {
        srand(time(0));
    }
    int operator()(int remaining) const
    {
        return randomValue(remaining);
    }
};

void show(std::string *begin, std::string *end)
{
    std::copy(begin, end,
              std::ostream_iterator<std::string>(std::cout, " "));
}
```

```

        std::cout << std::endl << std::endl;
    }

using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa"};
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Usando Baralhamento Padrão:\n";
    random_shuffle(words, words + size);
    show(words, words + size);

    cout << "Usando Gerador Aleatório:\n";
    RandomGenerator rg;
    random_shuffle(words, words + size, rg);
    show(words, words + size);

    srand(time(0) << 1);
    cout << "Usando a função randomValue():\n";
    random_shuffle(words, words + size, randomValue);
    show(words, words + size);

    return 0;
}
/*
Saída Gerada (por exemplo):

Usando Baralhamento Padrão:
lima oscar mike november papa kilo

Usando Gerador Aleatório:
kilo lima papa oscar mike november

Usando a função randomValue():
mike papa november kilo oscar lima
*/

```

17.4.40: remove()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- ForwardIterator remove(ForwardIterator first, ForwardIterator
                           last, Type &value);
```


- Descrição:

- Os elementos dentro da extensão apontada por '[first, last)' são reordenados de tal forma que todos os valores diferentes de 'value' são colocados no início da sequência. O iterador retornado aponta para o primeiro elemento que pode ser removido depois da reordenação. A sequência '[return value, last)' é chamada esquerda do algoritmo. Note que a esquerda do algoritmo só pode conter valores diferentes de 'value' e estes podem ser removidos com segurança, dado que estão presentes na sequência '[first, return value)'. A função usa 'operator==()' do tipo de dados para o qual os iteradores apontam para determinar quais elementos remover.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    string *removed;
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Removendo todos os \"alpha\"s:\n";
    removed = remove(words, words + size, "alpha");
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Os elementos iniciais são:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

Removendo todos os \"alpha\"s::
kilo lima mike november oscar papa quebec
Os elementos iniciais são:
oscar alpha alpha papa quebec
*/
```

17.4.41: remove_copy()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- OutputIterator remove_copy(InputIterator first, InputIterator  
                             last, OutputIterator result, Type &value);
```

- Descrição:

- Os elementos na extensão apontada por '[first, last)' que não coincidem com 'value' são copiados para a sequência '[result, returnvalue)', onde 'returnvalue' é o valor retornado pela função. A sequência '[first, last)' não é alterada. A função usa 'operator==()' do tipo de dados para o qual os iteradores apontam para determinar quais elementos não são copiados.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining
    [
        size -
        count_if
        (
            words, words + size,
            bind2nd(equal_to<string>(), string("alpha"))
        )
    ];
    string *returnvalue =
        remove_copy(words, words + size, remaining, "alpha");

    cout << "Removendo todos os \"alpha\"s:\n";
```

```

    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

Removendo todos os \"alpha\"s:
kilo lima mike november oscar papa quebec
*/

```

17.4.42: remove_if()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- ForwardIterator remove_if(ForwardIterator first, ForwardIterator
                             last, UnaryPredicate pred);
```

- Descrição:

- Os elementos na extensão apontada por '[first, last)' são reordenados de tal maneira que todos os valores que o predicado 'pred' avalia como falsos são colocados no início da sequência. O iterador retornado aponta para o primeiro elemento, depois de reordenada, para o qual 'pred' retorna verdadeiro. A série '[returnvalue, last)' é dita esquerda do algoritmo. A esquerda do algoritmo só contém valores diferentes do valor os quais podem ser removidos com segurança, já que também estão contidos na sequência '[first, returnvalue)'.

- Exemplo:

```

#include <functional>
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()

```

```

{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Removendo todos os \"alpha\"s:\n";

    string *removed = remove_if(words, words + size,
                                bind2nd(equal_to<string>(), string("alpha")));

    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Os elementos iniciais são:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

Removendo todos os \"alpha\"s:
kilo lima mike november oscar papa quebec
Os elementos iniciais são:
oscar alpha alpha papa quebec
*/

```

17.4.43: remove_copy_if()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```

- OutputIterator remove_copy_if(InputIterator first, InputIterator
                                last, OutputIterator result, UnaryPredicate
pred);

```

- Descrição:

- Os elementos na sequência apontada por '[first, last)' para os quais o predicado unário 'pred' retorna verdadeiro são copiados para a extensão '[result, returnvalue)', onde 'returnvalue' é o valor retornado pela função. A sequência '[first, last)' não é modificada.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[
        size -
        count_if
        (
            words, words + size,
            bind2nd(equal_to<string>(), "alpha")
        )
    ];
    string *returnvalue =
        remove_copy_if
        (
            words, words + size, remaining,
            bind2nd(equal_to<string>(), "alpha")
        );

    cout << "Removendo todos os \"alpha\"s:\n";
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

Removendo todos os \"alpha\"s:
kilo lima mike november oscar papa quebec
*/
```

17.4.44: replace()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- ForwardIterator replace(ForwardIterator first, ForwardIterator
                           last, Type &oldvalue, Type &newvalue);
```

- Descrição:

- Todos os elementos iguais a 'oldvalue' na série apontada por '[first, last)' são substituídos por uma cópia de 'newvalue'. O algoritmo usa 'operator==()' do tipo de dados apontados pelos iteradores.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    replace(words, words + size, string("alpha"), string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/
```

17.4.45: replace_copy()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```

        - OutputIterator replace_copy(InputIterator first, InputIterator
                                         last, OutputIterator result, Type &oldvalue,
Type &newvalue);

```

- Descrição:

- Todos os elementos iguais a 'oldvalue' na série apontada por '[first, last)', são substituídos por uma cópia de 'newvalue' numa nova série '[result, returnvalue)', onde 'returnvalue' é o valor retornado da função. O algoritmo usa 'operator==()' do tipo de dados apontados pelos iteradores.

- Exemplo:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[size];

    copy
    (
        remaining,
        replace_copy(words, words + size, remaining, string("alpha"),
                                                             string("ALPHA")),
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*

```

Saída Gerada:

```
kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
```

*/

17.4.46: replace_if()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- ForwardIterator replace_if(ForwardIterator first, ForwardIterator  
                             last, UnaryPredicate pred, Type const &value);
```

- Descrição:

- Os elementos na série apontada por '[first, last)' para os quais o predicado unário 'pred' evaluate como verdadeiros são substituídos por 'newvalue'.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    replace_if(words, words + size,
               bind1st(equal_to<string>(), string("alpha")),
               string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
Saída Gerada:

kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
```


*/

17.4.47: replace_if()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
– OutputIterator replace_copy_if(ForwardIterator first, ForwardIterator  
last, OutputIterator result, UnaryPredicate pred, Type const &value);
```

- Descrição:

- Os elementos na série apontada por '[first, last)' são copiados para a série '[result, returnvalue)', onde 'returnvalue' é o valor retornado pela função. Os elementos para os quais o predicado unário 'pred' retorna verdadeiro são substituídos por 'newvalue'. A série '[first, last)' não é modificada.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string result[size];

    replace_copy_if(words, words + size, result,
                    bind1st(greater<string>(), string("mike")),
                        string("ALPHA"));
    copy (result, result + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
```

```

/*
  Saída Gerada (todos em uma linha):

  ALPHA ALPHA ALPHA mike ALPHA november ALPHA oscar ALPHA ALPHA
                                                                papa quebec
*/

```

17.4.48: reverse()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

- Descrição:

- Os elementos na série apontada por '[first, last)' são revertidos

- Exemplo:

```

#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;

    while (getline(cin, line))
    {
        reverse(line.begin(), line.end());
        cout << line << endl;
    }

    return 0;
}

```

17.4.49: reverse_copy()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- OutputIterator reverse_copy(BidirectionalIterator first,
                             BidirectionalIterator last,   OutputIterator
result);
```

- Descrição:

- Os elementos na série apontada por '[first, last)' são copiados para a série '[result, returnvalue)' em ordem inversa. O valor 'returnvalue' é o valor retornado pela função.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;

    while (getline(cin, line))
    {
        size_t    size = line.size();
        char      copy[size + 1];

        cout << "line: " << line << endl <<
              "reversed: ";
        reverse_copy(line.begin(), line.end(), copy);
        copy[size] = 0;      // 0 não é parte da linha revertida

        cout << copy << endl;
    }
    return 0;
}
```

17.4.50: rotate()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```
- void rotate(ForwardIterator first, ForwardIterator middle,  
             ForwardIterator last);
```

- Descrição:

- Os elementos dentro da série '[first, midle)' são movidos para o fim do recipiente, os elementos da série '[midle, last)' são movidos para o início do recipiente, mantendo a ordem dos elementos nos dois subconjuntos intacta.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    size_t const size = sizeof(words) / sizeof(string);
    size_t const midsize = 6;

    rotate(words, words + midsize, words + size);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/
```

17.4.51: rotate_copy()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```

        - OutputIterator rotate_copy(ForwardIterator first, ForwardIterator
                                   middle, ForwardIterator last, OutputIterator
result);

```

- Descrição:

- Os elementos dentro da série '[middle, last)' e os elementos dentro da série '[first, middle)' são copiados para o destino com extensão '[result, returnvalue)', onde 'returnvalue' é o iterador retornado pela função. A ordem original dos elementos nos subconjuntos não é alterada.

- Exemplo:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    size_t const size = sizeof(words) / sizeof(string);
    size_t midsize = 6;
    string out[size];

    copy(out,
         rotate_copy(words, words + midsize, words + size, out),
         ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/

```

17.4.52: search()

- Arquivo cabeçalho:

```

#include <algorithm>

```

- **Protótipos da função:**

```

- ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1
                           last1, ForwardIterator2 first2,
ForwardIterator2 last2);

- ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1
last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);

```

Descrição:

- **Primeiro protótipo:** É retornado um iterador dentro da primeira série '[first1, last1)', onde os elementos da segunda série, '[first2, last2)', são encontrados, usando 'operator==(())' do tipo de dados apontados pelos iteradores. Se não existe dito elemento é retornado 'last1'.
- **Segundo protótipo:** É retornado um iterador dentro da primeira série '[first1, last1)', onde os elementos da segunda série, '[first2, last2)', são encontrados, usando o predicado 'pred' para comparar os elementos nas duas séries. Se não existir dito elemento 'last1' é retornado.

- **Exemplo:**

```

#include <algorithm>
#include <iostream>
#include <iterator>

class absInt
{
public:
    bool operator()(int i1, int i2)
    {
        return abs(i1) == abs(i2);
    }
};

using namespace std;

int main()
{
    int range1[] = {-2, -4, -6, -8, 2, 4, 6, 8};
    int range2[] = {6, 8};

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2),

```

```

        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Saída Gerada:

    6 8
    -6 -8 2 4 6 8
*/

```

17.4.53: search_n()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- ForwardIterator1 search_n(ForwardIterator1 first1,
                             ForwardIterator1 last1, Size count, Type const &
value);

- ForwardIterator1 search_n(ForwardIterator1 first1, ForwardIterator1
last1, Size count, Type const & value, BinaryPredicate pred);

```

- Descrição:

- Primeiro protótipo: É retornado um iterador dentro da primeira série '[first1, last1)', onde n elementos com valor 'value' são encontrados usando 'operator==()' do tipo dos dados apontados pelos iteradores, para comparar os elementos. Se não existir dito elemento é retornado 'last1'.
- Segundo protótipo: É retornado um iterador dentro da primeira série '[first1, last1)',

onde n elementos com valor 'value' são encontrados usando o predicado binário 'pred' para comparar os elementos. Se dito elemento não existir será retornado 'last1'.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <iterator>

class absInt
{
    public:
        bool operator()(int i1, int i2)
        {
            return abs(i1) == abs(i2);
        }
};

using namespace std;

int main()
{
    int range1[] = {-2, -4, -4, -6, -8, 2, 4, 4, 6, 8};
    int range2[] = {6, 8};

    copy
    (
        search_n(range1, range1 + 8, 2, 4),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    copy
    (
        search_n(range1, range1 + 8, 2, 4, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
Saída Gerada:

4 4
-4 -4 -6 -8 2 4 4
*/
```


17.4.54: set_difference()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, InputIterator2 last2, OutputIterator result);
```

```
- OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,  
InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);
```

- Descrição:

- Primeiro protótipo: É retornada uma sequência ordenada de elementos apontados por '[first1, last1)' que não está presente na série '[first2, last2)', começando por 'result' e terminando em 'OutputIterator' retornado pela função. Os elementos nas duas séries devem ter sido ordenados usando-se 'operator<()' do tipo de dados que os iteradores apontam.
- Segundo protótipo: É retornada uma sequência ordenada de elementos apontados por '[first1, last1)' que não está presente na série '[first2, last2)', começando por 'result' e terminando em 'OutputIterator' retornado pela função. Os elementos nas duas séries devem ter sido ordenados usando-se a função objeto 'comp'.

- Exemplo:

```
#include <algorithm>  
#include <iostream>  
#include <string>  
#include <iterator>  
  
class CaseLess  
{  
public:  
    bool operator()(std::string const &left, std::string const &right)  
    {  
        return strcasecmp(left.c_str(), right.c_str()) < 0;  
    }  
};
```

```

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                     "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo" };
    string result[7];
    string *returned;

    copy(result,
         set_difference(set1, set1 + 7, set2, set2 + 3, result),
         ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO" };

    copy(result,
         set_difference(set1, set1 + 7, set3, set3 + 3, result,
                        CaseLess()),
         ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

kilo lima mike november oscar
kilo lima mike november oscar
*/

```

17.4.55: set_intersection()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- OutputIterator set_intersection(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);

- OutputIterator set_intersection(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,

```

```
OutputIterator result,
Compare comp);
```

- Descrição:

- Primeiro protótipo: É retornada uma sequência ordenada de elementos apontados por '[first1, last1)' que também estão presentes em '[first2, last2)', começando em 'result' e acabando em 'OutputIterator', retornado pela função. Os elementos nas duas séries precisam estar ordenados por 'operator<()' do tipo que os iteradores apontam.
- Segundo protótipo: É retornada uma sequência ordenada de elementos apontados por '[first1, last1)' que também estão presentes em '[first2, last2)', começando em 'result' e acabando em 'OutputIterator', retornado pela função. Os elementos nas duas séries precisam estar ordenados pela função objeto 'comp'.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
public:
    bool operator()(std::string const &left, std::string const &right)
    {
        return strcasecmp(left.c_str(), right.c_str()) < 0;
    }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                     "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo" };
    string result[7];
    string *returned;

    copy(result,
        set_intersection(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;
```

```

string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

copy(result,
      set_intersection(set1, set1 + 7, set3, set3 + 3, result,
                       CaseLess()),
      ostream_iterator<string>(cout, " "));
cout << endl;

return 0;
}
/*
Saída Gerada:

papa quebec
papa quebec
*/

```

17.4.56: set_symmetric_difference()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);

- OutputIterator set_symmetric_difference(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result,
    Compare comp);

```

- Descrição:

- Primeiro protótipo: É retornada uma seqüência ordenada de elementos apontados por '[first1, last1)' que não estejam presentes na série '[first2, last2)' e aqueles da série '[first2, last2)' não presentes na série '[first1, last1)', começando por 'result' e terminando em 'OutputIterator' retornados pela função. Os elementos nas duas séries devem ter sido ordenados por 'operator<()' do tipo de dados apontados pelos iteradores.
- Segundo protótipo: É retornada uma seqüência ordenada de elementos apontados por '[first1, last1)' que não estejam presentes na série '[first2, last2)' e aqueles da série '[first2,

last2)' não presentes na série '[first1, last1)', começando por 'result' e terminando em 'OutputIterator' retornados pela função. Os elementos nas duas séries devem ter sido ordenados usando a função objeto 'comp'.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
public:
    bool operator()(std::string const &left, std::string const &right)
    {
        return strcasecmp(left.c_str(), right.c_str()) < 0;
    }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                     "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo" };
    string result[7];
    string *returned;

    copy(result,
        set_symmetric_difference(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO" };

    copy(result,
        set_symmetric_difference(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

kilo lima mike november oscar romeo
```

```
kilo lima mike november oscar ROMEO
*/
```

17.4.57: set_union()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- OutputIterator set_union(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);

- OutputIterator set_union(
    InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2,
    OutputIterator result,
    Compare comp);
```

- Descrição:

- Primeiro protótipo: É retornada uma seqüência de elementos apontados por '[first1, last1)' que também estão presentes na série '[first2, last2)', começando em 'result' e terminando em 'OutputIterator' retornado pela função. Os elementos nas duas séries devem ter sido ordenados por 'operator<()' do tipo de dados apontados pelos iteradores.
- Segundo protótipo: É retornada uma seqüência de elementos apontados por '[first1, last1)' que também estão presentes na série '[first2, last2)', começando em 'result' e terminando em 'OutputIterator' retornado pela função. Os elementos nas duas séries devem ter sido ordenados pela função objeto 'comp'.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
public:
    bool operator()(std::string const &left, std::string const &right)
```

```

        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
    };

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                     "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo" };
    string result[7];
    string *returned;

    copy(result,
         set_union(set1, set1 + 7, set2, set2 + 3, result),
         ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO" };

    copy(result,
         set_union(set1, set1 + 7, set3, set3 + 3, result,
                  CaseLess()),
         ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Saída Gerada:

    kilo lima mike november oscar papa quebec romeo
    kilo lima mike november oscar papa quebec ROMEO
*/

```

17.4.58: sort()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- void sort(
    RandomAccessIterator first, RandomAccessIterator last);

- void sort(
    RandomAccessIterator first, RandomAccessIterator last,

```

```
Compare comp);
```

- Descrição:

- Primeiro protótipo: Os elementos da série '[first, last)' são ordenados em ordem ascendente usando 'operabr<()'' do tipo de dados apontados pelos iteradores.
- Segundo protótipo: Os elementos da série '[first, last)' são ordenados em ordem ascendente usando a função objeto 'comp' para comparar os elementos. O predicado binário 'comp' retorna verdadeiro se seu primeiro argumento estiver localizado antes, na sequência ordenada, que seu segundo argumento.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] = {"november", "kilo", "mike", "lima",
                     "oscar", "quebec", "papa"};

    sort(words, words + 7);
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    sort(words, words + 7, greater<string>());
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

kilo lima mike november oscar papa quebec
quebec papa oscar november mike lima kilo
*/
```

17.4.59: stable_partition()

- Arquivo cabeçalho:


```
#include <algorithm>
```

- **Protótipo da função:**

```
- BidirectionalIterator  
   stable_partition(BidirectionalIterator first,  
                   BidirectionalIterator last, UnaryPredicate pred);
```

- **Descrição:**

- Todos os elementos na série '[first, last)' para os quais o predicado unário avalie como verdadeiro são colocados antes dos elementos avaliados como falso. A ordem relativa dos elementos iguais é mantida. O valor de retorno aponta justo além do último elemento na série particionada para os quais 'pred' avalie como verdadeiros.

- **Exemplo:**

```
#include <algorithm>  
#include <iostream>  
#include <string>  
#include <functional>  
#include <iterator>  
using namespace std;  
  
int main()  
{  
    int org[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};  
    int ia[10];  
    int *split;  
  
    copy(org, org + 10, ia);  
    split = partition(ia, ia + 10, bind2nd(less_equal<int>(), ia[9]));  
    cout << "Último elemento <= 4 is ia[" << split - ia - 1 << "]\n";  
  
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));  
    cout << endl;  
  
    copy(org, org + 10, ia);  
    split = stable_partition(ia, ia + 10,  
                            bind2nd(less_equal<int>(), ia[9]));  
    cout << "Último elemento <= 4 is ia[" << split - ia - 1 << "]\n";  
  
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));  
    cout << endl;  
  
    return 0;  
}  
/*
```

```

Saída Gerada:

Último elemento <= 4 is ia[3]
1 3 4 2 9 10 7 8 6 5
Último elemento <= 4 is ia[3]
1 3 2 4 5 7 9 10 8 6
*/

```

17.4.60: `stable_sort()`

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- void stable_sort(
    RandomAccessIterator first, RandomAccessIterator last);
- void stable_sort(
    RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

```

- Descrição:

- Primeiro protótipo: Os elementos na série '[first, last)' são ordenados em ordem ascentente, usando 'operator<()' do tipo de dados apontados pelos iteradores: A ordem relativa de elementos iguais é matida.
- Segundo protótipo: Os elementos da série '[first, last)' são ordenados em ordem ascendente, usando o predicado binário 'pred' para comparar os elementos. Este predicado pode retornar verdadeiro se seu primeiro argumento for colocado antes do segundo no conjunto de elementos em ordenação.

- Exemplo (anotado abaixo):

```

#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <vector>
#include <iterator>

typedef std::pair<std::string, std::string> pss;    // 1 (see the text)

namespace std
{

```

```

        ostream &operator<<(ostream &out, pss const &p) // 2
        {
            return out << "      " << p.first << " " << p.second << endl;
        }
    }

class sortby
{
    std::string pss::*d_field;
public:
    sortby(std::string pss::*field) // 3
    :
        d_field(field)
    {}

    bool operator()(pss const &p1, pss const &p2) const // 4
    {
        return p1.*d_field < p2.*d_field;
    }
};

using namespace std;

int main()
{
    vector<pss> namecity; // 5

    namecity.push_back(pss("Hampson", "Godalming"));
    namecity.push_back(pss("Moran", "Eugene"));
    namecity.push_back(pss("Goldberg", "Eugene"));
    namecity.push_back(pss("Moran", "Godalming"));
    namecity.push_back(pss("Goldberg", "Chicago"));
    namecity.push_back(pss("Hampson", "Eugene"));

    sort(namecity.begin(), namecity.end(), sortby(&pss::first)); // 6

    cout << "ordenado pelos nomes:\n";
    copy(namecity.begin(), namecity.end(), ostream_iterator<pss>(cout));

    // 7
    stable_sort(namecity.begin(), namecity.end(), sortby(&pss::second));

    cout << "ordenado pelos nomes dentro da ordem por cidades:\n";
    copy(namecity.begin(), namecity.end(), ostream_iterator<pss>(cout));

    return 0;
}
/*
Saída Gerada:

ordenado pelos nomes:

```

```

Goldberg Eugene
Goldberg Chicago
Hampson Godalming
Hampson Eugene
Moran Eugene
Moran Godalming
ordenado pelos nomes dentro da ordem por cidades:
Goldberg Chicago
Goldberg Eugene
Hampson Eugene
Moran Eugene
Hampson Godalming
Moran Godalming
*/

```

Note que o exemplo implanta uma solução para um problema freqüente: Como ordenar usando múltiplos critérios hierárquicos. O exemplo merece uma atenção especial:

1. Primeiro um 'typedef' é usado para reduzir o grupo onde ocorre o uso repetido de 'pair<string, string>'.
2. Em seguida 'operator<<()' é sobrecarregado para ser capaz de inserir um par num objeto 'ostream'. Esta é somente uma função de serviço para facilitar. Se este espaço nomeado de envoltório é omitido, não será usado, já que o 'operator<<()' da 'ostream' tem que ser parte de 'std namespace'.
3. Então a classe 'sortby' é definida, permitindo a construção de um objeto anônimo que recebe um ponteiro a um dos pares de dados membros que são usados para ordenar. Neste caso, como ambos membros são objetos 'string', o construtor pode facilmente ser definido: seu parâmetro é um apontador a uma 'string' membro da classe 'pair<string, string>'.
4. O membro 'operator()()' receberá dois pares de referência e usará o apontador a seus membros, guardados no objeto 'sortby', para comparar os campos apropriados dos pares.
5. Em main(), primeiro alguns dados são guardados num vetor.
6. Então a primeira ordenação tem lugar. O critério menos importante deve ser ordenado primeiro e para isto um simples 'sort()' é suficiente. Como queremos que os nomes estejam ordenados dentro das cidades, os nomes representam o critério menos importante, assim ordenamos por nomes: 'sortby(&pss::first)'.
7. O critério seguinte em importância, das cidades, é aplicado. Como a ordem relativa dos nomes não será alterada por 'stable_sort()', as ligações observadas quando as cidades são ordenadas é resolvido de tal maneira que a ordem relativa existente não será quebrada. Assim, terminamos com

`Goldberg' em `Eugene' antes que `Hampson' em `Eugene', antes que `Moran' em `Eugene'. Para ordenar pelas cidades, usamos outro objeto 'sortby' anônimo: 'sortby(&pss::second)'.

17.4.61: swap()

- Arquivo cabeçalho:

```
#include <algorithm>
```

Protótipo da função:

```
- void swap(Type &object1, Type &object2);
```

- Descrição:

- Os elementos 'object1' e 'object2' trocam seus valores.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Antes:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (size_t idx = 0; idx < n; ++idx)
        swap(first[idx], second[idx]);

    cout << "Depois:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;
```

```

    return 0;
}
/*
Saída Gerada:

Antes:
alpha bravo charley
echo foxtrot golf
Depois:
echo foxtrot golf
alpha bravo charley
*/

```

17.4.62: swap_ranges()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipo da função:

```

- ForwardIterator2 swap_ranges(ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 result);

```

- Descrição:

- Os elementos na série apontados por '[first1, last1)' são trocados com os elementos da série '[result, returnvalue)', onde 'returnvalue' é o valor retornado pela função. As duas seqüências precisam ser separados.

- Exemplo:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};

```

```

size_t const n = sizeof(first) / sizeof(string);

cout << "Before:\n";
copy(first, first + n, ostream_iterator<string>(cout, " "));
cout << endl;
copy(second, second + n, ostream_iterator<string>(cout, " "));
cout << endl;

swap_ranges(first, first + n, second);

cout << "After:\n";
copy(first, first + n, ostream_iterator<string>(cout, " "));
cout << endl;
copy(second, second + n, ostream_iterator<string>(cout, " "));
cout << endl;

return 0;
}
/*
Saída Gerada:

Before:
alpha bravo charley
echo foxtrot golf
After:
echo foxtrot golf
alpha bravo charley
*/

```

17.4.63: transform()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- OutputIterator transform(InputIterator first, InputIterator last,
                           OutputIterator result, UnaryOperator op);
```

```
- OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, OutputIterator result,
                           BinaryOperator op);
```

- Descrição:

- Primeiro protótipo: O operador unário 'op' é aplicado a cada lelemento da série '[first,

last)' e o resultado é guardado na série começando por 'result'. O valor de retorno aponta justo além do último elemento gerado.

- Segundo protótipo: O operador binário 'op' é aplicado a cada elemento da série '[first1, last1)' e o elemento correspondente da série que começa em 'first2'. O resultado é guardado na série a partir de 'result'. O valor de retorno aponta justo além do último elemento gerado.

- Exemplo:

```
#include <functional>
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
#include <cctype>
#include <iterator>

class Caps
{
public:
    std::string operator()(std::string const &src)
    {
        std::string tmp = src;

        transform(tmp.begin(), tmp.end(), tmp.begin(), toupper);
        return tmp;
    }
};

using namespace std;

int main()
{
    string words[] = {"alpha", "bravo", "charley"};

    copy(words, transform(words, words + 3, words, Caps()),
          ostream_iterator<string>(cout, " "));
    cout << endl;

    int values[] = {1, 2, 3, 4, 5};
    vector<int> squares;

    transform(values, values + 5, values,
              back_inserter(squares), multiplies<int>());

    copy(squares.begin(), squares.end(),
          ostream_iterator<int>(cout, " "));
```



```

    cout << endl;

    return 0;
}
/*
Saída Gerada:

ALPHA BRAVO CHARLEY
1 4 9 16 25
*/

```

Comparado com o algoritmo genérico 'for_each()' (seção 17.4.17), as seguintes diferenças com o algoritmo genérico 'transform()' podem ser notadas:

- Com o 'transform()' o valor de retorno da função objeto membro 'operator()()' é usada; o argumento que é passado a 'operator()()' não muda.
- Com 'for_each()' a função objeto de 'operator()()' recebe uma referência a um argumento, que pode ser mudada pela função objeto de 'operator()()'.

17.4.64: unique()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- ForwardIterator unique(ForwardIterator first,
                          ForwardIterator last);

- ForwardIterator unique(ForwardIterator first,
                          ForwardIterator last, BinaryPredicate pred);

```

- Descrição:

- Primeiro protótipo: Elementos consecutivos iguais (usando 'operator==()') do tipo de dados apontados pelos iteradores) na série '[first, last)' são colapsados num só elemento. O iterador retornado marca o mais a esquerda do algoritmo e contém elementos únicos que apareciam anteriormente na série antes de 'unique()' ser chamada.
- Segundo protótipo: Elementos consecutivos iguais na série '[first, last)', para os quais o predicado binário 'pred' retorne verdadeiro, são colapsados num só elemento. O iterador retornado marca o mais a esquerda do algoritmo e contém elementos únicos que apareciam anteriormente na série antes de 'unique()' ser chamada.

- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseString
{
public:
    bool operator()(std::string const &first,
                    std::string const &second) const
    {
        return !strcasecmp(first.c_str(), second.c_str());
    }
};

using namespace std;

int main()
{
    string words[] = {"alpha", "alpha", "Alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    string *removed = unique(words, words + size);
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    removed = unique(words, words + size, CaseString());
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

alpha Alpha papa quebec
Trailing elements are:
quebec
alpha papa quebec
Trailing elements are:
quebec quebec
*/
```

17.4.65: unique_copy()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- OutputIterator unique_copy(InputIterator first, InputIterator last,  
                             OutputIterator result);  
- OutputIterator unique_copy(InputIterator first, InputIterator last,  
                             OutputIterator      Result,
```

```
BinaryPredicate pred);
```

- Descrição:

- Primeiro protótipo: Os elementos na série '[first, last)' são copiados ao recipiente resultante, começando em 'result'. Elementos iguais consecutivos (usando 'operator==()') do tipo de dados apontados pelos iteradores) são copiados uma vez. O iterador de saída retornado aponta justo além do último elemento copiado.
- Segundo protótipo: Os elementos na série '[first, last)' são copiados ao recipiente resultante, começando em 'result'. Elementos consecutivos na série para os quais o predicado binário retorne verdadeiro são copiados uma vez. O iterador de saída retornado aponta justo além do último elemento copiado.

- Exemplo:

```
#include <algorithm>  
#include <iostream>  
#include <string>  
#include <vector>  
#include <functional>  
#include <iterator>  
  
class CaseString  
{  
public:  
    bool operator()(std::string const &first,  
                   std::string const &second) const  
    {  
        return !strcasecmp(first.c_str(), second.c_str());  
    }  
};  
  
using namespace std;
```

```

int main()
{
    string words[] = {"oscar", "Alpha", "alpha", "alpha",
                     "papa", "quebec" };

    size_t const size = sizeof(words) / sizeof(string);
    vector<string> remaining;

    unique_copy(words, words + size, back_inserter(remaining));

    copy(remaining.begin(), remaining.end(),
         ostream_iterator<string>(cout, " "));
    cout << endl;

    vector<string> remaining2;

    unique_copy(words, words + size,
                back_inserter(remaining2), CaseString());

    copy(remaining2.begin(), remaining2.end(),
         ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
Saída Gerada:

oscar Alpha alpha papa quebec
oscar Alpha papa quebec
*/

```

17.4.66: upper_bound()

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

- ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value);
- ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);

- Descrição:

- Primeiro protótipo: Nos elementos ordenados da série '[first, last)' é buscado o primeiro elemento maior que 'value'. O iterador retornado marca o local na sequência onde 'value' pode ser inserido sem quebrar a ordem dos elementos, usando 'operator<()' do tipo de dados apontados pelos iteradores. Se não existir tal elemento, 'last' é retornado.
- Segundo protótipo: Os elementos da série '[first, last)' devem estar ordenados pela função ou função objeto 'comp'. Cada elemento da série é comparado a 'value' usando a função 'comp'. Um iterador ao primeiro para o qual o predicado binário 'comp', aplicado aos elementos da série e 'value' retorne verdadeiro. Se não existir tal elemento é retornado o último.
- Exemplo:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int        ia[] = {10, 15, 15, 20, 30};
    size_t     n = sizeof(ia) / sizeof(int);

    cout << "Sequência: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 pode ser inserido antes de " <<
        *upper_bound(ia, ia + n, 15) << endl;
    cout << "35 pode ser inserido depois do " <<
        (upper_bound(ia, ia + n, 35) == ia + n ?
         "último elemento" : "???)") << endl;

    sort(ia, ia + n, greater<int>());

    cout << "Sequência: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 pode ser inserido antes de " <<
        *upper_bound(ia, ia + n, 15, greater<int>()) << endl;
    cout << "35 pode ser inserido antes do " <<
        (upper_bound(ia, ia + n, 35, greater<int>()) == ia ?
         "primeiro elemento " : "???)") << endl;

    return 0;
}
```

```

}
/*
Generated output:

Sequência: 10 15 15 20 30
15 pode ser inserido antes de 20
35 pode ser inserido depois do último elemento
Sequência: 30 20 15 15 10
15 pode ser inserido antes de 10
35 pode ser inserido antes do primeiro elemento
*/

```

17.4.67: Algoritmos das Pilhas

Uma pilha é uma forma de árvore binária representada por um conjunto. Na pilha padrão, a chave de um elemento não é menor que a chave de seu filho. Este tipo de pilha é dita pilha máxima. uma árvore onde as chaves são números poderia estar organizada como segue:

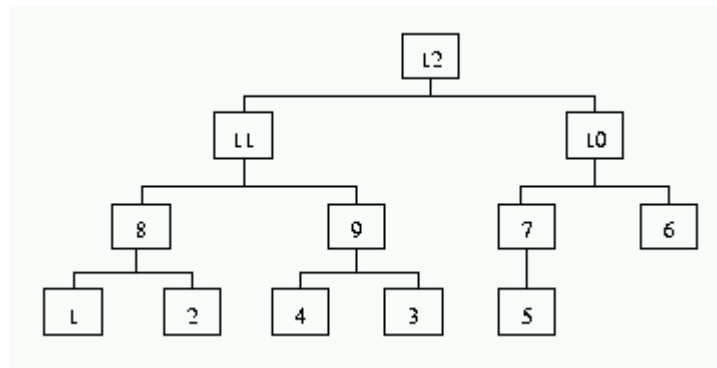


Figure 18 (Representação de uma árvore binária de uma pilha (heap))

Esta pilha pode estar organizada como:

12, 11, 10, 8, 9, 7, 6, 1, 2, 4, 3, 5

Na seguinte descrição seguinte, guarde duas coisas na mente: Um apontador para o nó indica a localização do nó seguinte da árvore, um apontador filho aponta o elemento seguinte que é um filho do apontador ao nó. Inicialmente o nó aponta para o primeiro elemento e o filho aponta para o segundo elemento.

- '*node++ (== 12)'. 12 é o nó superior. Seus filhos são '*child++ (11)' e '*child++ (10)', ambos menores que 12.
- O nó seguinte ('*node++ (== 11)'), por sua vez, tem '*child++ (8)' e '*child++ (9)' como filhos.
- O nó seguinte ('*node++ (== 10)') tem como filhos '*child++ (7)' e

```

'*child++ (6)'.
    - O nó seguinte ('*node++ (== 8)') tem como filhos '*child++ (1)' e
'*child++ (2)'.
    - Então ('*node++ (== 9)') tem como filhos '*child++ (4)' e '*child++
(3)'.
    - Finalmente (no que concerne aos filhos) ('*node++ (==7)') tem como filho
'*child++ (5)'.

```

Como 'child' agora aponta para além do conjunto, os nós restantes não possuem filhos. Assim, os nós 6, 1, 2, 4, 3 e 5 não têm filhos.

Note que os ramos esquerdo e direito não estão ordenados: 8 é menor que 9, mas 7 é maior que 6.

A pilha é criada percorrendo-se uma árvore binária em níveis, começando do nó do topo. O nó do topo é 12 no nível zero. No primeiro nível encontramos 11 e 10. no segundo nível 6, 7, 8 e 9, etc..

As pilhas podem ser criadas em recipientes que suportem acesso aleatório. Assim, uma pilha não é construída, por exemplo, numa lista. As pilhas podem ser construídas de um conjunto (`array`) (não ordenado) (usando `'make_heap()'`). O elemento do topo pode ser retirado da pilha, seguido de um reordenamento da pilha (usando-se `'pop_heap()'`), um novo elemento pode ser inserido à pilha, seguido de reordenamento da pilha (usando-se `'push_heap()'`) e os elementos de uma pilha podem ser reordenados (usando-se `'sort_heap()'`, que invalida a pilha).

As seguintes subsecções mostram os protótipos dos algoritmos de pilha, a subsecção final dá um pequeno exemplo onde são usados algoritmos de pilha.

17.4.67.1: A função `'make_heap()'`

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```

- void make_heap(RandomAccessIterator first,
                  RandomAccessIterator last);

- void make_heap(RandomAccessIterator first,
                  RandomAccessIterator last, Compare comp);

```

- Descrição:

- Primeiro protótipo: Os elementos na série '[first, last)' são reordenados para formar uma pilha máxima, usando 'operator<()' do tipo de dados para os que apontam os iteradores.
- Segundo protótipo: Os elementos na série '[first, last)' são reordenados para formar uma pilha, usando a função objeto de comparação 'comp' para comparar os elementos.

17.4.67.2: A função 'pop_heap()'

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- void pop_heap(RandomAccessIterator first,
               RandomAccessIterator last);

- void pop_heap(RandomAccessIterator first,
               RandomAccessIterator last, Compare comp);
```

- Descrição:

- Primeiro protótipo: O primeiro elemento na série '[first, last)' é movido para 'last-1'. Então os elementos na série '[first, last-1)' são reordenados para formar uma pilha máxima, usando 'operator<()' do tipo de dados para os quais os iteradores apontam.
- Segundo protótipo: O primeiro elemento na série '[first, last)' é movido para 'last-1'. Então os elementos na série '[first, last-1)' são reordenados para formar uma pilha máxima, usando a comparação binária 'comp'.

17.4.67.3: The 'push_heap()' function

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- void push_heap(RandomAccessIterator first,
```



```
RandomAccessIterator last);
```

```
- void push_heap(RandomAccessIterator first,  
RandomAccessIterator last, Compare comp);
```

- Descrição:

- Primeiro protótipo: Assumindo-se que a série '[first, last-2)' contém uma pilha válida e o elemento 'last-1' contém um elemento a ser inserido na pilha, os elementos da série '[first, last-1)' são reordenados para formar uma pilha máxima, usando 'operator<()' do tipo de dados que os iteradores apontam.
- Segundo protótipo: Assumindo-se que a série '[first, last-2)' contém uma pilha válida e o elemento 'last-1' contém um elemento a ser inserido na pilha, os elementos da série '[first, last-1)' são reordenados para formar uma pilha máxima, usando a comparação binária 'comp' para comparar os elementos.

17.4.67.4: A função 'sort_heap()'

- Arquivo cabeçalho:

```
#include <algorithm>
```

- Protótipos da função:

```
- void sort_heap(RandomAccessIterator first,  
RandomAccessIterator last);  
- void sort_heap(RandomAccessIterator first,  
RandomAccessIterator last, Compare comp);
```

- Descrição:

- Primeiro protótipo: Assumindo que os elementos na série '[first, last)' formam uma pilha máxima válida, os elementos na série são reordenados usando 'operator<()' do tipo de dados para os quais os iteradores apontam.
- Segundo protótipo: Assumindo que os elementos na série '[first, last)' formam uma pilha máxima válida, os elementos na série são reordenados usando a função objeto de comparação binária 'comp' para comparar os elementos.

17.4.67.5: Um exemplo usando as funções de pilha

Aqui está um exemplo mostrando vários algoritmos genéricos de manipulação de pilhas:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>

void show(int *ia, char const *header)
{
    std::cout << header << ":\n";
    std::copy(ia, ia + 20, std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
}

using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

    make_heap(ia, ia + 20);
    show(ia, "The values 1-20 in a max-heap");

    pop_heap(ia, ia + 20);
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20);
    show(ia, "Adding 20 (at the end) to the heap again");

    sort_heap(ia, ia + 20);
    show(ia, "Sorting the elements in the heap");

    make_heap(ia, ia + 20, greater<int>());
    show(ia, "The values 1-20 in a heap, using > (and beyond too)");

    pop_heap(ia, ia + 20, greater<int>());
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20, greater<int>());
    show(ia, "Adding 20 (at the end) to the heap again");

    sort_heap(ia, ia + 20, greater<int>());
    show(ia, "Sorting the elements in the heap");

    return 0;
}
```

```

}
/*
Saída Gerada:

The values 1-20 in a max-heap:
20 19 15 18 11 13 14 17 9 10 2 12 6 3 7 16 8 4 1 5
Removing the first element (now at the end):
19 18 15 17 11 13 14 16 9 10 2 12 6 3 7 5 8 4 1 20
Adding 20 (at the end) to the heap again:
20 19 15 17 18 13 14 16 9 11 2 12 6 3 7 5 8 4 1 10
Sorting the elements in the heap:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
The values 1-20 in a heap, using > (and beyond too):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Removing the first element (now at the end):
2 4 3 8 5 6 7 16 9 10 11 12 13 14 15 20 17 18 19 1
Adding 20 (at the end) to the heap again:
1 2 3 8 4 6 7 16 9 5 11 12 13 14 15 20 17 18 19 10
Sorting the elements in the heap:
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
*/

```

Capítulo 18: Modelos de Funções

A linguagem C++ suporta construções sintáticas que permitem aos programadores definir e usar funções ou classes completamente gerais (ou abstratas), baseadas em tipos genéricos e/ou (possivelmente deduzidas de) valores constantes. No capítulo sobre recipientes abstratos (Capítulo 12) e STL (Capítulo 17) já usamos essas construções, comumente conhecidos como mecanismos de modelos.

O mecanismo de modelo nos permite especificar classes e algoritmos, quase independentemente dos tipos em uso para os quais os modelos serão eventualmente usados. Sempre que o modelo é usado, o compilador gerará o código, na medida dos tipo(s) de dados usados com o modelo. Este código é gerado em tempo de compilação a partir da definição do modelo. O trecho de código gerado é chamado de instanciação do modelo.

Neste capítulo cobriremos as peculiaridades sintáticas dos modelos. As noções de tipo de parâmetro modelo, parâmetro modelo sem tipo, função modelo e classe modelo serão introduzidas e vários exemplos de modelos serão oferecidos, ambos neste capítulo e no Capítulo 20, fornecendo exemplos concretos em C++.

Os modelos padrão oferecidos pela linguagem já vistos nos recipientes nos permitem construir estruturas de dados altamente complexas de quase todos os padrões comumente usados na ciência da computação. Ainda mais, as classes 'string' (Capítulo 4) e 'stream' (Capítulo 5) são comumente implantadas usando modelos. Dessa forma os modelos ocupam um lugar central atualmente em C++ e não podem ser considerados uma característica esotérica da linguagem.

Os modelos podem ser considerados algo similares aos algoritmos genéricos: São um meio de vida: Um engenheiro em software com C++ deve buscar ativamente oportunidades de os usar. Inicialmente os modelos pareceram muito complexos e a maioria deu-lhe as costas. Contudo, com o tempo, seu poder e benefícios serão mais e mais apreciados. Eventualmente se reconhece oportunidades de uso de modelos. Esse é o momento de não se esforçar em construções concretas (i.e., sem modelo) de funções e classes, mas de construir modelos.

Este capítulo começa por introduzir modelos de funções. Esta parte do capítulo enfatiza na sintaxe requerida quando definimos tais funções. Na primeira parte está a fundamentação sobre a qual a segunda parte do capítulo esta baseada, introduzindo os modelos de classes e oferecendo muitos exemplos da vida real.

18.1: Definição de modelo de funções

A definição de modelo de função é muito similar à definição de função normal. Um modelo de função tem um cabeçalho, um corpo e um tipo de retorno, e possivelmente definições de sobrecarga, etc.. Contudo, à diferença das funções concretas, os modelos de funções sempre usam um ou mais tipos formais: Tipos para os quais quase todos (classe ou primitiva) os tipos podem ser usados. Começemos com um exemplo simples. A seguinte função 'add()' espera dois argumentos, e retorna sua soma:

```
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

Note como junto à definição da função acima segue sua descrição: Toma dois argumentos e retorna sua soma. Agora considere que aconteceria se tivéssemos que definir esta função], p.ex., para valores inteiros:

```
int add(int const &lvalue, int const &rvalue)
{
    return lvalue + rvalue;
}
```

Muito bem. Contudo, onde somamos valores duplos, teremos que sobrecarregá-la:

```
double add(double const &lvalue, double const &rvalue)
{
    return lvalue + rvalue;
}
```

Agora não há fim no número de versões sobrecarregadas que estamos obrigados a construir: Uma versão sobrecarregada para 'std::string', para size_t, para.... Em geral necessitaríamos uma versão sobrecarregada para cada tipo suportado por 'operator+()' e um construtor de cópia. Todas essas versões da mesma função básica são requeridas devido à natureza fortemente votada aos tipos de C++. Devido a isto, uma função realmente genérica não pode ser construída sem a utilização do mecanismo de modelagem.

Afortunadamente já vimos a parte consistente de um modelo de função. Nossa função inicial 'add()' é uma implantação de tal função. Contudo não é um modelo completo ainda. Se houvésssemos dado a primeira função 'add()' ao compilador, este produziria uma mensagem de erro como:

```
error: `Type' was not declared in this scope
error: parse error before `const'
```

E corretamente, como falhamos em definir 'Type'. Prevenimos o erro transformando 'add()' numa definição completa de modelo. Para tanto, olhamos a função e decidimos que 'Type' é um nome de tipo formal. Comparando-a com as implantações alternativas, estaria claro que poderíamos mudar 'Type'

por 'int' para termos a primeira implantação, por 'double' para a segunda.

A definição completa do modelo requer para este símbolo formal uma definição de tipo. Usando a palavra chave 'template', pomos como início de uma linha para nossa definição inicial, obtendo a seguinte definição de um modelo de função:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

Nesta definição distinguimos:

- A palavra chave 'template', iniciando a definição de um modelo ou declaração;
- Os parênteses angulares a lista do modelo. Uma lista com, talvez, mais que uma vírgula separando elementos. Esta lista entre parênteses angulares é chamada lista de parâmetros do modelo. Quando múltiplos elementos são usados, pareceria com:

```
typename Type1, typename Type2
```

- Dentro da lista de parâmetros do modelo encontramos o tipo formal 'Type'. É o nome de um tipo formal, comparável a um parâmetro formal numa definição de função. Até aqui encontramos só nomes formais de variáveis em funções. Os tipos dos parâmetros sempre são conhecidos no momento em que a função é definida. Os modelos ampliam a noção de nomes formais um degrau acima, permitindo ao nome do tipo ser formalizado, antes que só os nomes das variáveis dos parâmetros. O fato de que 'Type' é um nome de tipo formal está indicado pela palavra chave 'typename', na lista de parâmetros do modelo. Um nome de tipo formal, como 'Type' também é chamado de um parâmetro de tipo modelo. Existem também modelos além de tipo, que são introduzidos abaixo.

Outros textos C++ às vezes usam a palavra chave 'class' onde usamos e não 'class typename'. Assim, em outros textos de definição de modelos podem começar pela linha:

```
template <class Type>
```

Usar 'class' em lugar de 'typename' agora, contudo, é considerado um anacronismo e desaprovado: Um tipo de parâmetro de modelo é, apesar de tudo, um nome de tipo.

- O cabeçalho da função: É como o cabeçalho de uma função normal, apesar de que o tipo dos parâmetros do modelo deve ser usados em sua lista de parâmetros. Quando a função é chamada, usando argumentos com tipos concretos, estes tipos são então usados pelo compilador para determinar qual versão (sobrecarregada para atender os tipos de argumentos) do modelo da

função deve usar. Neste ponto (i.e., onde a função é chamada), o compilador criará a função concreta, um processo chamado instanciação. O cabeçalho da função pode usar um parâmetro formal também para especificar o valor de retorno. Esta característica é usada na definição do modelo de 'add()'.

- Os parâmetros da função são especificados como 'Type const &'. Isto tem o significado usual: Os parâmetros são referências a objetos do tipo 'Type' ou valores que não serão modificados pela função.
- O corpo da função: É como o corpo de uma função normal. No corpo se usam os nomes dos tipos formais para definir ou declarar as variáveis, que então podem ser usadas como qualquer outra variável. Mesmo assim existem algumas restrições. Observando o corpo de 'add()' fica claro que o 'operator+()' é usado, bem como um construtor de cópias, já que a função retorna um valor. Isto nos permite formular as seguintes restrições para o tipo formal 'Type':

- 'Type' deve suportar 'operator+()'
- 'Type' deve suportar um construtor de cópias

Conseqüentemente enquanto 'Type' pode ser um 'std::string', nunca deverá ser um 'ostream', como nem 'operator+()' nem um construtor de cópias estão disponíveis para as 'streams'.

As regras de escopo e visibilidade dos identificadores normais são aplicadas às definições de modelos. O escopo nas definições de modelos sobrepõem as regras de escopo para identificadores com nomes idênticos de amplo escopo.

Repassemos outra vez os parâmetros da função, como definidos na lista de parâmetros. Especificando 'Type const &' em lugar de 'Type' se previne cópias supérfluas, ao mesmo tempo se permite valores de tipos primitivos serem passados como argumentos da função. Assim, quando 'add(3, 4)' é chamada, 'int(4)' será adjudicado a 'Type const &rvalue'. Em geral, os parâmetros da função serão definidos como 'Type const &' para prevenir cópias desnecessárias. O compilador é suficientemente esperto para manipular 'referências a referências' neste caso, algo que a linguagem normalmente não suporta. Por exemplo, considere a seguinte função 'main()' (aqui e nos exemplos seguintes assumimos o modelo, os cabeçalhos requeridos e declarações de espaços nomeados foram fornecidos):

```
int main()
{
    size_t const &uc = size_t(4);
    cout << add(uc, uc) << endl;
}
```

Aqui 'uc' é uma referência a uma constante `size_t`. É passada como argumento a `'add()'`, portanto iniciando 'lvalue' e 'rvalue' como 'Type const & to size_t const & values', com o compilador interpretando 'Type' como `size_t`. Alternativamente os parâmetros foram especificados usando 'Type &', no lugar de 'Type const &'. A desvantagem desta especificação (não constante) é que valores temporários não podem ser mais passados à função. O seguinte exemplo falhará ao ser compilado:

```
int main()
{
    cout << add(string("a"), string("b")) << endl;
}
```

Aqui, uma 'string const &' não pode ser usada para iniciar uma 'string &'. Por outro lado, o seguinte será compilado, com o compilador decidindo que o 'Type' pode ser considerado uma 'string const':

```
int main()
{
    string const &s = string("a");
    cout << add(s, s) << endl;
}
```

Que podemos deduzir dos exemplos?

- Em geral, os parâmetros das funções devem ser especificados como 'Type const &' para prevenir cópias desnecessárias.
- O mecanismo de modelagem é razoavelmente flexível, assim interpreta tipos formais como tipos concretos, tipos constantes, tipos ponteiros, etc., dependendo dos tipos em uso. A regra prática é que os tipos formais são usados como máscaras genéricas do tipo em uso, com o nome do tipo formal encobrindo qualquer que seja a parte que necessite de disfarce. Alguns exemplos, assumindo que o parâmetro é definido como 'Type const &':

```
argument type    Type ==

size_t const    size_t

size_t    size_t

size_t *    size_t *

size_t const *    size_t const *
```

Como segundo exemplo de um modelo de função considere a seguinte definição de função:

```
template <typename Type, size_t Size>
Type sum(Type const (&array)[Size])
{
    Type t = Type();
```



```

    for (size_t idx = 0; idx < Size; idx++)
        t += array[idx];

    return t;
}

```

Esta definição de modelo introduz os seguintes conceitos novos e características:

- Sua lista de parâmetros de modelo tem dois elementos. O primeiro já é bem conhecido, mas o segundo elemento tem um tipo bem específico: Um 'size_t'. Parâmetros de modelos de tipo específico (i.e., não formal), tipos usados nas listas de parâmetros dos modelos são ditos parâmetros modelo sem tipo.

Um parâmetro modelo sem tipo representa uma expressão constante, que precisa ser conhecida no momento em que o modelo seja instanciado e que é especificada nos termos de tipos existentes, tal como um 'size_t'.

Observando o cabeçalho da função vemos um parâmetro:

```
Type const (&array)[Size]
```

Este parâmetro define um conjunto como um parâmetro de referência a um conjunto com 'Size' elementos do tipo 'Type', isto não pode ser modificado.

- Na definição de parâmetros, ambos, 'Type' e 'Size' são usados. 'Type' é, claro está, o tipo de parâmetro do modelo, mas 'Size' também é um parâmetro do modelo. É um 'size_t', cujo valor deve ser inferido pelo compilador na compilação de chamadas ao modelo da função 'sum()'. Consequentemente 'Size' deve ser um valor constante. Tais expressões constantes são chamadas parâmetros modelo sem tipo e são nomeadas na lista de parâmetros do modelo
- Quando a função modelo é chamada o compilador tem que estar capacitado a inferir não só os valores concretos de 'Type', mas também os valores de 'Size'. Como a função 'sum()' tem um só parâmetro, o compilador só está capacitado a inferir o valor de 'Size' dos argumentos da função. Ele só o pode fazer se o argumento fornecido for um conjunto (de tamanho fixo e conhecido), antes que um ponteiro aos elementos 'Type'. Assim, na seguinte função 'main()' o primeiro comando compila corretamente, já o segundo não:

```

int main()
{
    int values[5];
    int *ip = values;

    cout << sum(values) << endl;    // compila ok
}

```

```

        cout << sum(ip) << endl;           // não compila
    }

```

- Dentro da função, o comando 'Type t = Type()' é usado para iniciar 't' com um vlaor padrão. Note que aqui não é usado um valor fixo (como 0). Qualquer valor padrão pode ser obtido usando-se seu construtor padrão, antes que usando um valor numérico fixo. Claro que nem todas as classes aceitam um valor numérico como argumento a um de seus construtores. Mas todos os tipos, mesmo os tipos primitivos, suportam construtores padrão (atualmente algumas classes não implantam um construtor padrão, mas a maioria sim). O construtor padrão de tipos primitivos iniciam suas variáveis em 0 (ou falso). Ainda mais, o comando 'Type t = Type()' é uma iniciação real: 't' é iniciado pelo construtor padrão de 'Type', antes que usando o construtor de cópias de 'Type' para adjudicar uma cópia de 'Type' a 't'. Altyernativamente a construção sintática 'Type t (Type())' poderia ter sido usada.
- comparável à primeira função modelo, 'sum()' também assume a existência de certos membros públicos na classe 'Type'. Neste momento 'operator+=(())' e um construtor de cópias de 'Type'.

Como as definições de classes, as definições de modelos não contêm diretivas 'using' ou declarações: O modelo deve ser usado numa situação onde tal diretiva anula as intenções do programador: Ambigüidades ou outros conflitos podem resultar se o autor e o programador usem diferentes diretivas 'using' (P.ex., uma variável 'cout' definida no espaço nomeado 'std' e o programador ewm seu espaço nomeado). Em modelos deve-se usar somente nomes inteiramente qualificados, incluindo todas as especificações de espaços nomeados.

18.2: Dedução de Argumentos

Nesta seção nos concentraremos no processo usado pelo compilador para deduzir os tipos concretos dos tipos formais dos parâmetros dos modelos, quando um modelo de função é chamado, processo chamado dedução do argumento do modelo. Como já vimos, o compilador está capacitado a substituir um simples tipo formal de parâmetro por uma grande variedade de tipos. Mesmo assim, nem todas as conversões são possíveis. Em particular quando uma função tem múltiplos parâmetros do mesmo tipo de modelo de parâmetro, o compilador é muito restritivo no tipo de argumentos aceitará.

Quando o compilador deduz os tipos dos modelos de parâmetros, só considera os tipos dos argumentos. Nem as variáveis locais nem os valores de retorno são considerados neste processo. Isto é compreensível: Quando uma função é chamada, o compilador vê só os argumentos do modelo da função com certa segurança. No ponta de chamada não vê, definitivamente, os tipos das variáveis locais da função e o valor de retorno não é usado aí ou pode estar adjudicado ao tipo de uma variável de um

subconjunto (ou super-conjunto) de um tipo deduzido do parâmetro do modelo. Assim, no seguinte exemplo, o compilador nunca será capaz de deduzir o tipo do tipo de parâmetro do modelo 'Type':

```
template <typename Type>
Type fun()           // nunca pode ser chamada
{
    return Type();
}
```

Em geral, quando uma função tem múltiplos parâmetros com tipos formais idênticos, o tipo atual deve ser exatamente o mesmo. Assim:

```
void binarg(double x, double y);
```

pode ser chamada usando-se 'int' e 'double', com conversão implícita do correspondente argumento 'int' em 'double', mas a função modelo correspondente não pode ser chamada usando um argumento 'int' e um 'double': O compilador, por sua conta, não promoverá um 'int' a 'double' ou decidirá que o 'Type' seguinte pode ser um 'double':

```
template <typename Type>
void binarg(Type const &p1, Type const &p2)
{}

int main()
{
    binarg(4, 4.5); // ?? não compila: tipos diferentes
}
```

Então, quais são as conversões que o compilador aplicará ao deduzir os tipos concretos de tipos formais? Ele só realiza três tipos de conversões de tipos de parâmetros (e uma quarta para parâmetros de funções de qualquer tipo fixo (i.e., de um tipo de parâmetro de função não modelo)). Se o compilador não pode deduzir os tipos usando essas conversões o modelo da função não será considerado. Estas conversões são:

- Transformações em 'lvalue', criando 'rvalue' de um 'lvalue';
- Conversões qualificadas, inserindo um modificador constante a um tipo de argumento não constante;
- Conversão a uma classe de base instanciada de uma classe modelo, usando um modelo de classe de base quando um argumento de um tipo de modelo de classe derivada foi fornecido na chamada;
- Conversões padrão para tipos de parâmetros de funções não modelo. Estas não são conversões de um tipo de parâmetro modelo, mas se refere a qualquer tipo não modelo de parâmetro de modelos de funções. Para estes parâmetros de funções o compilador fará uma conversão

estandarte disponível (p.ex., 'int' a 'size_t', 'int' a 'double', etc.).

18.2.1: Transformações em 'lvalue'

Há três tipos de transformações em 'lvalue':

- Conversões de 'lvalue' a 'rvalue'.

Uma conversão de 'lvalue' a 'rvalue' é aplicada quando é requerido um 'rvalue' e um 'lvalue' é usado como argumento. Isto acontece quando uma variável é usada como argumento de uma função especificando o valor de um parâmetro. P.ex.:

```
template<typename Type>
Type negate(Type value)
{
    return -value;
}
int main()
{
    int x = 5;
    x = negate(x);    // lvalue (x) to rvalue (copies x)
}
```

- Conversões de 'array' a ponteiro

Uma conversão de conjunto a ponteiro é aplicada quando o nome de um conjunto é adjudicado a um ponteiro. Esta é comum em funções que definem parâmetros apontadores. Quando chamamos tais funções, freqüentemente se especifica como seus argumentos conjuntos. O endereço do conjunto é adjudicado ao parâmetro apontador e seu tipo é usado para deduzir o tipo do parâmetro do modelo. Por exemplo:

```
template<typename Type>
Type sum(Type *tp, size_t n)
{
    return accumulate(tp, tp + n, Type());
}
int main()
{
    int x[10];
    sum(x, 10);
}
```

Neste exemplo o local do conjunto 'x' é passado a 'sum()', que espera um apontador a algum tipo. Usando a conversão de conjunto a ponteiro, o endereço de 'x' é considerado um apontador que é

adjudicado a 'tp', deduzindo que o 'Type' no processo é inteiro.

- Conversões de função a ponteiro

Esta conversão é vista mais frequentemente em modelos de funções que definem um parâmetro que aponta para uma função. Quando chamamos tais funções o nome de uma função pode ser especificado como argumento. O endereço da função é então adjudicado ao parâmetro apontador, deduzindo, no processo, o tipo de parâmetro do modelo. Isto é chamado conversão de função a ponteiro. Por exemplo:

```
#include <cmath>

template<typename Type>
void call(Type (*fp)(Type), Type const &value)
{
    (*fp)(value);
}

int main()
{
    call(&sqrt, 2.0);
}
```

Neste exemplo o endereço da função 'sqrt()' é passado a 'call()', que espera um apontador a uma função que retorna um 'Type' e espera um 'Type' como argumento. Usando a conversão de função a ponteiro, o endereço de 'sqrt()' é considerado um ponteiro que é adjudicado a 'fp', no processo é deduzido que o tipo de 'Type' é um 'double'. Note que o argumento 2,0 não poderia ser especificado como 2, já que não há protótipo 'sqrt(int)'. Note também que o primeiro parâmetro da função especifica 'Type (*fp)(Type)' antes que 'Type (*fp)(Type const &)' como seria de se esperar das discussões anteriores sobre como especificar os tipos de parâmetros de modelos, preferindo referências a valores. Contudo, o argumento de 'Type', 'fp', não é um parâmetro de um modelo de função, mas um parâmetro da função para o qual aponta. Como o protótipo de 'sqrt()' é 'double sqrt(double)', antes que 'double sqrt(double const &)', o parâmetro de 'call()', 'fp', precisa ser especificado como 'Type (*fp)(Type)'. Assim de estrito.

18.2.2: Conversões de Qualificação

Uma conversão de qualificação agrega a qualificação a ponteiros 'const' ou 'volatile'. A conversão é aplicada quando o parâmetro do modelo da função for explicitamente definido usando o modificador constante (ou volátil) e o argumento da função não é uma entidade constante ou volátil. Nesse caso a conversão adiciona constante ou volátil e em seguida deduz o tipo do parâmetro do modelo. Por exemplo:

```
template<typename Type>
Type negate(Type const &value)
```

```

{
    return -value;
}
int main()
{
    int x = 5;
    x = negate(x);
}

```

Aqui vemos o parâmetro do modelo da função 'Type const &': Uma referência a um 'Type' constante. Contudo o argumento não é um 'const int', mas um inteiro que pode ser modificado. Aplicando-se uma conversão de qualificação, o compilador agrega constante ao tipo de 'x' e então combina com 'int const x' com 'Type const &value', deduzindo que 'Type' tem que ser inteiro.

18.2.3: Conversão a uma classe de base

Apesar de que modelos de classes só podem ser construídos mais tarde neste capítulo (na seção 19), modelos de classes têm sido já extensivamente usados antes. Por exemplo, os recipientes abstratos (no Capítulo 12) são definidos como modelos de classes. Como classes concretas (i.e., não modelos de classes), os modelos de classes podem participar na construção de hierarquias de classes.

Na seção 19.9 se mostra como um modelo de classe pode derivar de outro modelo de classe.

Como a derivação de modelo de classe fica por ser coberta, a seguinte discussão será necessariamente algo abstrata. Opcionalmente o leitor, claro está, pode saltar brevemente à seção 19.9 para consultar essa seção.

Nesta seção deve-se agora assumir, em benefício do argumento, que um modelo de classe 'Vector', de alguma forma derivou de 'std::vector'. Ainda mais, assumir que o seguinte modelo de classe foi construído para ordenar um vetor usando uma função objeto 'obj':

```

template <typename Type, typename Object>
void sortVector(std::vector<Type> vect, Object const &obj)
{
    sort(vect.begin(), vect.end(), obj);
}

```

Para ordenar objetos 'std::vector<string>' insensivelmente a maiúscula / minúscula a classe 'Caseless' pode ser como segue:

```

class CaseLess
{
public:
    bool operator() (std::string const &before,
                    std::string const &after) const

```

```

        {
            return strcasecmp(before.c_str(), after.c_str()) < 0;
        }
    };

```

Agora vários vetores podem ser ordenados usando 'sortVector()':

```

int main()
{
    std::vector<string> vs;
    std::vector<int> vi;

    sortVector(vs, CaseLess());
    sortVector(vi, less<int>());
}

```

Aplicando a conversão a uma classe de base, instanciada de um modelo de classe, o modelo de função 'sortVector()' pode agora também ser usada para ordenar um objeto 'Vector'. Por exemplo:

```

int main()
{
    Vector<string> vs;
    Vector<int> vi;

    sortVector(vs, CaseLess());
    sortVector(vi, less<int>());
}

```

Neste exemplo foi passado 'Vectors' como argumento a 'sortVector()'. Aplicando a conversão a uma classe de base instanciada de um modelo de classe, o compilador considerará 'Vector' como sendo um 'std::vector' e é capaz de deduzir o tipo de parâmetro do modelo. Um 'std::string' para o 'Vector' 'vs' e um inteiro para 'Vector' 'vi'.

Por favor, faça várias deduções de conversão de argumentos de modelos. Elas não têm como objetivo a concordância de argumentos de funções com parâmetros de funções, mas concordar argumentos com parâmetros, as conversões pode ser aplicadas para determinar os tipos de vários tipos de parâmetros de modelos.

18.2.4: O algoritmos de dedução de modelos de argumentos

O compilador usa o seguinte algoritmo para deduzir os tipos de seus modelos de tipos de parâmetros:

- Por turno são identificados os parâmetros do modelo da função usando os argumentos da função chamada.

- Para cada modelo de parâmetro usado na lista de modelos de parâmetros da função, o tipo de modelo do parâmetro é concordado com o tipo de argumento correspondente (p.ex., `Type` é `int` se o argumento for `int x` e o parâmetro da função for `Type &value`);
- Quando concordando os tipos de argumentos com os tipos de parâmetros, as três conversões (veja seção 18.2) para tipos de parâmetros são aplicadas onde necessário.
- Se modelos idênticos de tipos de parâmetros são usados em funções com múltiplos parâmetros, os tipos deduzidos dos modelos devem ser exatamente os mesmos. Assim, o seguinte modelo de função não pode ser chamada com um argumento `int` e um `double`:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

Ao chamar o modelo desta função, dois tipos idênticos devem ser usados (apesar de que as três conversões padrão são permitidas). Se o mecanismo de dedução de modelos não retorna tipos idênticos para tipos de modelos idênticos, então a função não será instanciada.

18.3: Declarando modelos de funções

Até aqui somente definimos modelos de funções. Há várias consequências de incluirmos definições de modelos de funções em múltiplos arquivos fonte, nenhuma delas grave, mas mal conhecidas.

- Como as interfaces de classe, as definições de modelos usualmente são incluídos nos arquivos cabeçalho. Cada vez que um arquivo cabeçalho que contém uma definição de modelo é lido pelo compilador, o compilador tem que processar completamente a definição, mesmo que não tiver necessidade no momento do modelo. Isto retarda relativamente a compilação. Por exemplo, compilar um arquivo cabeçalho como 'algorithm' em minha velha laptop toma cerca de quatro vezes mais tempo que compilar um cabeçalho pleno como 'cmath'. O arquivo cabeçalho 'iostream' é ainda mais árduo de processar, requer quase 15 vezes o tempo de 'cmath'. Fica claro que processar modelos é um sério trabalho para o compilador.
- Cada vez que um modelo de função é instanciada, seu código aparece no módulo objeto resultante. Contudo, se existir múltiplas instanciações de um modelo, usando o mesmo tipo para seus modelos de parâmetros, em múltiplos arquivos objetos, então o linkador descartará as instanciações supérfluas. No programa final só uma instanciação para um conjunto particular do

tipo de modelo dos parâmetros será usada (veja também a seção 18.4 para uma ilustração). Portanto, o linkador terá uma tarefa adicional a executar (dispensar múltiplas instanciações), o que retardará o processo de linkagem.

- Às vezes as próprias definições não são requeridas, mas só referências ou apontadores aos modelos. Nesses casos será desnecessário o retardo no processo de compilação.

Em lugar de incluir as definições dos modelos uma e outra vez em vários arquivos fonte, os modelos também podem ser declarados. Quando os modelos são declarados, o compilador não processará as definições dos modelos uma e outra vez e múltiplas instanciações idênticas não serão criadas. Claro que como qualquer outra declaração a implementação deve estar disponível em outra parte. A diferença da situação encontrada com funções concretas, que usualmente estão em bibliotecas, isto correntemente não é possível para os modelos de funções. Conseqüentemente, isto coloca uma carga nos ombros do engenheiro de software, que terá que assegurar que existam as instanciações requeridas. Um modo simples de se conseguir isto está abaixo.

Uma declaração de um modelo de função é simples de criar: O corpo da função é substituído por dois pontos. Note que este é idêntico ao modo que são construídas as declarações das funções concretas. Portanto, o modelo previamente definido da função 'add()' pode ser simplesmente declarado como:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue);
```

Nós já encontramos declarações de modelos. O arquivo cabeçalho 'iosfwd' pode ser incluído em fontes que não requerem instanciação de elementos da classe 'ios' e suas classes derivadas. Por exemplo, para compilar a declaração:

```
std::string getCsvline(std::istream &in, char const *delim);
```

Não é necessário incluir os arquivos cabeçalho 'string' e 'istream'. Em lugar disso uma simples declaração como:

```
#include <iosfwd>
```

É suficiente, requerendo cerca de um nono do tempo que toma para compilar a declaração com 'string' e 'istream'.

18.3.1: Instanciação de declarações

Portanto se declarando os modelos de funções acelera as fases de compilação e de linkagem

de um programa, como podemos assegurar-nos de que as instanciações requeridas dos modelos de funções estarão disponíveis quando o programa seja linkado?

Para esta variante de declaração está disponível, a assim chamada declaração explícita de instanciação.

Uma declaração explícita de instanciação contém os seguintes elementos:

- Começa com a palavra chave 'template', omitindo a lista de modelos de parâmetros;
- Em seguida o tipo de retorno e nome são especificados;
- O nome da função é seguido pela lista de especificação de tipos, uma lista de tipos entre parênteses angulares, cada tipo especificando o tipo do modelo de parâmetro correspondente à lista de modelos de parâmetros.
- Finalmente a lista de parâmetros da função é especificada, terminando por ponto e vírgula.

Apesar de que esta é uma declaração, é entendida pelo compilador como uma requisição de instanciar essa variante da função.

Usando-se declarações explícitas de instanciação todas as instanciações de modelos requeridas pelo programa podem ser coletadas num arquivo. Este arquivo, que pode ser um arquivo comum de fontes, pode incluir o arquivo cabeçalho com as definições de modelos e pode em seguida especificar as declarações de instanciação requeridas. Como é um arquivo fonte, não será incluído em outros arquivos fonte. Assim, as diretivas de espaços nomeados e declarações podem ser usadas com segurança, uma vez incluídos os cabeçalhos requeridos. Eis um exemplo mostrando a instanciação requerida de nosso modelo 'add()', instanciada para os tipos 'double', 'int' e 'std::string':

```
#include "add.h"
#include <string>
using namespace std;

template int add<int>(int const &lvalue, int const &rvalue);
template double add<double>(double const &lvalue, double const &rvalue);
template string add<string>(string const &lvalue, string const &rvalue);
```

Se somos desleixados e esquecemos de mencionar uma instanciação requerida por nosso programa, então a reparação pode ser feita com facilidade: Só ajunte a declaração de instanciação faltante na lista acima. Depois re-compile e re-linke o programa e pronto.

18.4: Instanciando modelos de funções

Um modelo não é instanciado quando sua definição é lida pelo compilador. Um modelo é somente uma receita que diz ao compilador como criar um código particular quando seja o momento de o fazer. É muito parecido a uma receita num livro de cozinha: Ao se ler uma receita de bolo não quer dizer que assamos um bolo no momento da leitura da receita.

Então, quando o modelo de uma função é instanciado? O compilador tem dois argumentos para instanciar os modelos:

- São instanciados quando usados (p.ex., a função 'add()' é chamada com um par de valores `size_t`);
- Quando endereços de modelos de funções são encontrados são instanciados. Por exemplo:

```
#include "add.h"

char (*addptr)(char const &, char const &) = add;
```

O local de comandos que causam a instanciação de modelos pelo compilador são ditos pontos de instanciação de modelos. O ponto de instanciação tem sérias implicações para o código do modelo de função. Estas implicações são discutidas na seção 18.9.

O compilador nem sempre é capaz de deduzir os tipos dos modelos de parâmetros sem ambigüidades. Nesse caso o compilador reporta uma ambigüidade que deve ser resolvida pelo engenheiro de software. Considere o seguinte código:

```
#include <iostream>
#include "sumvector.h"

size_t fun(int (*f)(int *p, size_t n));
double fun(double (*f)(double *p, size_t n));

int main()
{
    std::cout << fun(sumVector) << std::endl;
}
```

Quando este pequeno programa é compilado, o compilador reporta ambigüidade que não pode resolver. Ele tem duas funções candidatas, uma para cada versão sobrecarregada de 'call()' uma instanciação própria de 'add()' pode ser construída:

```
error: call of overloaded `call(<unknown type>)' is ambiguous
note: candidates are: int fan(size_t (*)(int*, size_t))
note:                  double fan(double (*)(double*, size_t))
```

É claro que situações como esta devem ser evitadas. Os modelos de funções só devem ser instanciados se não houver ambigüidade. As ambigüidades aparecem quando emergem múltiplas funções no mecanismo de seleção de funções do compilador (veja seção 18.8). Compete a nós resolver estas ambigüidades. As ambigüidades como a acima podem ser resolvidas usando um abrupto 'static_cast' (já que selecionamos entre alternativas, todas possíveis e disponíveis):

```
#include <iostream>
#include "add.h"

int call(int (*f)(int const &lvalue, int const &rvalue));
double call(double (*f)(double const &lvalue, double const &rvalue));

int main()
{
    std::cout << call(
        static_cast<int (*) (int const &, int const &)>(add)
    ) << std::endl;
    return 0;
}
```

Mas se possível, devemos evitar 'casts' de tipo. Como evitar tais situações está explicado na seção seguinte (18.5).

Como mencionado na seção 18.3, o linkador remove instanciações múltiplas idênticas de um modelo no programa final, deixando somente uma instância para cada conjunto único do tipo de modelo dos parâmetros. vejamos um exemplo mostrando o comportamento do linkador. Para ilustrar o comportamento do linkador faremos como segue:

- Primeiro construímos diversos arquivos fonte:
 - 'source1.cc' define uma função 'call()', instanciando 'add()' para argumentos do tipo inteiro, incluindo a definição do modelo de 'add()'. ele mostra o endereço de 'add()'. Eis 'source1.cc':

```
union PointerUnion
{
    int (*fp)(int const &, int const &);
    void *vp;
};

#include <iostream>
#include "add.h"
#include "pointerunion.h"

void call()
{
    PointerUnion pu = { add };
}
```

```
std::cout << pu.vp << std::endl;
}
```

- 'source2.cc' define a mesma função, mas só declara o modelo de 'add()' adequado, usando uma declaração (não uma declaração de instanciação). Eis o 'source2.cc':

```
#include <iostream>
#include "pointerunion.h"

template<typename Type>
Type add(Type const &, Type const &);

void call()
{
    PointerUnion pu = { add };

    std::cout << pu.vp << std::endl;
}
```

- 'main.cc' outra vez inclui a definição do modelo de 'add()', declara a função 'call()' e define 'main()', definindo 'add()' para o tipo de argumentos inteiros também e mostra o endereço de 'add()'. Também chama a função 'call()'. Aqui está 'main.cc':

```
#include <iostream>
#include "add.h"
#include "pointerunion.h"

void call();

int main()
{
    PointerUnion pu = { add };

    call();
    std::cout << pu.vp << std::endl;
}
```

- Todas as fontes são compiladas em módulos objeto. Note a diferença de tamanhos de 'source1.o' (2480 bytes, usando g++ versão 3.3.3. Todos os tamanhos reportados aqui podem diferir um pouco para diferentes compiladores e/ou bibliotecas em tempo de execução) e 'source2.o' (2372 bytes): Como 'source1.o' contém a instanciação de 'add()' é algo maior que 'source2.o', que contém somente a declaração do modelo. Agora estamos prontos a linkar nosso pequeno experimento.
- Linkando 'main.o' e 'source1.o', linkamos, obviamente, dois módulos objeto, cada um contendo sua própria instanciação do mesmo modelo de função. O programa resultante produz a seguinte saída:

0x8048752
0x8048752

Ainda mais, o tamanho do programa resultante é 14514 bytes.

- Linkando 'main.o' e 'source2.o', agora linkamos um módulo objeto contendo a instanciação do modelo de 'add()' e outro módulo objeto contendo só a declaração do mesmo modelo da função. Assim, o programa resultante só pode conter uma instanciação do modelo da função requerida. Este programa tem o mesmo tamanho e produz exatamente a mesma saída que o primeiro programa.

Portanto de nosso pequeno experimento podemos concluir que, efetivamente, o linkador remove instanciações idênticas de modelos do programa final e que usando meramente declarações não resultará em instanciações de modelos.

18.5: Usando modelos explícitos de tipos

Na seção anterior (18.4) vimos que o compilador pode encontrar ambigüidades ao instanciar um modelo. Vimos um exemplo onde versões sobrecarregadas de uma função 'call()' existentes, que esperam diferentes tipos de argumentos e que ambas poderiam ser instâncias, indiferentemente, do modelo da função. O modo intuitivo de resolver tal ambigüidade é usar um 'static_cast', mas como foi dito, os 'casts' devem ser evitados.

Quando estão envolvidos modelos de funções tal 'static_cast' pode, sem dúvida, ser evitado, usando-se argumentos com modelos explícitos de tipos. Quando são usados modelos explícitos de tipos o compilador é informado explicitamente sobre os tipos dos modelos de parâmetros que deve usar na instanciação do modelo. Aqui o nome da função é seguido por uma lista de tipos de parâmetros que deve ser seguida pela lista de argumentos da função. Os tipos mencionados na lista de modelo de parâmetros são usados pelo compilador para `deduzir` os tipos atuais correspondentes aos modelos da lista dos tipos do modelo de parâmetros da função. Eis o mesmo exemplo da seção anterior, agora usando modelos explícitos de tipos de argumentos:

```
#include <iostream>
#include "add.h"

int call(int (*f)(int const &lvalue, int const &rvalue));
double call(double (*f)(double const &lvalue, double const &rvalue));

int main()
{
```

```

        std::cout << call(add<int>) << std::endl;
        return 0;
    }

```

18.6: Sobrecarregando modelos de funções

Vejamos uma vez mais nosso modelo de 'add()'. Esse modelo foi projetado para retornar a soma de duas entidades. Se quiséssemos computar a soma de três entidades poderíamos escrever:

```

int main()
{
    add(2, add(3, 4));
}

```

Esta é uma solução perfeitamente aceitável para a situação ocasional. Contudo, se tivéssemos que somar três entidades regularmente uma versão sobrecarregada da função 'add()', que espere três argumentos, seria útil. A solução deste problema é simples: Os modelos de funções podem ser sobrecarregados.

Para se definir uma versão sobrecarregada só se põe múltiplas definições do modelo no seu arquivo cabeçalho. Assim, com a função 'add()' seria algo como:

```

template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}

template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}

```

A função sobrecarregada não tem que ser definida em termos de simples valores. Como todas funções sobrecarregadas, um único conjunto de parâmetros da função é suficiente para definir uma versão sobrecarregada. Por exemplo, aqui está uma versão sobrecarregada usada para computar a soma dos elementos de um vetor:

```

template <typename Type>
Type add(std::vector<Type> const &vect)
{
    return accumulate(vect.begin(), vect.end(), Type());
}

```

A sobrecarga de modelos não se restringe à lista de parâmetros da função. A própria lista de tipos de modelos de parâmetros pode ser sobrecarregada. A última definição do modelo de 'add()' nos

permite especificar um 'std::vector' como seu primeiro argumento, mas não 'deque' ou mapa. Versões sobrecarregadas para esses tipos de recipientes, claro, podem ser construídas, mas onde está o fim disto? Em vez disto, vejamos as características comuns desses recipientes e se encontramos, definir um modelo da função sobrecarregado sobre essas características. Uma característica comum dos recipientes mencionados é que suportam os membros 'begin()' e 'end()', retornando iteradores. Usando isto podemos definir um modelo de tipo de parâmetro que represente recipientes que suportam esses membros. Mas mencionando um 'tipo de recipiente' pleno nada nos diz para que tipo de dados foram instanciados. Assim, necessitamos um segundo modelo de tipo de parâmetro que represente o tipo de dados, sobrecarregando a lista de modelos de tipos de parâmetros. Eis a versão sobrecarregada do modelo de 'add()':

```
template <typename Container, typename Type>
Type add(Container const &cont, Type const &init)
{
    return std::accumulate(cont.begin(), cont.end(), init);
}
```

Com todas essas versões sobrecarregadas em seus lugares, podemos compilar a seguinte função:

```
using namespace std;

int main()
{
    vector<int> v;

    add(3, 4);           // 1 (veja o texto)
    add(v);              // 2
    add(v, 0);           // 3
}
```

- Com o primeiro comando o compilador reconhece dois tipos idênticos, ambos inteiros. Portanto instanciará 'add<int>', nossa primeira versão do modelo de 'add()';
- Com o segundo comando um simples argumento é usado. Conseqüentemente o compilador buscará uma versão sobrecarregada de 'add()' que requeira só um argumento. Encontra a versão que espera 'std::vector', deduzindo que o tipo de modelo do parâmetro tem que ser inteiro. Instanciando:

```
add<int>(std::vector<int> const &)
```

- Com o comando número três o compilador novamente encontra uma lista de argumentos com dois argumentos. Contudo, os tipos de argumentos são distintos, assim, não pode usar o modelo de 'add()' da primeira definição. Mas pode usar a última definição, que espera entidades com tipos diferentes. Como um 'std::vector' suporta 'begin()' e 'end()', o compilador, agora, está capacitado de instanciar o modelo de função:


```
add<std::vector<int>, int>(std::vector<int const &)
```

Tendo definido 'add()' usando dois tipos de modelo de parâmetros diferentes e um modelo de função tendo uma lista de parâmetros contendo dois parâmetros desses tipos, exaurimos as possibilidades de definir uma função 'add()' tendo listas de parâmetros da função de dois tipos diferentes. Mesmo que os tipos de parâmetros sejam diferentes, somos capazes de definir um modelo de função 'add()' retornando a soma de dois tipos de entidades:

```
template <typename T1, typename T2>
T1 add(T1 const &lvalue, T2 const &rvalue)
{
    return lvalue + rvalue;
}
```

Contudo, agora não somos capazes de instanciar 'add()' usando dois tipos diferentes de argumentos: O compilador não está capacitado de resolver a ambigüidade, não pode escolher entre as duas versões sobrecarregadas que definem dois tipos diferentes de parâmetros para a função:

```
int main()
{
    add(3, 4.5);
}
/*
O compilador reporta:

error: call of overloaded `add(int, double)` is ambiguous
error: candidates are: Type add(const Container&, const Type&)
                        [with Container = int, Type = double]
error:                  T1 add(const T1&, const T2&)
                        [with T1 = int, T2 = double]
*/
```

Consideremos uma vez mais a função sobrecarregada que aceita três argumentos:

```
template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

Pode ser considerado uma desvantagem que só tipos iguais de argumentos são aceitos pela função. P.ex., três inteiros, três duplos ou três 'strings'. Para remediar isto, definimos ainda outra versão sobrecarregada da função, esta vez aceitando argumentos de qualquer tipo. Claro, quando chamamos esta função devemos estar seguros de que o 'operator+()' está definido entre eles, aparte disto parece não haver problemas. Eis a versão que aceita qualquer tipo:

```
template <typename Type1, typename Type2, typename Type3>
Type1 add(Type1 const &lvalue, Type2 const &mvalue, Type3 const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

```
}
```

Depois de definir estas duas versões sobrecarregadas, chamemo-as:

```
add(1, 2, 3);
```

Neste caso poderíamos esperar que o compilador reportasse ambigüidade. Depois de tudo o compilador deve selecionar a forma da função, deduzindo que 'Type == int', mas poderia selecionar a última função, deduzindo que 'Type1 == int', 'Type2 == int' e 'Type3 == int'. Contudo o compilador não reporta ambigüidade. A razão para isto é a seguinte: Se um modelo de função sobrecarregado é definido usando tipos de modelos de parâmetros mais especializados (P.EX., todos de igual tipo) então outra função sobrecarregada com tipos de parâmetros mais gerais (p.ex., todos diferentes) existir, então o compilador selecionará a mais especializada sempre que possível.

Como regra prática: Quando são definidas versões sobrecarregadas do modelo de uma função, cada versão sobrecarregada deve usar uma única combinação de tipos dos modelos de parâmetros para evitar ambigüidades quando os modelos sejam instanciados. Note que a ordem dos tipos dos modelos dos parâmetros na lista de parâmetros da função não é importante. Quando se tentar instanciar o seguinte modelo de 'binarg()', uma ambigüidade ocorrerá:

```
template <typename T1, typename T2>
void binarg(T1 const &first, T2 const &second)
{}
// e:
template <typename T1, typename T2>
void binarg(T2 const &first, T1 const &second) // troca entre T1 e T2
{}

```

A ambigüidade aparece sem surpresa. Depois de tudo, tipos de modelo de parâmetros são só nomes formais. Seus nomes (T1, T2 ou o que seja) não possuem um significado concreto.

Finalmente, funções sobrecarregadas podem ser declaradas usando-se declarações plenas ou declarações de instanciação e tipos de modelos de parâmetros também podem ser usados. Por exemplo:

- Declaração de uma função 'add()' que aceite recipientes de certo tipo:

```
template <typename Container, typename Type>
Type add(Container const &container, Type const &init);

```

- A mesma função, mas agora usando uma declaração de instanciação (note que esta requer que o compilador já tenha conhecimento da definição do modelo:

```
template int add<std::vector<int>, int>
(std::vector<int> const &vect, int const &init);

```

- Para eliminar ambigüidades entre múltiplas possibilidades detetadas pelo compilador, argumentos explícitos podem ser usados. Por exemplo:

```
std::vector<int> vi;
int sum = add<std::vector<int>, int>(vi, 0);
```

18.7: Especializando os modelos para tipos com desviação

O modelo inicial de 'add()', que define dois tipos de parâmetros idênticos funciona bem para todos os tipos que suportem 'operator+()' e um construtor de cópias. Contudo, estes pressupostos nem sempre são encontrados. Por exemplo, quando 'char *s' é usado, nem o 'operator+()' e nem um construtor de cópias estão disponíveis. O compilador não sabe disto e tentará instanciar o simples modelo da função:

```
template <typename Type>
Type add(Type const &t1, Type const &t2);
```

Mas não consegue, já que 'operator+()' não está definido para ponteiros. Nestas situações é claro que a concordância entre o tipo do parâmetro(s) do modelo e o tipo(s) usado é possível, mas a implantação estandarte não tem sentido ou produz erros.

Para resolver este problema uma especialização explícita do modelo pode ser definida. Uma especialização explícita do modelo define o modelo da função para o qual uma definição genérica já existe, usando tipo específico de modelo de parâmetro.

No caso mencionado acima uma especialização explícita é requerida para 'char const *', mas provavelmente também para o tipo 'char *'. Provavelmente, já que o compilador ainda usa o processo de dedução de tipo padrão, mencionado anteriormente. Assim, quando nosso modelo de função 'add()' for especializada para argumentos 'char *', então seu tipo de retorno deve ser também um 'char *', visto que precisa ser 'char const *' se os argumentos são valores 'char const *'. Nestes casos o tipo do modelo de parâmetro 'Type' será deduzido propriamente. Com 'Type == char *', por exemplo, o cabeçalho da função instanciada fica:

```
char *add(char *const &t1, char *const &t2)
```

Se isto é considerado indesejável, uma versão sobrecarregada poderia ser designada para esperar ponteiros. A seguinte definição de modelo de função espera dois ('const') ponteiros e retorna um ponteiro não constante:

```
template <typename T>
T *add(T const *t1, T const *t2)
{
    std::cout << "Ponteiros\n";
    return new T;
}
```

Mas ainda não estamos onde queremos, já que esta versão sobrecarregada agora só aceita

ponteiro a elementos 'T' constantes. Ponteiros a elementos 'T' não constantes não serão aceitos. À primeira vista aparece como surpresa que o compilador não aplicará uma conversão de qualificação. Mas não há necessidade do compilador o fazer: Quando são usados ponteiros não constantes o compilador simplesmente usa a definição inicial do modelo da função 'add()', que espera qualquer dois argumentos de iguais tipos.

Portanto devemos definir ainda outra versão sobrecarregada, que espere apontadores não constantes? É possível, mas em algum lugar nos ficará claro que sobrepassamos nosso objetivo. Como funções e classes concretas, os modelos devem ter propósitos bem descritos. Tentando agregar definições sobrecarregadas de modelos sobre definições sobrecarregadas de modelos, rapidamente o modelo se torna uma babel. Não siga esta solução. Uma solução melhor, provavelmente, é construir o modelo de tal forma que se oriente ao propósito original, dar abertura a ocasionais casos específicos e descrever seu propósito claramente na documentação do modelo.

Não obstante, há situações onde uma especialização explícita de um modelo pode ser considerada de mérito. Duas especializações para apontadores a caracteres constantes e não constantes podem ser consideradas para nosso modelo da função 'add()'. Especializações explícitas de modelos são construídas como segue:

- Começam com a palavra chave 'template';
- Em seguida colocamos parênteses angulares vazios. Isto indica ao compilador que deve existir um modelo cujo protótipo coincide com o que estamos definindo. Se não há tal modelo, então o compilador reportará um erro como:

```
error: template-id 'add<char*>' for 'char* add(char* const&, char*
      const&)' does not match any template declaration
```

- Em seguida o cabeçalho da função é definido, que deve seguir a mesma sintaxe que a declaração de instanciação explícita de modelos (veja a seção 18.3.1): Deve especificar o tipo correto de retorno, nome da função, explicitações de tipos de modelos de parâmetros bem como a lista de parâmetros da função;
- O corpo da função, definindo a implementação especial requerida para os tipos especiais de modelos de parâmetros.

Eis duas especializações explícitas para o modelo da função 'add()', esperando como argumentos 'char *' e 'char const *' (note que 'const' que ainda aparece na primeira especialização do modelo não tem relação com o tipo especializado ('char *'), mas se refere a 'const &' mencionado na definição do modelo original. Portanto, neste caso, sua referência a um ponteiro constante a um 'char', implicando que os caracteres podem ser modificados):

```

template <> char *add<char *>(char * const &p1,
                               char * const &p2)
{
    std::string str(p1);
    str += p2;
    return strcpy(new char[str.length() + 1], str.c_str());
}

template <> char const *add<char const *>(char const *const &p1,
                                           char const *const &p2)
{
    static std::string str;
    str = p1;
    str += p2;
    return str.c_str();
}

```

As especializações explícitas de modelos normalmente são incluídas no arquivo que contém as implantações de outros modelos de funções.

Uma especialização explícita de modelo pode ser declarada de modo usual, i.e., substituindo seu corpo com ponto e vírgula.

Em particular, note a importância do par de parênteses angulares que segue a palavra chave 'template' na declaração de uma especialização explícita de modelo. Se os parênteses angulares são omitidos, então teremos uma declaração de instanciação de modelo. O compilador a processará silenciosamente, às expensas de um tempo algo maior de compilação.

Ao declarar uma especialização explícita de modelo (ou uma declaração de instanciação) a especificação explícita do tipo de modelo dos parâmetros pode ser omitida se o compilador for capaz de deduzir esses tipos dos argumentos da função. Como este é o caso de 'char (const) *' nas especializações, também poderiam ser declaradas como segue:

```

template <> char const *add(char const *const &p1,
                           char const *const &p2);
template <> char const *add(char const *const &p1,
                           char const *const &p2);

```

Além disso, 'template <>'. Contudo, isto removeria o caráter de modelo da declaração, resultando na declaração de nada mais que de uma função plena. Isto não é um erro: Modelos de funções e funções concretas podem sobrecarregar uma à outra. As funções ordinárias não são tão restritivas como os modelos de funções em relação às conversões de tipos. Isto poderia ser a razão de sobrecarregar um modelo com uma função ordinária mais freqüentemente.

18.8: O mecanismo de seleção de modelos de funções

Quando o compilador encontra uma chamada a uma função tem que decidir qual função chamar, quando estão disponíveis funções sobrecarregadas. Nesta seção é descrito o mecanismo de seleção de funções.

Em nossa discussão assumimos que pedimos ao compilador para compilar a seguinte função 'main()':

```
int main()
{
    double x = 12.5;
    add(x, 12.5);
}
```

Ainda mais, assumimos que o compilador examinou as seguintes seis declarações de funções quando compila 'main()':

```
template <typename Type>                                // função 1
Type add(Type const &lvalue, Type const &rvalue);

template <typename Type1, typename Type2>                // função 2
Type1 add(Type1 const &lvalue, Type2 const &rvalue);

template <typename Type1, typename Type2, typename Type3> // função 3
Type1 add(Type1 const &lvalue, Type1 const &mvalue, Type2 const &rvalue);

double add(float lvalue, double rvalue);                // função 4
double add(std::vector<double> const &vd);                // função 5
double divide(double lvalue, double rvalue);            // função 6
```

O compilador, tendo lido o comando de 'main()', deve agora decidir qual função deve ser chamada. Ele procede como segue:

- Primeiro, um conjunto de funções candidatas é construído. Este conjunto contém todas as funções que:
 - São visíveis no ponto da chamada;
 - Têm o mesmo nome que a função chamada

Como a função 6 tem nome diferente, é removida do conjunto. O compilador ficou com um conjunto de cinco funções candidatas: de 1 a 5.

- Segundo, o conjunto de funções viáveis é construído. Funções viáveis são aquelas para as quais as conversões de tipo existentes podem ser aplicadas para concordarem com os tipos dos

parâmetros atuais. Isto implica que o número de argumentos deve concordar com o número de parâmetros das funções viáveis. Como as funções 3 e 5 têm diferentes números de parâmetros, são removidas do conjunto. A função 1 é removida, já que requer tipos idênticos de parâmetros. A função 2, contudo, é mantida, já que 'Type1' concorda com 'double' depois de aplicada a transformação para um 'lvalue' ao argumento 'x', seguida de uma transformação de qualificação agregando 'const' a 'double'. Da mesma forma, a função 4 é mantida, já que existe uma conversão padrão de 'double' a 'float'. O segundo argumento de ambas funções estão diretamente concordados. Assim, o conjunto de funções viáveis consiste das funções 2 e 4.

- Terceiro, as funções restantes são ordenadas em preferência e a primeira será usada. A ordenação é feita dando pontos a cada um dos parâmetros da função e adicionando esses pontos para todos os parâmetros da função. A função recebe um ponto se possui uma concordância direta e nenhum ponto se for necessária uma conversão. Vejamos como ocorre:

- Para o modelo da função, a função 'add<double, double>' ser instanciada. Os tipos dos dois parâmetros e argumentos concordam, nenhuma conversão é requerida. Cada parâmetro recebe um ponto, resultando dois pontos para o modelo da função.
- Para a função 'add<float, double>' o tipo do segundo parâmetro concorda exatamente com o tipo do segundo argumento, mas uma conversão (padrão) de 'double' a 'float' é requerida para o primeiro argumento. Conseqüentemente a pontagem desta função é igual a 1.

Desta seleção o modelo da função sai vitoriosa. Portanto é selecionada para ser usada. É instanciada e chamada por 'main()'.

Como exercício submeta as seis declarações acima e a 'main()' ao compilador e espere os erros de linkagem: O linkador emitirá a mensagem:

```
void add<double, double>(double const&, double const&)
is an undefined reference.
```

Quando diversas funções aparecem, o compilador não reporta automaticamente uma ambigüidade. Como vimos antes, um função mais especializada é selecionada em face de uma mais geral, e uma função concreta em face de um modelo de função (mas só se tiverem a mesma pontagem na seleção).

Como regra prática considere que existem muitas funções viáveis no topo do conjunto das funções viáveis, então as instanciações do modelo pleno da função são removidas. Se restam muitas funções, especializações explícitas do modelo são removidas. Se restar só uma função é selecionada, de outra forma a chamada é ambígua.

18.9: Compilando definições de modelos e instâncias

Considere a seguinte definição do modelo da função 'add()':

```
template <typename Container, typename Type>
Type add(Container const &container, Type init)
{
    return std::accumulate(container.begin(), container.end(), init);
}
```

Nesta definição de modelo 'std::accumulate()' é chamada usando os membros 'begin()' e 'end()' de 'container'.

As chamadas 'container.begin()' e 'container.end()' são ditos dependentes do tipo dos parâmetros do modelo. O compilador, sem conhecimento da interface de 'container' terá os membros 'begin()' e 'end()' retornando iteradores de entrada, segundo o requerido por 'std::accumulate()'.

Por outro lado, 'std::accumulate()' é uma chamada a função independente de qualquer tipo de modelo de parâmetro. Seus argumentos o são, mas ela própria não. Os comando no corpo de um modelo independentes dos tipos de parâmetros do modelo são ditos independentes dos tipos de parâmetros.

Quando o compilador lê uma definição de modelo, verificará a correção sintática de todos os comandos independentes dos tipos de parâmetros. I.e., deve ter examinado todas as definições de classe, todas as definições de tipo, todas as declarações de funções, etc., que são usadas nos comandos independentes dos tipos de parâmetros do modelo. Se esta condição não é satisfeita, o compilador não aceitará a definição. Conseqüentemente quando se definir o modelo acima, o arquivo cabeçalho 'numeric' deve estar incluído, pois ali está declarada std::accumulate().

Por outro lado, com comandos dependentes dos tipos de parâmetros do modelo o compilador não pode realizar estes exames extensivos, como não tem, por exemplo, meios de verificar a existência de um membro 'begin()' para o tipo, ainda não especificado, 'Container'. Nestes casos o compilador realizará exames especiais, assumindo que os membros requeridos, operadores e tipos se tornarão disponíveis.

O local no programa fonte onde o modelo é instanciado é chamado ponto de instanciação. Nesse ponto o compilador deduzirá os tipos dos tipos de parâmetros do modelo. No ponto de instanciação examinará a correção sintática dos comandos do modelo que dependem dos tipos de parâmetros do modelo. Isto implica que só no ponto de instanciação as declarações requeridas devem ter sido lidas pelo compilador. Como regra prática, assegure-se de que todas as declarações requeridas (usualmente arquivos cabeçalhos) tenham sido lidas pelo compilador em cada ponto de instanciação do modelo. Para a definição do modelo um requerimento mais relaxado pode ser formulado. Quando uma definição é lida

só as declarações requeridas para os comandos não dependentes dos tipos de parâmetros do modelo devem ser conhecidas.

18.10: Sumário da sintaxe da declaração do modelo

Abaixo estão sumarizados exemplos básicos de formas sintáticas de declarações de modelos. Com formas que permitam definições, o ponto e vírgula de terminação pode ser substituído pelo corpo da função. Se uma declaração também possa ser mudada em definição, então isto está mencionado.

- Uma declaração plena de um modelo (uma definição é possível):

```
template <typename Type1, typename Type2>
void function(Type1 const &t1, Type2 const &t2);
```

- Uma declaração de instanciação de um modelo:

```
template
void function<int, double>(Type1 const &t1, Type2 const &t2);
```

- Um modelo usando tipos explícitos:

```
void (*fp)(double, double) = function<double, double>;
void (*fp)(int, int) = function<int, int>;
```

- Uma especialização de um modelo (é possível uma definição):

```
template <>
void function<char *, char *>(Type1 const &t1, Type2 const &t2);
```

- Uma declaração de modelos de funções amigas em modelos de classes (coberto na seção 19.8):

```
friend void function<Type1, Type2>(parameters);
```

Capítulo 19: Modelos de classes

Como para funções, pode-se construir modelos para classes completas. Um modelo de uma classe deve ser considerado quando uma classe deve estar capacitada a manipular diferentes tipos de dados. Modelos de classes são freqüentemente usados em C++. No Capítulo 12 que cobre estruturas de dados gerais como vetores, pilhas e filas, são definidos como modelos de classes. Com os modelos de classes, os algoritmos e os dados sobre os quais operam são considerados completamente separados um do outro. Para usar uma estrutura de dados particular, que opere sobre um tipo particular de dados, somente o tipo de dados necessita ser especificado quando o modelo de classe objeto é definido ou declarado, p.ex., 'stack<int> i stack'.

Abaixo a construção de modelos de classes é discutida. Num sentido, modelos de classes compete à programação orientada ao objeto (cf., Capítulo 14), onde um mecanismo algo similar aos modelos é visto. O polimorfismo permite ao programador pós-por as definições dos algoritmos, através da derivação de classes de uma classe de base na qual os algoritmos operantes primeiro são definidos em classes derivadas junto com as funções membro que foram definidas como funções puramente virtuais na classe de base para manipular dados. Por outro lado, os modelos permitem ao programador pospor a identificação dos dados sobre os quais o algoritmo opera. Isto se parece mais claramente com os recipientes abstratos, especificando completamente os algoritmos mas ao mesmo tempo deixando o tipo de dados sobre os que os algoritmos operam completamente sem especificação.

Geralmente os modelos de classes são de fácil uso. Com certeza é mais fácil escrever 'stack<int> istack' para criar uma pilha de inteiros que derivar uma nova classe 'Istack: public stack' e implantar todas as funções membro necessárias para se estar capacitado a uma pilha similar de inteiros usando a programação orientada a objetos. Por outro lado, para cada tipo diferente que é usado com o modelo de uma classe a classe completa é re-instanciada, ao passo que no contexto da programação orientada ao objeto as classes derivadas usam, no lugar de uma cópia, as funções que já estão disponíveis na classe de base (mas veja também seção 19.9).

19.1: Definindo modelos de classes

Agora que vimos a construção de modelos de funções, estamos prontos para o passo seguinte, a construção de modelos de classes. Muitos modelos úteis de classes já existem. No lugar de ilustrar como foi construído um modelo existente de classe discutamos a construção de um novo modelo

útil de classe.

No Capítulo 17 encontramos a classe 'auto_ptr' (seção 17.3). A 'auto_ptr', também dita 'o ponteiro esperto', permite-nos definir um objeto, que atua como um apontador. Usando 'auto_ptr' no lugar de apontadores plenos asseguramos não só uma gestão própria da memória, mas também podemos prevenir fugas de memórias quando objetos de classes que usam membros de dados ponteiros não podem ser completamente construídos.

Uma desvantagem de 'auto_ptr' é que só pode ser usada para objetos singulares e não para apontadores a conjuntos de objetos. Aqui construiremos o modelo da classe 'FBB::auto_ptr', comportando-se como 'auto_ptr', mas gerenciando apontadores a conjuntos de objetos.

Usando uma classe existente como nosso ponto de partida, mostra também um princípio importante: Frequentemente é mais fácil construir um modelo (função ou classe) a partir de um modelo existente que construir um modelo completamente desde o princípio. Neste caso, a classe existente 'std::auto_ptr' atua como projeto. Para isso queremos a classe com os seguintes membros:

- Construtores para criar um objeto da classe 'FBB::auto_ptr';
- Um destrutor;
- Um 'operator=()' sobrecarregado
- Um 'operator[]()' para recuperar e re-adjudicar os elementos dando seus índices.
- Todos os outros membros de 'std::auto_ptr', à exceção do operador de deferência ('operator*()'), já que os objetos de nossa 'FBB::auto_ptr' conterão múltiplos objetos e portanto o operador de deferência não pode ser definido sem ambigüidade.

Agora que decidimos que membros necessitamos, a interface de classe pode ser construída. Como os modelos de funções, uma definição de modelo de classe começa com a palavra chave 'template', que também é seguida por uma lista não vazia de modelos de tipos e/ou parâmetros sem tipo, envolta em parênteses angulares. A palavra chave 'template' seguida pela lista de parâmetros envolta em parênteses angulares é chamado de anúncio de modelo nas Anotações C++. Em alguns casos a lista de parâmetros do anúncio de modelo pode estar vazia, deixando apenas os parênteses angulares.

Seguindo ao anúncio de modelo segue a interface de classe, onde os tipos formais de parâmetros podem ser usados para representar tipos e constantes. A interface de classe é construída como usual. Começa com a palavra chave 'class' e termina com ponto e vírgula.

Considerações normais de projeto devem ser seguidas quando se constroi as funções membro do modelo de classe ou os construtores do modelo de classe: Os tipos de parâmetros do modelo de classe, de preferência, devem ser definidos como 'Type const &', antes que 'Type', para prevenir cópias desnecessárias de longas estruturas de dados. Os modelos dos construtores da classe devem usar iniciadores antes que adjudicação no corpo dos construtores, ainda para prevenir dupla adjudicação de objetos compostos: uma vez pelo construtor padrão do objeto, outra pela adjudicação mesma.

Eis nossa versão inicial da classe 'FBB::auto_ptr', mostrando todos seus membros:

```
namespace FBB
{
    template <typename Data>
    class auto_ptr
    {
        Data *d_data;

    public:
        auto_ptr()
        :
            d_data(0)
        {}
        auto_ptr(auto_ptr<Data> &other)
        {
            copy(other);
        }
        auto_ptr(Data *data)
        :
            d_data(data)
        {}
        ~auto_ptr()
        {
            destroy();
        }
        auto_ptr<Data> &operator=(auto_ptr<Data> &rvalue);
        Data &operator[](size_t index)
        {
            return d_data[index];
        }
        Data const &operator[](size_t index) const
        {
            return d_data[index];
        }
        Data *get()
        {
            return d_data;
        }
        Data const *get() const
        {
```

```

        return d_data;
    }
    Data *release();
    void reset(Data *);
private:
    void destroy()
    {
        delete [] d_data;
    }
    void copy(auto_ptr<Data> &other)
    {
        d_data = other.release();
    }
    Data &element(size_t idx) const;
};

template <typename Data>
inline auto_ptr<Data>::auto_ptr()
:
    d_data(0)
{}

template <typename Data>
inline auto_ptr<Data>::auto_ptr(auto_ptr<Data> &other)
{
    copy(other);
}

template <typename Data>
inline auto_ptr<Data>::auto_ptr(Data *data)
:
    d_data(data)
{}

template <typename Data>
inline auto_ptr<Data>::~~auto_ptr()
{
    destroy();
}

template <typename Data>
inline Data &auto_ptr<Data>::operator[](size_t index)
{
    return d_data[index];
}

template <typename Data>
inline Data const &auto_ptr<Data>::operator[](size_t index) const
{
    return d_data[index];
}

```

```

template <typename Data>
inline Data *auto_ptr<Data>::get()
{
    return d_data;
}

template <typename Data>
inline Data const *auto_ptr<Data>::get() const
{
    return d_data;
}

template <typename Data>
inline void auto_ptr<Data>::destroy()
{
    delete[] d_data;
}

template <typename Data>
inline void auto_ptr<Data>::copy(auto_ptr<Data> &other)
{
    d_data = other.release();
}

template <typename Data>
auto_ptr<Data> &auto_ptr<Data>::operator=(auto_ptr<Data> &rvalue)
{
    if (this != &rvalue)
    {
        destroy();
        copy(rvalue);
    }
    return *this;
}

template <typename Data>
Data *auto_ptr<Data>::release()
{
    Data *ret = d_data;
    d_data = 0;
    return ret;
}

template <typename Data>
void auto_ptr<Data>::reset(Data *ptr)
{
    destroy();
    d_data = ptr;
}

} // FBB

```

A interface de classe mostra as seguintes características:

- Se for assumido que o tipo modelo 'Data' é um tipo ordinário, a interface de classe parece não ter características especiais. Aparente como qualquer interface de classe anterior. Isto em geral é verdadeiro. Frequentemente um modelo de classe pode ser construído facilmente depois de se ter construído a classe para um ou dois tipos concretos, seguido por uma fase de abstração mudando todas as referências necessárias a tipos de dados concretos por tipos de dados genéricos, que então se transformam nos modelos de tipos de parâmetros.
- Numa inspeção mais próxima, podemos discernir algumas características especiais. Os parâmetros do construtor de cópias da classe e os operadores de adjudicação sobrecarregados não são referências plenas a objetos 'auto_ptr' (ou suas referências ou apontadores) sempre requerem o tipo de parâmetros do modelo para serem especificados.
- A diferença dos construtores de cópias e operadores de adjudicação sobrecarregados padrões, seus parâmetros são referências não constantes. Isto não tem nada a ver com o fato de ser um modelo de classe, mas é uma consequência do próprio projeto da 'auto_ptr': Ambos, o construtor de cópias e o operador de adjudicação sobrecarregado, toma o apontador a outros objetos, efetivamente mudando o outro objeto num apontador nulo.
- Como as classes ordinárias, os membros podem ser definidos em linha. Todos os membros do modelo de classe tem que ser definidos em linha, já que modelos pré-compilados, correntemente, não são suportados. Até mesmo a definição pode ser posta dentro da interface de classe ou fora dela (i.e., seguindo-a). Como regra prática os mesmos princípios das classes concretas devem ser seguidos aqui, mas desta vez, a razão principal é a legibilidade da interface: Implantações grandes da interface tendem a obscurecê-la.
- Quando os objetos de um modelo de classe são instanciados, as definições de todos os modelos das funções membro que são usados (mas só esses) devem ser vistos pelo compilador. Apesar que essa característica dos modelos poderia ser refinada ao ponto em que cada definição estar guardada num arquivo de definição de função separado, incluindo só as definições dos modelos das funções necessárias no momento, mesmo assim seria difícil (mesmo que aceleraria o tempo de compilação). Em vez disso, o modo usual de definição dos modelos de classes é definir a interface, com algumas definições em linha e definindo os restantes modelos de funções imediatamente abaixo da interface do modelo de classe.
- Além do operador de deferência ('operator*()), o par bem conhecido de 'operator[]()' são definidos. Como a classe não recebe informação sobre o tamanho dos conjuntos ('arrays') objetos,

estes membros não suportam exame de seus limites.

Examinemos algumas das funções membros definidas abaixo da interface de classe:

- Os membros estudados 'copy()' e 'destroy()' (veja seção 7.5.1) são muito simples, mas foram agregadas à implantação para estandarização de classes com membros apontadores;
- O construtor de adjudicação sobrecarregado tem que examinar até auto-adjudicação. Quando definida fora da interface sua implantação é:

```
namespace FBB
{
    template <typename Data>
    auto_ptr<Data> &auto_ptr<Data>::operator=(auto_ptr<Data> &rvalue)
    {
        if (this != &rvalue)
        {
            destroy();
            copy(rvalue);
        }
        return *this;
    }
}
```

Note em particular:

- Esta é a definição do modelo. Como é uma definição precisa começar com a frase de modelo. A declaração da função também tem que começar com a frase de modelo, mas isto por implicação da interface, que já fornece a frase requerida em seu começo;
- Todos os tipos 'auto_ptr' mencionados incluem o tipo do parâmetro do modelo. Isto é obrigatório;
- A implantação dos membros 'release()' e 'reset()' também são bastante simples. Aqui estão:

```
namespace FBB
{
    template <typename Data>
    Data *auto_ptr<Data>::release()
    {
        Data *ret = d_data;
        d_data = 0;
        return ret;
    }

    template <typename Data>
```



```

void auto_ptr<Data>::reset(Data *ptr)
{
    destroy();
    d_data = ptr;
}
}

```

Agora que a classe foi definida, pode ser usada. Para usá-la, seu objeto precisa ser instanciado para um tipo particular de dados. O exemplo define um novo conjunto 'std::string', que armazena todos os argumentos de uma linha de comando. Então o primeiro argumento da linha de comando é impresso. Em seguida, o objeto 'auto_ptr' é usado para iniciar outro 'auto_ptr' do mesmo tipo. É mostrado que o 'auto_ptr' original guarda, agora, um ponteiro nulo e o segundo os argumentos da linha de comando:

```

#include <iostream>
#include <algorithm>
#include <string>
#include "autoptr.h"
using namespace std;

int main(int argc, char **argv)
{
    FBB::auto_ptr<string> sp(new string[argc]);
    copy(argv, argv + argc, sp.get());

    cout << "Primeiro auto_ptr, nome do programa: " << sp[0] << endl;

    FBB::auto_ptr<string> second(sp);

    cout << "Primeiro auto_ptr, aponta agora: " << sp.get() << endl;
    cout << "Segundo auto_ptr, nome do programa: " << second[0] << endl;

    FBB::auto_ptr<> iap(new int[10]);

    cout << iap[0] << endl;
    return 0;
}
/*
Saída Gerada:

Primeiro auto_ptr, nome do programa: a.out
Primeiro auto_ptr, aponta agora: 0
Segundo auto_ptr, nome do programa: a.out
*/

```

19.1.1: Parâmetros padrão dos modelos de classe

A diferença dos modelos de funções, aos modelos de parâmetros dos modelos de classes se

pode dar valores padrão.

Isto é verdade para ambos modelos de parâmetros tipificados e não tipificados. Se um modelo de classe é instanciado sem especificar argumentos para seus modelos de parâmetros e se os valores padrão dos modelos dos parâmetros foram definidos, então os padrões são usados. Quando se define esses padrões tenha presente que devem ser adequados para a maioria das instâncias da classe. P.ex., para o modelo da classe 'FBB::auto_ptr' o tipo de modelo da lista de parâmetros poderia ter sido alterado especificando-se inteiro como seu tipo padrão:

```
template <typename Data = int>
```

Mesmo argumentos padrão podem ser especificados, o compilador ainda precisa ser informado que as definições de objetos se referem aos modelos. Assim, quando instanciamos objetos do modelo de uma classe para os quais foram definidos valores, as especificações de tipo podem ser omitidas, mas os parênteses angulares são necessários. Dessa forma, assumindo um tipo padrão para a classe 'FBB::auto_ptr', um objeto dessa classe pode ser definido como:

```
FBB::auto_ptr<> intAutoPtr;
```

Nenhum padrão precisa ser especificado para os modelos de membros definidos fora de sua interface de classe. Os modelos das funções, mesmo modelos das funções membro, não podem especificar valores para os parâmetros. Assim, a definição do, p.ex., membro 'release()' sempre começará sempre com a mesma especificação de modelo:

```
template <typename Data>
```

Quando um modelo de classe usa múltiplos modelos de parâmetros, a todos pode-se dar valores padrão. Contudo, como os argumentos padrão das funções, uma vez que um valor padrão é usado, todos os restantes parâmetros precisam usar seus valores padrão. Uma lista de especificação de modelos tipos não pode iniciar com vírgula e nem pode conter muitas vírgula consecutivas.

19.1.2: Declarando modelos de classes

Os modelos de classes também podem ser declarados. Isto pode ser útil em situações onde são requeridas declarações de classes em avanço. Para declarar um modelo de classe substitua sua interface (a parte entre chaves) por ponto e vírgula:

```
namespace FBB
{
    template <typename Type>
    class auto_ptr;
}
```

Aqui também se pode especificar tipos padrão. Contudo, valores de tipos padrões não, em

ambos, declaração e definição de um modelo de classe. Como regra prática os valores devem ser omitidos das declarações, já que as declarações de modelos de classes nunca são usadas para instanciar objetos, mas só para ocasionais referências avançadas. Note que isto difere das especificações de valores padrão de parâmetros para funções membro nas classes concretas. Tais padrões podem ser especificados nas declarações e não em suas definições.

19.1.3: Distingüindo membros e tipos formais de tipos de classes

Como um nome de modelo de tipo pode se referir a qualquer tipo, um nome de modelo de tipo também pode se referir a um modelo ou a uma classe. Assumamos um modelo de uma classe 'Handler' que define um 'typename' 'Container' como seu tipo de parâmetro e um membro de dados que armazena o iterador de 'container' 'begin()'. Ainda mais, o modelo da classe 'Handler' tem um construtor que aceita qualquer recipiente que suporte o membro 'begin()'. O esqueleto de nossa classe 'Handler' pode ser:

```
template <typename Container>
class Handler
{
    Container::const_iterator d_it;

public:
    Handler(Container const &container)
    :
        d_it(container.begin())
    {}
};
```

Quais eram as considerações que tínhamos na cabeça quando projetamos essa classe?

- O 'typename Container' representa qualquer recipiente que suporte iteradores;
- O recipiente presumivelmente suporta o membro 'begin()'. A iniciação 'd_it(container.begin())' claramente depende do tipo de parâmetro do modelo, portanto só é examinado para uma correta estrutura sintática básica;
- Igualmente, o recipiente presumivelmente suporte um tipo 'const_iterator', definido na classe 'Container'. Como 'container' é uma referência constante, o iterador retornado por 'begin()' é um 'const_iterator' antes que um iterador pleno.

Agora, ao instanciar um 'Handler' usando a seguinte função 'main()' caímos num erro de compilação:

```
#include "handler.h"
```

```
#include <vector>
using namespace std;

int main()
{
    vector<int> vi;
    PairHandler<vector<int> > ph(vi);
}
/*
    Erro Reportado:

handler.h:4: error: syntax error before `;' token
*/
```

Aparentemente a linha

```
Container::const_iterator d_it;
```

na classe 'Handler' causa um problema. O problema é o seguinte:

Quando usando tipos de modelos de parâmetros, um exame pleno da sintaxe permite ao compilador decidir se um 'container' se refere a um 'Container'. Tal 'container' suporta bem um membro 'begin()', daí 'container.begin()' está sintaticamente ok. Contudo, esse membro 'begin()' aqui não existe para um tipo particular de 'Container'. Claro está, a existência de 'begin()' só será conhecida quando o tipo de 'Container' for especificado. Por outro lado, o compilador é incapaz qual 'Container::const_iterator' é. Assumindo que é membro do misterioso 'Container', um exame sintático pleno falha, já que o comando

```
Container::const_iterator d_it;
```

é sempre sintaticamente errada quando 'const_iterator' é um membro de 'Container'. O compilador, sem dúvida, assume que 'X::a' é um membro da classe 'X' e é ilustrado pela mensagem de erro que sai ao tentarmos compilar 'main()' usando a seguinte implantação do construtor de 'Handler':

```
Handler(Container const &container)
:
    d_it(container.begin())
{
    size_t x = Container::ios_end;
}
/*
    Erro Reportado:

error: `ios_end' is not a member of type `std::vector<int,
        std::allocator<int> >'
*/
```

Nestes casos, onde a intenção é se referir a um tipo definido numa classe formal como

'Container', isto necessita estar explicitamente indicado ao compilador, usando a palavra chave 'typename'. Aqui está a classe 'Handler' agora usando 'typename':

```
template <typename Container>
class Handler
{
    typename Container::const_iterator d_it;

public:
    Handler(Container const &container)
    :
        d_it(container.begin())
    {}
};
```

Agora 'main()' compilará corretamente.

19.1.4: Parâmetros sem tipificação

Como vimos com os modelos de funções, os modelos de parâmetros são modelos de tipos de parâmetros ou modelos de parâmetros sem tipificação. Os modelos de classes também definem parâmetros sem tipificação. Como os parâmetros não constantes usados com os modelos de funções, precisam ser constantes e seus valores conhecidos no momento de instanciação de um objeto.

Contudo, seus valores não são deduzidos pelo compilador usando argumentos passados aos construtores. Assuma que modificamos o modelo da classe 'FBB::auto_ptr' para que tenha um parâmetro adicional sem tipificação 'size_t Size'. Em seguida usamos este parâmetro 'Size' num novo construtor que define um conjunto de elementos 'Size' do tipo 'Data' como seu parâmetro. O novo modelo da classe 'FBB::auto_ptr' fica (mostrando só os construtores relevantes, note os dois modelos de parâmetros tipificados que agora são requeridos, p.ex., quando especificamos o tipo do parâmetro do construtor de cópias):

```
namespace FBB
{
    template <typename Data, size_t Size>
    class auto_ptr
    {
        Data *d_data;
        size_t d_n;

public:
        auto_ptr(auto_ptr<Data, Size> &other);
        auto_ptr(Data2 *data);
        auto_ptr(Data const (&arr)[Size]);
        ...
    }
```

```

};

template <typename Data, size_t Size>
inline auto_ptr<Data, Size>::auto_ptr(Data const (&arr)[Size])
:
    d_data(new Data2[Size]),
    d_n(Size)
{
    std::copy(arr, arr + Size, d_data);
}
}

```

Desafortunadamente esta nova configuração não satisfaz nossas necessidades, já que os valores do modelo do parâmetro sem tipificação não é deduzido pelo compilador. Quando o compilador é interrogado para compilar a seguinte função 'main()' reporta incompatibilidade entre o número de parâmetros passado e o do modelo:

```

int main()
{
    int arr[30];

    FBB::auto_ptr<int> ap(arr);
}
/*
Erro reportado pelo compilador:

In function `int main()':
    error: wrong number of template arguments (1, should be 2)
    error: provided for `template<class Data, size_t Size>
        class FBB::auto_ptr'
*/

```

Fazendo 'Size' um parâmetro sem tipificação e com valor padrão, tão pouco funciona. O compilador usará o padrão, a menos que esteja explicitamente especificado o contrário. assim, deduzindo que 'Size' pode ser 0 a menos que saibamos o contrário, especificamos 'size_t Size = 0' na lista de tipos de parâmetros do modelo causa incompatibilidade entre o valor 0 e o tamanho atual do conjunto 'arr', definido na função 'main()' acima. O compilador, usando o valor padrão, agora reporta:

```

In instantiation of `FBB::auto_ptr<int, 0>':
...
error: creating array with size zero (`0')

```

Assim, apesar de que os modelos de classes podem usar parâmetros sem tipificação, precisam ser especificados como os parâmetros tipificados quando um objeto da classe é definido. Podem ser especificados valores padrão para esses parâmetros sem tipificação, mas então o padrão será usado quando o parâmetro estiver sem especificação.

Pondo a definição do construtor fora da interface, também não funciona. Ainda é igual que

no modelo de função, mesmo porque agora está fora de sua interface de classe. Em particular note que nenhum valor padrão de modelo de parâmetro tipificado ou não pode ser usado quando modelos de funções membro são definidos fora de sua interface. A modelos de funções (e portanto: a modelos de funções membro de classes) não se pode dar valores padrão a modelos de parâmetros tipificados ou não.

Da mesma forma que para os modelos de parâmetros não tipificados das funções, os parâmetros não tipificados dos modelos de classes precisam ser especificados constantes:

- As variáveis globais possuem endereços constantes que podem ser usados como argumentos de parâmetros não tipificados;
- As variáveis locais e alocadas dinamicamente possuem endereços que só são conhecidos pelo compilador quando o arquivo fonte for compilado. Estes endereços, portanto, não podem ser usados como argumentos para parâmetros não tipificados;
- As transformações de 'lvalue' são permitidas: se um ponteiro é definido como parâmetro não tipificado, um nome de conjunto pode ser especificado;
- Conversões de qualificação são permitidas: Um apontador a um objeto não constante pode ser usado com um parâmetro não tipificado definido como ponteiro constante;
- São permitidas promoções: Uma constante de tipo de dados 'mais estreito' pode ser usada para a especificação de um parâmetro não tipificado de um tipo 'mais amplo' (p.ex., um 'short' pode ser usado quando um 'int' é chamado, um long quando é chamado um 'double');
- Conversões inteiras são permitidas: Se um parâmetro é especificado `size_t`, um inteiro também pode ser usado.
- Não é permitido o uso de variáveis para especificar modelos de parâmetros não tipificados, já que seus valores não são expressões constantes. Contudo, variáveis com o modificador 'const' podem ser usadas, já que seus valores nunca mudam.

Embora nossa tentativa seja definir um construtor da classe `FBB::auto_ptr` que aceite um conjunto como seu argumento, permitindo-nos usar o tamanho dos conjuntos no código do construtor falhou, não esgotamos todas opções. Na próxima seção uma solução será descrita que permite-nos atingir esse objetivo por fim.

19.2: Modelos de Membros

Nosso intento anterior de definir um modelo de parâmetro não tipificado que seja iniciado pelo compilador com o número de elementos de um conjunto falhou porque os modelos dos parâmetros não são deduzidos implicitamente quando o construtor é chamado, mas são explicitamente especificados quando um objeto do modelo da classe é definido. Como os parâmetros são especificados justo antes do modelo do construtor ser chamado, não há nada mais que deduzir e o compilador usará os argumentos explicitamente especificados.

Por outro lado, quando os modelos das funções são usados, os modelos dos parâmetros necessários são deduzidos dos argumentos usados na chamada da função. Isto abre uma rota de solução ao nosso problema. Se o construtor é feito num membro que é o modelo de uma função (contendo o anúncio de modelo de si próprio), então o compilador será capaz de deduzir o valor do parâmetro não tipificado.

As funções membro (ou classes) de modelos de classes que elas próprias são modelos são chamadas modelos de membros.

Os modelos de membros são definidos da mesma forma que outros modelos, incluindo o cabeçalho 'template <typename ...>'.
'

Quando convertendo nosso construtor anterior

```
FBB::auto_ptr (Data const (&array) [Size])
```

num modelo de membro usamos o tipo de parâmetro do modelo da classe 'Data', mas devemos fornecer o modelo de um membro com parâmetro não tipificado. A implantação do construtor em linha fica:

```
template <typename Data>
class auto_ptr
{
    ...
public:
    template <size_t Size>
    auto_ptr(Data const (&arr) [Size]);
    ...
};
```

and the constructor's implementation becomes:

```
template <typename Data>
template <size_t Size>
inline auto_ptr<Data>::auto_ptr(Data const (&arr) [Size])
:
```



```

        d_data(new Data[Size]),
        d_n(Size)
    {
        std::copy(arr, arr + Size, d_data);
    }

```

Os modelos de membros têm as seguintes características:

- Normas usuais de acesso: O construtor pode ser usado pelo programa geral para construir um objeto 'FBB::auto_ptr' de um tipo dado de dados. Como é usual para modelos de classes, o tipo de dados deve ser especificado quando o objeto é construído. Para construir um objeto 'FBB::auto_ptr' de um conjunto 'int array[30]' definimos:

```
FBB::auto_ptr<int> object(array);
```

- Qualquer membro pode ser definido como o modelo de um membro, não só o construtor;
- Os modelos de membros podem ser definidos em linha ou fora de suas classes. Quando um membro é definido abaixo da classe, a lista de modelos de parâmetros da classe deve preceder a lista de modelos de parâmetros da função do modelo do membro. Por exemplo:

```

template <typename Data>
template <unsigned Size>
auto_ptr<Data>::auto_ptr(Data const (&arr)[Size])
:
    d_data(new Data[Size]),
    d_n(Size)
{
    std::cout << "fixed array\n";
    std::copy(arr, arr + Size, d_data);
}

```

Quando definindo o modelo de um membro abaixo de sua classe;

- A definição pode ser dada dentro de seu espaço nomeado. A organização dos arquivos que definem modelos de classes dentro de um espaço nomeado, portanto, deve ser:

```

namespace SomeName
{
    template <typename Type, ...>    // definição do modelo da classe
    class ClassName
    {
        ...
    };

    template <typename Type, ...>    // definição(s) de membro não em linha
    ClassName<Type, ...>::member(...)
    {
        ...
    }
}

```

```

    }
} // fecha o espaço nomeado

```

- Dois anúncios de modelo têm que ser usados: O da classe é especificado antes, seguido pelo anúncio de modelo do membro;
- A definição deve especificar o escopo próprio de modelo de membro: O modelo de membro é definido como membro da classe 'FBB::auto_ptr', instanciada para o tipo formal de modelo de parâmetro 'Data'. Como já estamos dentro do espaço nomeado 'FBB', o cabeçalho da função começa com 'auto_ptr<Data>::auto_ptr';
- Quando são definidos modelos de membros abaixo de suas classes, o nome do modelo formal do parâmetro na declaração e sua implantação têm que ser idênticos.

Um pequeno problema persiste. Quando construímos um objeto 'FBB::auto_ptr' de um conjunto com tamanho fixo o construtor acima não é usado. Em seu lugar o construtor:

```
FBB::auto_ptr<Data>::auto_ptr(Data *data)
```

é ativado. Como o construtor anterior não é um modelo de membro, é considerado uma versão mais especializada de um construtor da classe 'FBB::auto_ptr' que o construtor precedente. Como ambos construtores aceitam um conjunto, o compilador chamará 'auto_ptr(Data *)' em vez de 'auto_ptr(Data const (&array [Size]))'. Este pequeno problema pode ser resolvido simplesmente fazendo o construtor 'auto_ptr(Data *data)' um modelo de membro também. A única sutileza restante é que os modelos de parâmetros dos modelos de membros não podem fazer sombra para os modelos de parâmetros de suas classes. Pondo 'Data2' no lugar de 'Data' resolve esta sutileza. Eis a definição (em linha) do construtor 'auto_ptr(Data *)', seguido por um exemplo onde os dois construtores são usados:

```

template <typename Data2>
auto_ptr(Data2 *data) // tem que ser alocada dinamicamente
:
    d_data(data),
    d_n(0)
{}

```

Chamando ambos construtores em 'main()':

```

int main()
{
    int array[30];

    FBB::auto_ptr<int> ap(array);
    FBB::auto_ptr<int> ap2(new int[30]);

    return 0;
}

```

19.3: Membros de dados estáticos

Quando são definidos membros estáticos num modelo de classe, eles são instanciados a cada nova instanciação. Como são membros estáticos, existirá só um membros quando muitos objetos do mesmo tipo(s) de modelo são definidos. Por exemplo, numa classe como:

```
template <typename Type>
class TheClass
{
    static int s_objectCounter;
};
```

Haverá um 'TheClass<Type>::objectCounter' para cada 'Type' diferente de especificação. Contudo, o seguinte instancia só uma variável estática, compartilhada entre os diversos objetos:

```
TheClass<int> theClassOne;
TheClass<int> theClassTwo;
```

Mencionar membros estáticos nas interfaces não significa que esses membros foram definidos: Foram somente declarados por suas classes e devem ser definidos separadamente. Com os membros estáticos dos modelos de classes isto não é diferente. A definição de membros estáticos são postas imediatamente seguindo (i.e., abaixo) da interface do modelo da classe. O membro estático 's_objectCounter' será definido como segue:

```
template <typename Type>                // definição, seguindo
int TheClass<Type>::s_objectCounter = 0; // a interface
```

No caso acima 'objectCounter' é um inteiro e independe do tipo do modelo do parâmetro 'Type'.

Numa construção como lista, onde um ponteiro a objetos de uma classe é requerido, o tipo do modelo de parâmetro 'Type' precisa ser usado para definir a variável estática, como mostra o seguinte exemplo:

```
template <typename Type>
class TheClass
{
    static TheClass *s_objectPtr;
};

template <typename Type>
TheClass<Type> *TheClass<Type>::s_objectPtr = 0;
```

Como usual, a definição pode ser lida do nome da variável de trás para frente da definição: 'objectPtr' da classe 'TheClass<Type>' é um ponteiro a um objeto de 'TheClass<Type>'.

Finalmente, quando é definida uma variável estática do tipo do modelo do parâmetro, pode

não ser dado o valor inicial 0, claro. O construtor padrão é (p.ex., 'Type()') usualmente é mais apropriado):

```
template <typename Type>                                // definição de s_type's
Type TheClass<Type>::s_type = Type();
```

19.4: Especializando modelos de classes para desviação de tipos

Nossa classe anterior 'FBB::auto_ptr' pode ser usada para muitos tipos diferentes. Sua característica comum é que podem ser adjudicados ao membro da classe 'd_data', p.ex., usando 'using auto_ptr(Data *data)'. Contudo, nem sempre isto é tão simples como parece. Que passa se 'Data' for 'char *'? Exemplos de 'char**', como tipo resultante de dados, são bem conhecidos: Os 'main()' 'argv' e 'envp', por exemplo são parâmetros 'char **'.

Neste caso especial não estamos simplesmente interessados na re-adjudicação do parâmetro do construtor 'd_data' para a classe, mas copiar a estrutura completa 'char **'. Para isto, pode ser usada a especialização do modelo de classe.

Especializações de modelos de classes são usados nos casos onde os modelos de funções não podem (ou não devem) ser usados para um tipo especial de modelo de parâmetro. Nesses casos modelos de membros especializados podem ser construídos, dadas as necessidades do tipo em vista.

As especializações de membros da classe são especializações de membros existentes. Como os membros da classe existe, as especializações não serão parte da interface de classe. São definidas abaixo da interface como membros, re-definindo os membros mais genéricos usando tipos explícitos. Ainda mais, como são especializações de membros da classe existentes, os protótipos das funções devem corresponder exatamente aos protótipos das funções membro das quais são especializações. Para nosso 'Data = char *' uma especialização poderia ser:

```
template <>
auto_ptr<char *>::auto_ptr(char **argv)
:
    d_n(0)
{
    char **tmp = argv;
    while (*tmp++)
        d_n++;
    d_data = new char *[d_n];

    for (size_t idx = 0; idx < d_n; idx++)
    {
        std::string str(argv[idx]);
        d_data[idx] =
            strcpy(new char[str.length() + 1], str.c_str());
    }
}
```

```
}
```

Agora, a especialização acima será usada para construir o seguinte objeto 'FBB::auto_ptr':

```
int main(int argc, char **argv)
{
    FBB::auto_ptr<char *> ap3(argv);
    return 0;
}
```

Apesar de que definir uma especialização do modelo do membro pode nos permitir o uso do tipo excepcional, também é bem possível que uma simples especialização do modelo do membro não seja suficiente. É o caso quando projetamos a especialização 'char *', já que a implantação do modelo de 'destroy()' não está correta para o tipo especial 'Data = char *'. Quando múltiplos membros precisam ser especializados para um determinado tipo, então um modelo de classe inteiro deve ser considerado.

Uma classe completamente especializada tem as seguintes características:

- A especialização do modelo da classe segue a definição genérica do modelo da classe;
- A todos os modelos de parâmetros da classe são dados nomes de tipos específicos ou (para parâmetros sem tipificação) valores específicos. Estes valores específicos são colocados explicitamente numa lista de especificação de modelos de parâmetros (fechada em parênteses angulares) posta imediatamente em seguida ao nome do modelo da classe;
- Todos os modelos especializados dos membros especificam os tipos e valores especializados onde os modelos de parâmetros genéricos são usados na definição do modelo genérico;
- Não todos os modelos de membros têm que ser definidos, mas, para assegurar generalidade da especialização, devem ser definidos. Se um membro é deixado de lado da especialização, não pode ser usado para um tipo(s) especializado;
- Membros adicionais podem ser definidos na especialização. Contudo, aqueles definidos no modelo genérico também devem ter membros correspondentes (usando o mesmo protótipo, apesar de usar os modelos de parâmetros genéricos) na definição genérica do modelo da classe. O compilador não protestará quando são definidos membros adicionais e permitirá o uso desses membros com objetos do modelo especializado da classe;
- As funções membro dos modelos especializados das classes podem ser definidos em suas especializações de classe ou podem ser declarados na especialização da classe. Quando são só declarados então suas definições devem ser dadas abaixo da interface da classe especializada. Tal implantação pode não começar com o anúncio 'template <>', mas deve começar imediatamente

com o cabeçalho da função membro.

Abaixo é dada uma especialização completa da classe 'FBB::auto_ptr' para o tipo 'Data = char *', ilustrando as características acima dadas. As especializações podem ser acrescentadas no arquivo que já contenha o modelo genérico da classe. Para reduzir o tamanho do exemplo os membros que só são declarados podem ser assumidos com implantação idêntica à usada no modelo genérico:

```
#include <iostream>
#include <algorithm>
#include "autoptr1.h"

namespace FBB
{
    template<>
    class auto_ptr<char *>
    {
        char **d_data;
        size_t d_n;

    public:
        auto_ptr<char *>();
        auto_ptr<char *>(auto_ptr<char *> &other);
        auto_ptr<char *>(char **argv);

        // template <size_t Size>          NI
        // auto_ptr(char *const (&arr)[Size])

        ~auto_ptr();
        auto_ptr<char *> &operator=(auto_ptr<char *> &rvalue);
        char *&operator[](size_t index);
        char *const &operator[](size_t index) const;
        char **get();
        char *const *get() const;
        char **release();
        void reset(char **argv);
        void additional() const;    // just an additional public
                                    // member

    private:
        void full_copy(char **argv);
        void copy(auto_ptr<char *> &other);
        void destroy();
    };

    inline auto_ptr<char *>::auto_ptr()
    :
        d_data(0),
        d_n(0)
    {}

    inline auto_ptr<char *>::auto_ptr(auto_ptr<char *> &other)
```

```

{
    copy(other);
}

inline auto_ptr<char *>::auto_ptr(char **argv)
{
    full_copy(argv);
}

inline auto_ptr<char *>::~~auto_ptr()
{
    destroy();
}

inline void auto_ptr<char *>::reset(char **argv)
{
    destroy();
    full_copy(argv);
}

inline void auto_ptr<char *>::additional() const
{}

inline void auto_ptr<char *>::full_copy(char **argv)
{
    d_n = 0;
    char **tmp = argv;
    while (*tmp++)
        d_n++;
    d_data = new char *[d_n];

    for (size_t idx = 0; idx < d_n; idx++)
    {
        std::string str(argv[idx]);
        d_data[idx] =
            strcpy(new char[str.length() + 1], str.c_str());
    }
}

inline void auto_ptr<char *>::destroy()
{
    while (d_n--)
        delete d_data[d_n];
    delete[] d_data;
}
}

```

19.5: Especializações Parciais

Na seção anterior vimos que é possível projetar modelos especializados de classes. Foi mostrado que ambos, modelos de membros de classes e que os modelos inteiros de classes podem ser especializados. Ainda mais, vimos especializações para tipos de modelos de parâmetros.

Nesta seção introduziremos uma variante nas especializações, em número e tipos de modelos de parâmetros que são especializados. Especializações parciais podem ser definidas para modelos de classes com muitos modelos de parâmetros. Com a especialização parcial a um subconjunto (qualquer subconjunto) de tipos de modelos de parâmetros é dado valores específicos.

Tendo discutido as especializações de tipos de modelos de parâmetros na seção anterior, discutiremos especializações de parâmetros sem tipificação na seção atual. As especializações parciais de modelos com parâmetros sem tipificação serão ilustradas usando alguns conceitos simples definidos na álgebra de matrizes, um ramo da álgebra linear.

Uma matriz, comumente falando, consiste numa tabela com certo número de linhas e colunas, preenchida com números. Reconhecemos imediatamente uma abertura para usar modelos: Os números podem ser valores de duplos plenos, mas podemos ter também, muito bem, números complexos, para os quais nosso recipiente complexo (veja seção 12.4) é útil. Assim, nosso modelo de classe pode se beneficiar, com nome 'DataType', para a qual uma classe concreta pode ser especificada quando uma matriz é construída. Algumas matrizes simples, com valores duplos são:

```
1    0    0          Uma matriz identidade,
0    1    0          uma matriz 3 x 3.
0    0    1

1.2  0    0    0      Uma matriz retangular,
0.5  3.5  18  23      uma matriz 2 x 4.

1    2    4    8      Uma matriz de uma linha,
                        uma matriz 1 x 4, também conhecida como
                        `vetor linha' de 4 elementos.
                        (vetores coluna são definidos analogamente)
```

Como as matrizes são constituídas de linhas e colunas (as dimensões da matriz), que normalmente não muda enquanto usamos as matrizes, podemos especificar seu valor como modelo de parâmetros sem tipificação. Enquanto a seleção 'DataType = double' será usado na maioria dos casos, assim, duplo pode ser selecionado como o tipo padrão do modelo. Como há um padrão sensível, o modelo do tipo do parâmetro 'DataType' é posto por último na lista de tipos de modelos de parâmetros. Assim, nosso modelo da classe 'Matrix' começa como segue:

```
template <size_t Rows, size_t Columns, typename DataType = double>
```



```
class Matrix
...
```

Sobre as matrizes são definidas várias operações. São, por exemplo adicionadas, subtraídas ou multiplicadas. Aqui não focalizaremos essas operações. Melhor nos concentraremos numa operação simples: computar os marginais e as somas. O marginal de cada linha é obtido pela soma de todos os seus elementos, pondo essas somas de linhas num vetor coluna pondo-os na linha correspondente somada. Analogamente, os marginais das colunas são obtidos somando-se todos os elementos de cada coluna, pondo essas somas de colunas num vetor linha cada um em sua coluna correspondente. Finalmente a soma dos elementos da matriz pode ser computada. A soma, como é óbvio, é igual à soma dos marginais. O seguinte exemplo mostra uma matriz, seus marginais e sua soma:

	matriz:			marginais das	
				linhas:	
	1	2	3	6	
	4	5	6	15	
marginais	5	7	9	21	(soma)
das colunas					

O que queremos que nosso modelo de classe ofereça?

- Necessita um lugar para guardar os elementos da matriz. Pode ser definido como um conjunto 'Rows' de linhas de elementos de 'Columns' do tipo 'DataType'. Pode ser um conjunto, antes que um ponteiro, já que as dimensões da matriz são conhecidas a priori. Um vetor de elementos colunas (uma linha da matriz), bem como um vetor de elementos linhas (uma coluna da matriz) são freqüentemente usados, podem ser considerados 'typedefs'. A seção inicial da interface de classe contém:

```
typedef Matrix<1, Columns, DataType>      MatrixRow;
typedef Matrix<Rows, 1, DataType>        MatrixColumn;

MatrixRow d_matrix[Rows];
```

deve oferecer construtores: Um construtor padrão e, por exemplo, um construtor que inicie a matriz de uma 'stream'. Não é requerido um construtor de cópias, já que o construtor de cópias padrão realiza sua tarefa a contento. Analogamente, não requer operador de adjudicação sobrecarregado ou destrutor. Eis os construtores, definidos na seção pública:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, Columns, DataType>::Matrix()
{
    std::fill(d_matrix, d_matrix + Rows, MatrixRow());
}

template <size_t Rows, size_t Columns, typename DataType>
```

```
Matrix<Rows, Columns, DataType>::Matrix(std::istream &str)
{
    for (size_t row = 0; row < Rows; row++)
        for (size_t col = 0; col < Columns; col++)
            str >> d_matrix[row][col];
}
```

- O membro operador da classe 'operator[]()' (e sua variante 'const') só manipula o primeiro índice, retornando uma referência a uma 'MatrixRow' completa. Para manter a simplicidade do exemplo, não se implantará exame de fronteiras do conjunto:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType>
&Matrix<Rows, Columns, DataType>::operator[](size_t idx)
{
    return d_matrix[idx];
}
```

- Agora chegamos às partes interessantes: O cálculo das marginais e a soma de todos os elementos da matriz. Considerando que os marginais são vetores, ou 'MatrixRow', contendo os marginais colunas, ou 'MatrixColumn', contendo os marginais linhas, ou um simples valor, contendo a soma de um vetor marginal ou o valor de uma matriz 1 x 1, iniciado de uma 'Matrix' genérica, podemos agora construir especializações parciais para manipular as matrizes 1 x 1. Como estamos por definir essas especializações, usadas para calcular os marginais e a soma da matriz. Eis aqui a implantação desses membros:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType>
Matrix<Rows, Columns, DataType>::columnMarginals() const
{
    return MatrixRow(*this);
}
```

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, 1, DataType>
Matrix<Rows, Columns, DataType>::rowMarginals() const
{
    return MatrixColumn(*this);
}
```

```
template <size_t Rows, size_t Columns, typename DataType>
DataType Matrix<Rows, Columns, DataType>::sum() const
{
    return rowMarginals().sum();
}
```

Especializações parciais do modelo de classe pode ser definido como qualquer subconjunto

dos parâmetros do modelo. Podem ser definidos para parâmetros do modelo tipificados e sem tipificação. Nossa primeira especialização parcial define o caso especial onde construímos uma linha de uma matriz genérica, específica para a construção dos marginais colunas (mas não restrita a isto). Aqui está:

- A especialização parcial começa definindo todos tipos de modelo de parâmetros que não são especializados na especialização parcial. Este anúncio de modelo de especialização parcial não pode especificar nenhum padrão (como 'DataType = double'), já que os padrões já foram especificados pela definição geral do modelo da classe. Ainda mais, a especialização precisa seguir a definição geral do modelo da classe ou o compilador se queixará que não sabe que classe está sendo especializada. Seguindo o anúncio de modelo, a interface de classe começa. Como é uma especialização (parcial) do modelo de classe o nome da classe é seguido uma lista de modelos de parâmetros que especifica valores concretos ou tipos para todos os modelos de parâmetros da especialização e usando os nomes dos modelos genéricos dos parâmetros restantes tipificados ou não. Na especialização da 'Row' é especificada como 1, já que estamos aqui falando de uma só linha. Ambas 'Columns' e 'DataType' ficam por ser especificadas. Portanto, a especialização parcial de 'MatrixRow' começa assim:

```
template <size_t Columns, typename DataType> // no default specified
class Matrix<1, Columns, DataType>
```

- Uma 'MatrixRow' contém os dados de uma só linha. Portanto precisamos um membro de dados que guarde os valores de 'Column' do tipo 'DataType'. Como 'Column' é um valor constante, o membro de dados 'd_row' pode ser definido como um conjunto:

```
    DataType d_column[Columns];
```

- Os construtores requerem alguma atenção. O construtor padrão é simples. Só inicia os elementos de dados 'MatrixRow', usando o 'DataType':

```
template <size_t Columns, typename DataType>
Matrix<1, Columns, DataType>::Matrix()
{
    std::fill(d_column, d_column + Columns, DataType());
}
```

Também necessitamos um construtor que inicie um objeto 'MatrixRow' com a coluna marginal de um objeto 'Matrix' genérico. Isto requer fornecer um construtor com parâmetro 'Matrix' não especializado. Em casos como este, a regra prática é definir um modelo de membro que nos permita conservar a natureza genérica do parâmetro. Como o modelo genérico de 'Matrix' requer três parâmetros, dois deles já fornecidos pela especialização do modelo, o terceiro parâmetro precisa ser especificado no anúncio de modelo como modelo de membro. Como este parâmetro se refere ao número genérico de linhas da matriz, chamemo-lo simplesmente de 'Rows'. Eis, então, a definição do segundo construtor, iniciando os dados de 'MatrixRow' com as colunas marginais de um objeto genérico 'Matrix':

```

template <size_t Columns, typename DataType>
template <size_t Rows>
Matrix<1, Columns, DataType>::Matrix(
    Matrix<Rows, Columns, DataType> const &matrix)
{
    std::fill(d_column, d_column + Columns, DataType());

    for (size_t col = 0; col < Columns; col++)
        for (size_t row = 0; row < Rows; row++)
            d_column[col] += matrix[row][col];
}

```

Note a forma como o parâmetro do construtor está definido: É uma referência a um modelo 'Matrix', usando o modelo do parâmetro adicional 'Row' bem como os parâmetros da especialização parcial também.

- Na realidade não necessitamos de membros adicionais para satisfazer nossas necessidades atuais. Para acessar os elementos de 'MatrixRow' um operador sobrecarregado 'operator[]()', claro, é útil. Novamente, a variante 'const' pode ser implantada como a variante não constante. Eis a implantação:

```

template <size_t Columns, typename DataType>
DataType &Matrix<1, Columns, DataType>::operator[](size_t idx)
{
    return d_column[idx];
}

```

Agora que definimos a classe genérica 'Matrix' bem como sua especialização parcial, definindo uma simples linha, o compilador selecionará a especialização da linha sempre que uma 'Matrix' for definida usando 'Row = 1'. Por exemplo:

```

Matrix<4, 6> matrix;           // é usado o modelo genérico de Matrix
Matrix<1, 6> row;              // a especialização parcial é usada

```

A especialização parcial para 'MatrixColumn' é construída de maneira similar. Apresentamos suas linhas gerais (a definição do modelo completo da classe 'Matrix' bem como todas suas especializações são fornecidos no arquivo 'cplusplus.yo.zip' (em ftp.rug.nl no arquivo 'yo/templateclasses/examples/matrix.h')):

- A especialização parcial do modelo de classe novamente começa pelo anúncio de modelo. A definição de classe específica, agora, um valor fixo para o segundo modelo (genérico) de parâmetro, ilustrando que podemos construir especializações parciais para todo modelo de parâmetro; não só o primeiro ou o último:

```

template <size_t Rows, typename DataType>
class Matrix<Rows, 1, DataType>

```

- Seus construtores são implantados analogamente à forma como o foram os construtores de

'MatrixRow'. Suas implantações são deixadas como exercício ao leitor (e podem ser encontradas em 'matrix.h').

- Um membro adicional 'sum()' é definido para calcular a soma dos elementos do vetor 'MatrixColumn'. Sua implantação é feita simplesmente usando o algoritmo genérico 'accumulate()':

```
template <size_t Rows, typename DataType>
DataType Matrix<Rows, 1, DataType>::sum()
{
    return std::accumulate(d_row, d_row + Rows, DataType());
}
```

O leitor pode achar interessante saber o que acontece se especificarmos a seguinte matriz:

```
Matrix<1, 1> cell;
```

Esta é uma 'MatrixRow' ou uma 'MatrixColumn'? A resposta é: É ambígua, precisamente porque ambas as colunas e as linhas podem ser usadas com uma especialização parcial (diferentes). Se tal matriz é requerida, outra especialização parcial do modelo precisa ser desenvolvida. Como essa especialização do modelo pode ser útil para obter a soma dos elementos de uma 'Matrix', também está aqui coberta;

- Esta especialização parcial do modelo de classe também necessita um anúncio de modelo, esta vez só especificando 'DataType'. A definição de classe especifica dois valores fixos, usando 1 para ambos, o número de linhas e o número de colunas:

```
template <typename DataType>
class Matrix<1, 1, DataType>
```

- A especialização define a tarefa usual dos construtores. Novamente construtores que esperam um tipo mais genérico de 'Matrix' são implantados como modelos de membros. Por exemplo:

```
template <typename DataType>
template <size_t Rows, size_t Columns>
Matrix<1, 1, DataType>::Matrix(
    Matrix<Rows, Columns, DataType> const &matrix)
:
    d_cell(matrix.rowMarginals().sum())
{}

template <typename DataType>
template <size_t Rows>
Matrix<1, 1, DataType>::Matrix(Matrix<Rows, 1, DataType> const &matrix)
:
    d_cell(matrix.sum())
{}

```

- Como 'Matrix<1, 1>' é basicamente um envoltório do valor de 'DataType', necessitamos de

membros para aceder esse vaor. Um operador de conversão de tipopode ser útil, mas também necessitamos um membro 'get()' para obter o valor se o operador de conversão não for usado pelo compilador (que acontece quando é dada a escolha ao compilador, veja seção 9.3). Aqui está os acessores (deixando fora sua variante constante):

```
template <typename DataType>
Matrix<1, 1, DataType>::operator DataType &()
{
    return d_cell;
}

template <typename DataType>
DataType &Matrix<1, 1, DataType>::get()
{
    return d_cell;
}
```

A seguinte função 'main()' mostra como o modelo da classe 'Matrix' e suas especializações parciais podem ser usadas:

```
#include <iostream>
#include "matrix.h"
using namespace std;

int main(int argc, char **argv)
{
    Matrix<3, 2> matrix(cin);

    Matrix<1, 2> colMargins(matrix);
    cout << "Column marginals:\n";
    cout << colMargins[0] << " " << colMargins[1] << endl;

    Matrix<3, 1> rowMargins(matrix);
    cout << "Row marginals:\n";
    for (size_t idx = 0; idx < 3; idx++)
        cout << rowMargins[idx] << endl;

    cout << "Sum total: " << Matrix<1, 1>(matrix) << endl;
    return 0;
}
/*
Saída Gerada da entrada: 1 2 3 4 5 6

Colunas marginais:
9 12
Linhas marginais:
3
7
11
Soma total: 21
*/
```

19.6: Instanciando modelos de classes

Os modelos de classes são instanciados quando um objeto do modelo da classe é definido. Quando um objeto do modelo de classe é definido ou declarado os modelos dos parâmetros precisam ser especificados explicitamente.

Os modelos dos parâmetros também são especificados quando um modelo de classe define valores padrão para o modelo do parâmetro, apesar de que o compilador, nesses casos, fornece os padrões. Os valores ou tipos dos modelos de parâmetros nunca são deduzidos, como com os modelos de funções: Para definir uma 'Matrix' de números complexos, o seguinte construtor é usado:

```
Matrix<3, 5, std::complex> complexMatrix;
```

Enquanto a seguinte construção define uma matriz de elementos com valores duplos, com o compilador fornecendo o tipo (padrão) duplo:

```
Matrix<3, 5> doubleMatrix;
```

Um objeto de um modelo de classe pode ser declarado usando a palavra chave externo (extern).

Por exemplo, a seguinte construção é usada para declarar a matriz 'complexMatrix':

```
extern Matrix<3, 5, std::complex> complexMatrix;
```

Uma declaração de modelo de classe é suficiente se as declarações das funções são compiladas de funções que possuam valores de retorno ou parâmetros que são objetos do modelo da classe, apontadores ou referências. Este pequeno arquivo fonte que segue pode ser compilado, apesar de que o compilador não viu a definição do modelo da classe 'Matrix'. Note que classes genéricas bem como especializações (parciais) podem ser declaradas. Ainda mais, note que uma função que espera ou que retorne um objeto do modelo de classe, referência ou parâmetro automaticamente se torna um modelo de função, pois é necessário para especificar a natureza dos vários modelos de parâmetros:

```
template <size_t Rows, size_t Columns, typename DataType = double>
class Matrix;

template < size_t Columns, typename DataType>
class Matrix<1, Columns, DataType>;

template<unsigned Columns, typename DataType>
Matrix<1, 12> *function(Matrix<2, 18, size_t> &mat);
```

Quando modelos de classes são usados, devem ser processados pelo compilador primeiro. Assim, os modelos das funções membro precisam ser conhecidos pelo compilador quando o modelo é

instanciado. Isto não significa que todos os membros do modelo da classe são instanciados quando um objeto do modelo da classe é definido.

O compilador só instanciará aqueles membros usados. Isto é ilustrado pela seguinte classe simples 'Demo', que tem dois construtores e dois membros. Quando criamos uma função 'main()' onde um construtor é usado e um membro é chamado, podemos fazer uma nota dos tamanhos do arquivo objeto resultante e programa executável. Em seguida a definição da classe é modificada de tal maneira que o construtor e membro não usados fiquem como comentário. Novamente compilamos e linkamos a função 'main()' e os tamanhos resultantes são idênticos dos tamanhos obtidos antes (no meu computador, usando g++ versão 3.3.3 esses tamanhos são 2380 e 13504, respectivamente). Existem outros modos de ilustrar o fato de que somente os membros usados são instanciados, como usar o programa 'nm', que mostra o conteúdo simbólico dos arquivos objetos. Mas usando o programa 'nm' chegaremos à mesma conclusão: Somente os modelos das funções membro que são usadas são instanciadas. Aqui está um exemplo do modelo da classe 'Demo' usada para este pequeno experimento. Em 'main()' o primeiro construtor e a primeira função membro são chamados:

```
#include <iostream>

template <typename Type>
class Demo
{
    Type d_data;

public:
    Demo()
    :
        d_data(Type())
    {}
    void member1()
    {
        d_data += d_data;
    }

    // os seguintes membros são comentados antes da compilação
    // do segundo programa

    Demo(Type const &value)
    :
        d_data(value)
    {}
    void member2(Type const &value)
    {
        d_data += value;
    }
};

int main()
```



```

{
    Demo<int> demo;
    demo.member1();
}

```

19.7: Processando modelos de classes e instâncias

Na seção 18.9 estabelecemos a distinção entre código dependente dos modelos de parâmetros do código independente. A mesma distinção também é válida quando definimos e usamos modelos de classes.

O código que não depende dos modelos dos parâmetros é verificado pelo compilador quando o modelo é definido. P.ex., Se uma função membro num modelo de classe usa a função 'qsort()', então 'qsort()' não depende de um modelo de parâmetro. Conseqüentemente 'qsort()' deve ser conhecida pelo compilador quando a encontra numa chamada. Na prática isto implica que 'cstdlib' ou 'stdlib.h' deve ter sido processada pelo compilador antes de estar habilitado a processar a definição do modelo da classe.

Por outro lado, se um modelo define um tipo de parâmetro '<typename Type>', que é o tipo de retorno de algum modelo de função membro, p.ex:

```
Type member() ...
```

Então distinguimos as seguintes situações, onde o compilador encontra que 'member()' pertence:

- No ponto onde os objetos dos modelos de classe são definidos, o ponto de instanciação dos objetos do modelo de classe, o compilador terá lido a definição do modelo da classe, realizando um exame básico da sintaxe dos membros da função como 'member()'. Assim não aceitará uma definição ou declaração como 'Type &&member()', porque a C++ não suporta referências a referências. Ainda mais, examinará a existência do 'typename' usado para instanciar o objeto. Este 'typename' deve ser conhecido pelo compilador no ponto de instanciação do objeto;
- No ponto onde são usados os modelos das funções membro, também dito ponto de instanciação dos modelos das funções membro, o 'Type' do parâmetro é claro que tem que ser conhecido e os comandos de 'member()' que dependem do 'Type' do modelo do parâmetro são agora examinados sintaticamente. Por exemplo, se 'member()' possui um comando como

```
Type tmp(Type(), 15);
```

então este é, em princípio um comando sintaticamente válido. Contudo, quando 'Type = int' e é chamado 'member()', sua instanciação falhará, porque 'int' não tem um construtor que espere dois argumentos inteiros. Note que isto não é um problema quando o compilador instancia um objeto

da classe contendo 'member()': No ponto de instanciação do objeto sua função membro 'member()' não está instanciada e assim, sua construção inválida 'int' resta não detetada.

19.8: Declarando friends

As funções 'friend' normalmente são construídas como funções de suporte a uma classe que não pode contê-la como membro. O operador, bem conhecido, de inserção das 'streams' de saída é um caso destes. As classes 'friend' são mais freqüentes no contexto de classes aninhadas, onde a classe interior declara a classe exterior como sua amiga (ou outra maneira semelhante). Aqui outra vez vemos um mecanismo de suporte: A classe interior é construída para suportar a classe exterior.

Como as classes concretas, os modelos de classe podem declarar outras funções e classes como suas amigas. Inversamente, as classe concretas podem declarar modelos de classes como suas amigas. Aqui também, a amiga é construída como uma função ou classe especial para aumentar ou suportar a funcionalidade da classe que declara. Contudo a palavra chave 'friend' pode, portanto, ser usada em qualquer tipo de classe (concreta ou modelo) para declarar qualquer tipo de função ou classe como amiga, quando usando modelos de classes os seguintes casos podem ser distinguidos:

- Um modelo de classe pode declarar uma classe ou função concreta como sendo sua amiga. Esta é uma declaração comum de amiga, tal como o operador de inserção para objetos 'ostream';
- Um modelo de classe pode declarar outro modelo de classe ou função como sua amiga. Neste caso, os modelos de parâmetros amigos têm que ser especificados. Se os valores dos modelos de parâmetros amigos precisam ser iguais aos modelos de parâmetros da classe que declara o amigo, o amigo se diz ser um amigo limite do modelo de classe ou função. Neste caso os modelos dos parâmetros do modelo onde uma declaração de amigo é usada determina (liga) os modelos dos parâmetros do modelo da classe ou função, resultando numa correspondência um a um entre os modelos de parâmetros e os modelos de parâmetros da amiga;
- No caso mais geral, um modelo de classe pode declarar outro modelo de classe ou função para ser seu amigo, sem o respectivo do amigo modelo de parâmetros. Neste caso um modelo de classe ou função amigo não limite é declarado: Os modelos dos parâmetros do modelo da classe ou função ficam por ser especificados e não são relativos a algum modelo de parâmetros predefinidos da classe declarante. Por exemplo, se uma classe tem membros de dados de vários tipos, especificados pelos modelos de seus parâmetros e a outra classe é permitido acesso direto a esses membros de dados (assim podendo ser uma amiga), poderíamos especificar qualquer dos modelos de parâmetros a instanciar tal amiga. Melhor dito, especificar múltiplos amigos limite, um só amigo (não limite) pode ser declarado, especificando como amigos os modelos de

parâmetros somente quando for requerido.

- Os casos acima, onde um modelo é declarado como amigo, também pode ser encontrado quando classes concretas são usadas. A classe concreta que declara uma amiga concreta já foi visto (Capítulo 11). O equivalente a amigos limite ocorre se uma classe concreta especifica modelos de parâmetros na declaração de amigo. O equivalente de amigos não limites ocorre se uma classe concreta declare um modelo genérico como amigo.

19.8.1: Funções sem modelo ou classes como amigas

Um modelo de classe pode declarar uma função concreta, uma função membro concreta ou uma classe concreta como amiga. Tal amiga pode acessar os membros privados do modelo de classe

Classes concretas e funções ordinárias podem ser declaradas amigas, mas antes uma simples função membro da classe pode ser declarada amiga, o compilador deve ter visto a interface de classe que declara esse membro. Consideremos as várias possibilidades:

- Um modelo de classe pode declarar uma função concreta como amiga. Não está completamente claro porque declararíamos uma função concreta como amiga. Em classes ordinárias desejaríamos passar um objeto da classe declarante à função. Contudo, isto requer fornecer à função um modelo de parâmetro sem especificar seu tipo. Como a linguagem não suporta construções como:

```
void function(std::vector<Type> &vector)
```

a menos que 'function()' seja um modelo, não está imediatamente claro como e porque tal amigo deveria ser construído. Uma razão é permitir à função acessar a membros estáticos privados da classe. Ainda mais, tais amigos poderiam instanciar objetos da classe declarando-os como amigos e acessar diretamente tais objetos dos membros privados. Por exemplo:

```
template <typename Type>
class Storage
{
    friend void basic();
    static size_t s_time;
    std::vector<Type> d_data;
public:
    Storage()
    {}
};
template <typename Type>
size_t Storage<Type>::s_time = 0;
```

```

void basic()
{
    Storage<int>::s_time = time(0);
    Storage<double> storage;
    std::random_shuffle(storage.d_data.begin(), storage.d_data.end());
}

```

- Declarar uma classe concreta como sendo amiga de um modelo de classe talvez tenha mais sentido prático. Aqui a classe amiga pode instanciar qualquer tipo de objeto do modelo de classe, para acessar todos seus membros privados depois. Uma simples declaração da classe amiga na frente da definição do modelo de classe é suficiente para fazer isto funcionar:

```

class Friend;

template <typename Type>
class Composer
{
    friend class Friend;
    std::vector<Type> d_data;
public:
    Composer();
};

class Friend
{
    Composer<int> d_ints;
public:
    Friend(std::istream &input)
    {
        std::copy(std::istream_iterator<int>(input),
                  std::istream_iterator<int>(),
                  back_inserter(d_ints.d_data));
    }
};

```

- Alternativamente, só uma única função membro de uma classe concreta pode ser declarada amiga. Isto requer que o compilador tenha lido a interface de classe antes da declaração de amiga. Omitindo o destrutor requerido e os operadores sobrecarregados de adjudicação, a seguir é mostrado um exemplo de uma classe cujo membro 'randomizer()' é declarado como amigo da classe 'Composer':

```

template <typename Type>
class Composer;

class Friend
{
    Composer<int> *d_ints;
public:
    Friend(std::istream &input);
}

```

```

        void randomizer();
};

template <typename Type>
class Composer
{
    friend void Friend::randomizer();
    std::vector<Type> d_data;
public:
    Composer(std::istream &input)
    {
        std::copy(std::istream_iterator<int>(input),
                  std::istream_iterator<int>(),
                  back_inserter(d_data));
    }
};

Friend::Friend(std::istream &input)
:
    d_ints(new Composer<int>(input))
{}

void Friend::randomizer()
{
    std::random_shuffle(d_ints->d_data.begin(), d_ints->d_data.end());
}

```

Neste exemplo note que 'Friend::d_ints' é um apontador. Não pode ser 'Composer<int> object', pois a interface da classe 'Composer' ainda não foi vista pelo compilador quando lê a interface da classe 'Friend'. Não pondo atenção a este fato e definindo um membro de dados 'Composer<int> d_ints' resulta num erro de compilação:

```
error: field `d_ints' has incomplete type
```

Tipo incompleto, já que o compilador, neste ponto, sabe da existência da classe 'Composer' mas não viu a interface de 'Composer' e portanto não sabe que tamanho o membro de dados 'd_ints' terá.

19.8.2: Modelos instanciados para tipos específicos como amigos

Com modelos de classes amigas limite ou funções onde há uma correspondência um a um entre os valores dos modelos de parâmetros dos modelos amigos e os modelos dos parâmetros dos modelos das classes declarantes dos amigos, os amigos são modelos também. Aqui existem várias possibilidades:

- O modelo de uma função pode ser declarado como amigo de um modelo de classe. Neste caso não há os problemas encontrados com funções concretas declaradas amigas de modelos de

classes. Já que o modelo de função amiga é um modelo, pode ser provista com o modelo do parâmetro requerido permitindo-lhe especificar um parâmetro de um modelo de classe. Assim podemos passar um objeto da classe declarante para o modelo da função. A organização das várias declarações ficam:

- O modelo da classe declarante é também declarada;
- O modelo da função (a ser declarada como amiga) é declarada;
- O modelo da classe declarante do modelo da função limite como amiga é definido;
- O modelo da função (amiga) é definido agora, tendo acesso a todos os membros (privados) do modelo da classe.

Note que a declaração de modelo amigo especifica seus modelos de parâmetros imediatamente depois do nome do modelo da função. Sem a lista de modelos de parâmetros após o nome da função, seria uma função amiga ordinária. Aqui está um exemplo mostrando o uso de um amigo limite para criar subconjuntos de entradas num dicionário. Para exemplos da vida real um objeto de uma função dedicada que retorna '!key1.find(key2)' é provavelmente mais útil, mas no exemplo abaixo 'operator==()' é aceitável:

```
template <typename Key, typename Value>
class Dictionary;

template <typename Key, typename Value>
Dictionary<Key, Value>
    subset(Key const &key, Dictionary<Key, Value> const &dict);

template <typename Key, typename Value>
class Dictionary
{
    friend Dictionary<Key, Value> subset<Key, Value>
        (Key const &key, Dictionary<Key, Value> const &dict);
    std::map<Key, Value> d_dict;
public:
    Dictionary();
};

template <typename Key, typename Value>
Dictionary<Key, Value>
    subset(Key const &key, Dictionary<Key, Value> const &dict)
{
    Dictionary<Key, Value> ret;

    std::remove_copy_if(dict.d_dict.begin(), dict.d_dict.end(),
        std::inserter(ret.d_dict, ret.d_dict.begin()),
```

```

        std::bind2nd(std::equal_to<Key>(), key));
    return ret;
}

```

- Declarando um modelo completo de classe como um modelo de classe amiga, todos os membros da classe amiga podem ser acessar todos os membros privados da classe declarante. Como a classe amiga só tem que ser declarada, a organização da declaração é bem mais fácil que quando modelos de funções são declarados como amigos. No seguinte exemplo uma classe 'Iterator' é declarada como amiga de uma classe 'Dictionary'. Assim, a 'Iterator' é capaz de acessar os dados privados de 'Dictionary'. Existem alguns pontos interessantes a notar aqui:

- Para declarar um modelo de classe como amigo, essa classe é simplesmente declarada como um modelo de classe antes de ser declarada como amiga:

```

template <typename Key, typename Value>
class Iterator;

template <typename Key, typename Value>
class Dictionary
{
    friend class Iterator<Key, Value>;
}

```

- Contudo, a interface da classe amiga já pode ser usada, mesmo antes do compilador ter visto a interface da amiga:

```

Iterator<Key, Value> begin()
{
    // uses the friend's
    return Iterator<Key, Value>(*this); // constructor
}
Iterator<Key, Value> subset(Key const &key)
{
    // uses a member function
    return Iterator<Key, Value>(*this).subset(key);
}

```

- Claro que a interface da amiga deve ainda ser vista pelo compilador. Como é uma classe de suporte para 'Dictionary' pode, com segurança, definir um membro de dados 'std::map', que é iniciado pelo seu construtor, acessando o membro de dados privado 'd_dict' de 'Dictionary':

```

template <typename Key, typename Value>
class Iterator
{
    std::map<Key, Value> &d_dict;

public:
    Iterator(Dictionary<Key, Value> &dict)
    :
        d_dict(dict.d_dict)
    {}
}

```

- O membro de 'Iterator', 'begin()' simplesmente retorna um iterador 'map'. Contudo, como não é sabido pelo compilador como será a instanciação do mapa, um 'map<Key, Value>::iterator' é um nome de tipo (condenado) implícito. Para fazê-lo um tipo de nome explícito simplesmente se prefixa o tipo de nome com o tipo de retorno de 'begin()':

```
typename std::map<Key, Value>::iterator begin()
{
    return d_dict.begin();
}
```

- No exemplo anterior devemos decidir se só um 'Dictionary' estará apto a construir um 'Iterator', já que 'Iterator' está estreitamente ligada a 'Dictionary'. Isto pode ser realizado definindo o construtor de 'Iterator' em sua seção privada e declarando 'Dictionary' como amiga de 'Iterator'. Conseqüentemente, só 'Dictionary' pode criar seu próprio 'Iterator'. Declarando o construtor de 'Iterator' como amigo limite, asseguramos que cria os 'Iterators' usando modelos de parâmetros idênticos aos seus. Eis sua realização:

```
template <typename Key, typename Value>
class Iterator
{
    friend Dictionary<Key, Value>::Dictionary();

    std::map<Key, Value> &d_dict;

    Iterator(Dictionary<Key, Value> &dict)
    :
        d_dict(dict.d_dict)
    {}

    public:
```

Neste exemplo, o construtor de 'Dictionary' é definido como amigo de 'Iterator'. Aqui o amigo é um modelo de membro. Outros membros podem ser declarados como classes amigas também, nesse caso seus protótipos devem ser usados, incluindo os tipos de seus valores de retorno. Assim, assumindo que:

```
std::vector<Value> sortValues()
```

É um membro de 'Dictionary', que retorna um vetor ordenado de seus valores, então a declaração do amigo limite correspondente seria:

```
std::vector<Value> Dictionary<Key, Value>::sortValues()
```

19.8.3: Modelos como amigos sem limite

Quando um amigo é declarado sem limite, só declara um modelo existente como seu amigo, não importa como é instanciado. Isto pode ser útil em situações onde o amigo pode ser capaz de

instanciar objetos de modelos de classes que declaram como amigo, permitindo ao amigo acessar os objetos instanciados por membros privados. Novamente, as funções, as classes e as funções membro podem declara amigos sem limite.

Estas são as convenções sintáticas de declaração de amigos sem limite:

- Declarando um modelo de função como amigo: Qualquer instanciação do modelo da função pode instanciar objetos do modelo da classe e pode acessar seus membros privados. Assuma que o seguinte modelo de função foi definido:

```
template <typename Iterator, typename Class,
          void (Class::*member) (Data &)>
Class &ForEach(Iterator begin, Iterator end, Class &object,
              void (Class::*member) (Data &));
```

Este modelo de função pode ser declarado como um amigo sem limite no seguinte modelo da classe 'Vector2':

```
template <typename Type>
class Vector2: public std::vector<std::vector<Type> >
{
    template <typename Iterator, typename Class, typename Data>
    friend Class &ForEach(Iterator begin, Iterator end, Class &object,
                        void (Class::*member) (Data &));
    ...
};
```

Se o modelo da função for definido dentro de algum espaço nomeado, o espaço nomeado também precisa ser mencionado. P.ex., assumindo que 'ForEach()' está definida no espaço nomeado 'FBB' sua declaração como amiga fica:

```
template <typename Iterator, typename Class, typename Data>
friend Class &FBB::ForEach(Iterator begin, Iterator end, Class &object,
                          void (Class::*member) (Data &));
```

O seguinte exemplo ilustra o uso de um amigo sem limite. A classe 'Vector2' guarda vetores de elementos com modelo de parâmetro 'Type'. Seu membro 'process()' usa 'ForEach()' para chamar seus membros privados 'rows()', que em seu turno usa 'ForEach()' para chamar seus membros privados 'columns()'. Conseqüentemente, 'Vector2' usa duas instanciações de 'ForEach()' e portanto, um amigo sem limite é apropriado aqui. Assume-se que os objetos da classe 'Type' podem ser inseridos em objetos 'ostream' (a definição do modelo da função 'ForEach()' pode ser encontrada no arquivo cplusplus.yo.zip no servidor ftp ftp.rug.nl). Aqui está o programa:

```
template <typename Type>
class Vector2: public std::vector<std::vector<Type> >
{
    template <typename Iterator, typename Class, typename Data>
    friend Class &ForEach(Iterator begin, Iterator end, Class &object,
```

```

        void (Class::*member) (Data &));
public:
    void process()
    {
        ForEach(begin(), end(), *this, &Vector2<Type>::rows);
    }
private:
    void rows(std::vector<Type> &row)
    {
        ForEach(row.begin(), row.end(), *this,
                &Vector2<Type>::columns);

        std::cout << std::endl;
    }
    void columns(Type &str)
    {
        std::cout << str << " ";
    }
};

using namespace std;

int main()
{
    Vector2<string> c;
    c.push_back(vector<string>(3, "Hello"));
    c.push_back(vector<string>(2, "World"));

    c.process();
}
/*
    Saída Gerada:

    Hello Hello Hello
    World World
*/

```

- Analogamente, um modelo completo de classe pode ser declarado como amigo. Isto permite a todas instâncias das funções membro da amiga instanciarem o modelo declarando a classe amiga. Neste caso a classe que declara como amiga pode oferecer funcionalidades úteis a diferentes instâncias (i.e., usando diferentes argumentos para seus modelos de parâmetros) de sua classe amiga. A convenção sintática é comparável à convenção usada na declaração de funções amigas sem limite:

```

template <typename Type>
class PtrVector
{
    template <typename Iterator, typename Class>
    friend class Wrapper;          // classe amiga sem limite
};

```

Todos os membros do modelo da classe 'Wrapper' agora podem instanciar 'PtrVector' usando qualquer tipo de seu modelo de parâmetro 'Type', ao mesmo tempo permitindo instanciações para acessar todos os membros privados de 'PtrVector'.

- Quando só alguns membros de um modelo de classe precisam acessar os membros privados de outro modelo de classe (p.ex., o outro modelo de classe possui construtores privados e só alguns membros do primeiro modelo de classe precisa instanciar objetos do segundo modelo de classe), então o último modelo de classe pode declarar somente esses membros da formação do modelo de classe que requerem acesso a seus membros privados como seus amigos. Novamente, a interface da classe amiga não precisa ser especificada. Contudo, o compilador precisa ser informado de que os membros amigos da classe são em verdade de uma classe. Uma declaração adiantada dessa classe, por isso, deve ser dada também. No exemplo seguinte 'PtrVector' declara 'Wrapper::begin()' como amiga. Note a declaração avançada da classe 'Wrapper':

```
template <typename Iterator>
class Wrapper;

template <typename Type>
class PtrVector
{
    template <typename Iterator> friend
        PtrVector<Type> Wrapper<Iterator>::begin(Iterator const &t1);
    ...
};
```

19.9: Derivação de modelos de classe

Modelos de classes também podem ser usados na derivação de classes também. Quando um modelo de classe é usado na derivação de classe, as seguintes situações podem ser distinguidas:

- Um modelo de classe é usado como a classe de base para a derivação de uma classe concreta. Neste caso a classe resultante ainda é parcialmente um modelo de classe, mas isto é algo abstrato de se ver quando um objeto da classe derivada é construído.
- Um modelo existente de classe é usado como classe de base na derivação de outro modelo de classe. Aqui as características de modelo de classe permanecem claramente visíveis.
- Uma classe concreta é usada como classe de base na derivação de um modelo de classe. Este caso híbrido é interessante para construirmos modelos de classe que são parcialmente pré-compilados.

Estas três variantes de derivação do modelo de classe serão agora elaboradas.

Considere a seguinte classe de base:

```
template<typename T>
class Base
{
    T const &t;

public:
    Base(T const &t)
    :
        t(t)
    {}
};
```

A classe acima é um modelo de classe, que pode ser usada como classe de base para o seguinte modelo da classe 'Derived':

```
template<typename T>
class Derived: public Base<T>
{
public:
    Derived(T const &t)
    :
        Base(t)
    {}
};
```

Outras combinações são possíveis também: Especificando tipos concretos dos modelos dos parâmetros da classe de base, a classe de base é instanciada e a classe derivada é uma classe ordinária (não modelo):

```
class Ordinary: public Base<int>
{
public:
    Ordinary(int x)
    :
        Base(x)
    {}
};

// Com a seguinte definição de objeto:
Ordinary
o(5);
```

Esta construção permite-nos, em situações específicas, agregar funcionalidades a um modelo de classe, sem a necessidade de construir um modelo de classe derivada.

19.9.1: Derivando classes concretas de modelos de classes

Quando um modelo existente de classe é usado como classe de base para derivar uma classe concreta, os parâmetros do modelo de classe são especificados ao se definir a interface da classe derivada. A derivação de uma classe concreta de um modelo de classe pode ser útil se, dentro de um contexto, um modelo existente de classe carece de uma funcionalidade particular. Por exemplo, apesar de que a classe 'map' pode ser usada com facilidade em combinação com o algoritmo genérico 'find_if()' (seção 17.4.16) para localizar um elemento particular, requer a construção de uma classe e pelo menos duas funções objeto a mais dessa classe. Se consideramos isto um alto custo, num contexto particular, estender um modelo de classe com alguma funcionalidade especial pode ser considerado.

Um programa que execute comandos entrados por um teclado que aceite todas as abreviações únicas dos comandos que define. P.ex., o comando 'list' pode ser entrado como l, li, lis ou list. Derivando uma classe 'Handler' de:

```
map<string, void (Handler::*)(string const &cmd)>
```

E definindo um processo ('string const &cmd') para processar os comandos, o programa simplesmente terá que executar a seguinte função 'main()':

```
int main()
{
    string line;
    Handler cmd;

    while (getline(cin, line))
        cmd.process(line);
}
```

A classe 'Handler' é derivada de uma classe mapa complexo, onde os valores de 'map' são ponteiros a funções membro de 'Handler', que esperam a linha de comando entrada pelo usuário. Aqui estão as características de 'Handler':

- A classe é derivada de um 'std::map', que espera um comando associado a cada membro de processamento de comandos como suas chaves. Como 'Handler' usa a 'map' somente para definir associações entre o comando e as funções membro, usamos derivação privada aqui:

```
class Handler: public std::map<std::string,
                             void (Handler::*)(std::string const &cmd)>
```

- A associação atual pode ser definida usando membros de dados estáticos privados: 's_cmds' é um conjunto de valores 'Handler::value_type' e 's_cmds_end' é um ponteiro constante que aponta além do último elemento do conjunto:

```
static value_type s_cmds[];
static value_type *const s_cmds_end;
```

- O construtor simplesmente inicia o mapa desses dois membros de dados estáticos. Pode ser implantado em linha:

```
Handler()
:
    std::map<std::string,
            void (Handler::*)(std::string const &cmd)>
    (s_cmds, s_cmds_end)
{ }
```

- O membro 'process()' produz iterações ao longo dos elementos de 'map'. Uma vez que a primeira palavra na linha de comando coincida com o caracter inicial do comando, o comando correspondente é executado. Se não é encontrado tal comando uma mensagem de erro aparece:

```
void Handler::process(std::string const &line)
{
    istreambuf_iterator istr(line);
    string cmd;
    istr >> cmd;
    for (iterator it = begin(); it != end(); it++)
    {
        if (it->first.find(cmd) == 0)
        {
            (this->*it->second) (line);
            return;
        }
    }
    cout << "Unknown command: " << line << endl;
}
```

19.9.2: Derivando modelos de classes de modelos de classes

Apesar de que é perfeitamente aceitável derivar uma classe concreta de um modelo de classe, a classe resultante, está claro, tem uma generalidade limitada em comparação ao seu modelo de classe de base. Se a generalidade é importante, talvez seja melhor idéia derivar um modelo de classe de um modelo de classe. Isto nos permite estender um modelo de classe existente com algumas funcionalidades adicionais, como permitir ordenação hierárquica de seus elementos.

A seguinte classe 'SortVector' é um modelo de classe derivado do modelo de classe existente 'Vector'. Este nos permite realizar ordenação hierárquica de seus elementos, usando qualquer ordem de quaisquer membros que seus elementos de dados que possa conter. Para isto há um só requerimento: O tipo de dados de 'SortVector' precisa ter funções membro dedicadas à comparação de seus membros. Por exemplo, se o tipo de dados de 'SortVector' é um objeto da classe 'Multidata', então 'Multidata' deve implantar a função membro com o seguinte protótipo para cada um de seus membros de dados que podem ser comparados:

```
bool (MultiData::*)(MultiData const &rhv)
```

Assim, se 'Multidata' tem dois membros de dados, 'int d_value' e 'std::string d_text' e ambos podem ser requeridos para ordenação hierárquica, então 'Multidata' deve oferecer membros como:

```
bool intCmp(MultiData const &rhv); // retorna d_value < rhv.d_value
bool textCmp(MultiData const &rhv); // retorna d_text < rhv.d_text
```

Mais ainda, como conveniência também assume-se que 'operator<<()' e 'operator>>()' foram definidos para objetos 'Multidata', mas isto como tal é irrelevante nesta discussão.

O modelo de classe 'SortVector' é derivado diretamente do modelo de classe 'std::vector'. Nossa implantação herda todos os membros dessa classe de base, bem como dois simples construtores:

```
template <typename Type>
class SortVector: public std::vector<Type>
{
public:
    SortVector()
    {}
    SortVector(Type const *begin, Type const *end)
    :
        std::vector<Type>(begin, end)
    {}
}
```

Contudo, seu membro 'hierarchicalSort()' é a razão atual da existência dessa classe. Esta classe define o critério de ordenação hierárquico. Espera um conjunto de ponteiros membros às funções membro da classe indicada pelo parâmetro do modelo de 'SortVector' 'Type' bem como um 'size_t' que indica o tamanho do conjunto. O primeiro elemento do conjunto indica o mais significativo da classe ou o primeiro critério de ordenação, o último elemento do conjunto indica o menos significativo da classe ou o último critério de ordenação. Como o algoritmo genérico 'stable_sort()' foi projetado explicitamente para suportar ordenamento hierárquico, o membro usa este algoritmo genérico para ordenar os elementos de 'SortVector'. Com ordenação hierárquica, o elemento menos significativo pode ser ordenado primeiro. A implantação de 'hierarchicalSort()', portanto, é fácil, assumindo-se a existência de uma classe de suporte 'SortWith' cujos objetos são iniciados pelos endereços das funções membro passados ao membro 'hierarchicalSort()':

```
template <typename Type>
class SortWith
{
    bool (Type::*d_ptr)(Type const &rhv) const;
```

The class SortWith is a simple wrapper class around a pointer to a predicate function. Since it's dependent on SortVector's actual data type SortWith itself is also a template class:

```
template <typename Type>
```

```
class SortWith
{
    bool (Type::*d_ptr)(Type const &rhv) const;
```

It's constructor receives such a pointer and initializes the class' d_ptr data member:

```
SortWith(bool (Type::*ptr)(Type const &rhv) const)
:
    d_ptr(ptr)
{ }
```

Seu predicado binário 'operator()()' deve retornar verdadeiro se seu primeiro argumento está colocado antes do segundo pelo critério de ordenação:

```
bool operator()(Type const &lhv, Type const &rhv) const
{
    return (lhv.*d_ptr)(rhv);
}
```

Finalmente, uma ilustração é fornecida pela seguinte função 'main()':

- Primeiro, um objeto 'SortVector' é criado para objetos 'MultiData', usando o algoritmo genérico 'copy()' para preencher o objeto 'SortVector' com informação que aparece no programa numa 'stream' de entrada padrão. Tendo iniciado o objeto seus elementos são mostrados pela 'stream' de saída padrão:

```
SortVector<MultiData> sv;

copy(istream_iterator<MultiData>(cin),
     istream_iterator<MultiData>(),
     back_inserter(sv));
```

- Um conjunto de ponteiros a membros é iniciado com os endereços de duas funções membro. A comparação de texto é considerada o critério mais significativo de ordenação:

```
bool (MultiData::*arr[]) (MultiData const &rhv) const =
{
    &MultiData::textCmp,
    &MultiData::intCmp,
};
```

- Em seguida, o conjunto de elementos é ordenado e mostrado através da 'stream' padrão de saída:

```
sv.hierarchicalSort(arr, 2);
```

- Então os dois elementos do conjunto de apontadores às funções membros de 'MultiData' são trocadas e o passo anterior é repetido:


```
swap(arr[0], arr[1]);
sv.hierarchicalSort(arr, 2);
```

Depois da compilação do programa o seguinte comando pode ser dado:

```
echo a 1 b 2 a 2 b 1 | a.out
```

Aqui está a saída gerada:

```
a 1 b 2 a 2 b 1
====
a 1 a 2 b 1 b 2
====
a 1 b 1 a 2 b 2
====
```

19.9.3: Derivando modelos de classes de classes concretas

Uma classe existente pode ser usada como base para derivar um modelo de classe. A vantagem de tal árvore de herança é que os membros da classe de base podem todos serem compilados com anterioridade, assim, quando os objetos do modelo de classe são instanciados só os membros usados da (modelo de) classe precisam ser instanciados.

Esta solução pode ser usada para todos os modelos de classe com funções membros cuja implantação não depende dos modelos de parâmetros. Estes membros podem ser definidos numa classe separada que então é usada como classe de base do modelo de classe derivado dela.

Como ilustração desta solução desenvolveremos um tal modelo de classe nesta seção. Desenvolveremos a classe 'Table' derivada de uma classe concreta 'TableType'. A classe 'Table' mostrará elementos de algum tipo numa tabela com um número de colunas configurável. Os elementos são mostrados ou horizontalmente (os primeiros k elementos ocupando a primeira linha) ou verticalmente (os r primeiros elementos ocupando a primeira coluna).

Quando os elementos da tabela são mostrados são inseridos numa 'stream'. Isto permite-nos definir o manipulador da tabela numa classe separada ('TableType'), implantando a apresentação da tabela. Como os elementos da tabela são postos numa 'stream', a conversão a texto (ou string) deve ser implantada em 'Table', mas a manipulação da 'string' é deixada a 'TableType'. Cobriremos algumas características de 'TableType' de passagem, concentrando-nos primeiro na interface de 'Table':

- A classe 'Table' é um modelo de classe, requer só um tipo de modelo de parâmetro: 'Iterator' se refere a um tipo de iterador:

```
template <typename Iterator>
class Table: public TableType
{
```

- Não requer membros de dados: Toda manipulação de dados é feita por 'TableType';
- Possui dois construtores. O construtor dos dois primeiros parâmetros são 'Iterators' usados para iterar com os elementos ao entrar na tabela. Os construtores requerem a especificação o número de colunas que queremos que a tabela tenha, bem como 'FillDirection'. 'FillDirection' é um tipo 'enum' definido por 'TableType', tendo valores 'Horizontal' e 'Vertical'. Para permitir o controle dos usuários sobre os separadores de cabeçalhos, rodapés, título, horizontal e vertical, um construtor tem um parâmetro de referência 'TableSupport'. A classe 'TableSupport' será desenvolvida mais tarde como uma classe virtual, permitindo aos clientes realizarem esse controle. Eis os construtores da classe:

```
Table(Iterator const &begin, Iterator const &end,
      size_t nColumns, FillDirection direction);
Table(Iterator const &begin, Iterator const &end,
      TableSupport &tableSupport,
      size_t nColumns, FillDirection direction);
```

- Os construtores são os dois únicos membros públicos de 'Table'. Ambos usam um iniciador da classe de base para iniciar seus objetos da classe de base 'TableType' e então chamam o membro privado da classe 'fill()' para inserir dados nos objetos da classe de base 'TableType'. Eis a implantação dos construtores:

```
template <typename Iterator>
Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
                      TableSupport &tableSupport,
                      size_t nColumns, FillDirection direction)
:
    TableType(tableSupport, nColumns, direction)
{
    fill(begin, end);
}

template <typename Iterator>
Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
                      size_t nColumns, FillDirection direction)
:
    TableType(nColumns, direction)
{
    fill(begin, end);
}
```

- O membro da classe 'fill()' itera com os objetos entre '[begin, end)', como definido pelos primeiros dois parâmetros do construtor. Como veremos ligeiramente, 'TableType' define um membro de dados protegido 'std::vector<std::string> d_string'. Um dos requerimentos do tipo de

dados para o qual o iterador aponta é que este tipo de dado possa ser inserido numa 'stream'. Assim, 'fill()' usa um objeto 'ostringstream' para obter a representação textual dos dados, que são então passados a 'd_string':

```
template <typename Iterator>
void Table<Iterator>::fill(Iterator it, Iterator const &end)
{
    while (it != end)
    {
        std::ostringstream str;
        str << *it++;
        d_string.push_back(str.str());
    }
    init();
}
```

Isto completa a implantação da classe 'Table'. Note que este modelo de classe tem somente três membros, dois dos quais são construtores. Por isso, na maioria dos casos, só dois modelos de funções terão que ser instanciados: um construtor e o membro 'fill()'. Por exemplo, o seguinte constrói uma tabela com quatro colunas, preenchidas verticalmente por 'strings' extraídas da 'stream' de entrada padrão:

```
Table<istream_iterator<string> >
    table(istream_iterator<string>(cin), istream_iterator<string>(),
        4, TableType::Vertical);
```

Note aqui que o endereço de 'fill()' é especificado como 'TableType::Vertical'. Poderia também ter sido especificado usando 'Tble', mas como 'Table' é um modelo de classe a especificação se tornaria algo mais complexa: 'Table<istream_iterator<string> >::Vertical'.

Agora que a classe derivada 'Table' foi projetada, voltemos nossa atenção à classe 'TableType'. Eis suas características essenciais:

- É uma classe concreta, projetada como a classe de base de 'Table';
- Usa vários membros de dados privados, entre os quais 'd_colWidth', um vetor que guarda a largura do elemento mais largo por coluna e 'd_indexFun', que aponta para a função membro da classe que retorna o elemento em 'table[row][column]', relativo ao endereço da tabela a preencher. 'TableType' usa também um apontador 'TableSupport' e uma referência. O construtor não requer um objeto 'TableSupport' usa o 'TableSupport *' para alocar um objeto 'TableSupport' (padrão) e então usa o 'TableSupport &' como apelido do objeto. O outro construtor inicia o ponteiro em 0 e usa o membro de dados de referência para se referir ao objeto 'TableSupport' fornecido pelo seu parâmetro. Alternativamente um objeto estático 'TableSupport' poderia ter sido usado para iniciar o membro de dados de referência no construtor de formação. Os restantes membros de dados, provavelmente são auto-explicativos:

```

TableSupport      *d_tableSupportPtr;
TableSupport      &d_tableSupport;
size_t            d_maxWidth;
size_t            d_nRows;
size_t            d_nColumns;
WidthType         d_widthType;
std::vector<size_t> d_colWidth;
size_t            (TableType::*d_widthFun)
                  (size_t col) const;
std::string const &(TableType::*d_indexFun)
                  (size_t row, size_t col) const;

```

- Os objetos 'string' que habitam a tabela são guardados num membro de dados protegido:

```
std::vector<std::string> d_string;
```

- Os construtores (protegidos) realizam tarefas básicas: Iniciam os objetos membros de dados. Aqui está o construtor que espera uma referência a um objeto 'TableSupport':

```

#include "tabletype.ih"

TableType::TableType(TableSupport &tableSupport, size_t nColumns,
                     FillDirection direction)
:
    d_tableSupportPtr(0),
    d_tableSupport(tableSupport),
    d_maxWidth(0),
    d_nRows(0),
    d_nColumns(nColumns),
    d_widthType(ColumnWidth),
    d_colWidth(nColumns),
    d_widthFun(&TableType::columnWidth),
    d_indexFun(direction == Horizontal ?
                &TableType::hIndex
                :
                &TableType::vIndex)
{}

```

- Uma vez que 'd_data' foi preenchida, a tabela é iniciada por 'Table::fill()'. O membro protegido 'init()' re-dimensiona 'd_data' de tal maneira que seu tamanho seja exatamente linhas X colunas e determina a largura máxima dos elementos por linha. Sua implantação é clara:

```

#include "tabletype.ih"

void TableType::init()
{
    if (!d_string.size()) // sem elementos
        return;          // não faz nada

    d_nRows = (d_string.size() + d_nColumns - 1) / d_nColumns;
}

```

```

d_string.resize(d_nRows * d_nColumns); // força a tabela completa

// determina max. larg. por coluna,
// e max. larg. da coluna
for (size_t col = 0; col < d_nColumns; col++)
{
    size_t width = 0;

    for (size_t row = 0; row < d_nRows; row++)
    {
        size_t len = stringAt(row, col).length();
        if (width < len)
            width = len;
    }
    d_colWidth[col] = width;

    if (d_maxWidth < width) // max. width so far.
        d_maxWidth = width;
}
}

```

- O membro público 'insert()' é usado pelo operador de inserção ('operator<<()') para inserir uma tabela numa 'stream'. Primeiro ele informa o objeto 'TableSupport' sobre as dimensões da tabela. Em seguida mostra a tabela, permitindo ao objeto 'TableSupport' escrever cabeçalhos, rodapés e separadores:

```

#include "tabletype.ih"

ostream &TableType::insert(ostream &ostr) const
{
    if (!d_nRows)
        return ostr;

    d_tableSupport.setParam(ostr, d_nRows, d_colWidth,
                           d_widthType == EqualWidth ? d_maxWidth : 0);

    for (size_t row = 0; row < d_nRows; row++)
    {
        d_tableSupport.hline(row);

        for (size_t col = 0; col < d_nColumns; col++)
        {
            size_t colwidth = width(col);

            d_tableSupport.vline(col);
            ostr << setw(colwidth) << stringAt(row, col);
        }

        d_tableSupport.vline();
    }
}

```

```

        d_tableSupport.hline();

        return ostr;
    }

```

- O arquivo `cplusplus.yo.zip` contém a implantação completa de 'TableSupport'. Este arquivo é encontrado no diretório `yo/templateclasses/examples/table`. A maioria de seus membros restantes são privados. Entre eles os seguintes dois membros retornam os elementos da tabela '[row][column]' para, respectivamente, uma tabela preenchida horizontalmente e preenchida verticalmente:

```

std::string const &hIndex(size_t row, size_t col) const
{
    return d_string[row * d_nColumns + col];
}
inline std::string const &TableType::vIndex(size_t row, size_t col) const
{
    return d_string[col * d_nRows + row];
}

```

A classe de suporte 'TableSupport' é usada para mostrar cabeçalhos, rodapés, títulos e separadores. Possui quatro membros virtuais para realizar estas tarefas (e, claro, um construtor virtual):

- 'hline(size_t rowIndex)': Chamado justo antes dos elementos da linha 'rowIndex' ser mostrado;
- 'hline()': Chamado imediatamente depois do final de 'row';
- 'vline(size_t colIndex)': Chamado justo antes do elemento da coluna 'colIndex' ser mostrado;
- 'vline()': Chamado justo depois de todos os elementos de uma linha serem mostrados.

Para a implementação completa das classes 'Table', 'TableType' e 'TableSupport' o leitor pode se referir ao arquivo `cplusplus.yo.zip`. Eis um pequeno programa mostrando o uso delas:

```

/*
    table.cc
*/

#include <fstream>
#include <iostream>
#include <string>
#include <iterator>
#include <sstream>

#include "tablesupport/tablesupport.h"
#include "table/table.h"

using namespace std;

```

```

using namespace FBB;

int main(int argc, char **argv)
{
    size_t nCols = 5;
    if (argc > 1)
    {
        istringstream iss(argv[1]);
        iss >> nCols;
    }

    istream_iterator<string> iter(cin); //o primeiro iterador não é const

    Table<istream_iterator<string> >
        table(iter, istream_iterator<string>(), nCols,
            argc == 2 ? TableType::Vertical : TableType::Horizontal);

    cout << table << endl;
    return 0;
}
/*
Exemplo da saída gerada:
After: echo a b c d e f g h i j | demo 3
    a e i
    b f j
    c g
    d h
After: echo a b c d e f g h i j | demo 3 h
    a b c
    d e f
    g h i
    j
*/

```

19.9.4: Resolução de tipo em membros de classe de base

Considere o seguinte exemplo de um modelo de classe de base e uma derivada:

```

#include <iostream>

template <typename T>
class Base
{
public:
    void member()
    {
        std::cout << "This is Base<T>::member()\n";
    }
};

```

```

template <typename T>
class Derived: public Base<T>
{
    public:
        Derived()
        {
            member();
        }
};

```

Este exemplo não compilará e o compilador nos dirá algo assim:

```

error: there are no arguments to 'member' that depend on a template
parameter, so a declaration of 'member' must be available

```

A primeira vista, este erro pode causar alguma confusão, já que com classes não-modelo os membros da classe de base `public` e `protected` ficam imediatamente disponíveis. Isto também é verdade para modelos de classes, mas só se o compilador pode compreender o que queremos. Na situação acima, o compilador não pode, já que não sabe para que tipo o membro `T` da função `member` deve ser iniciado.

Para descobrirmos porque isto é verdade, considere a situação onde definimos uma especialização:

```

template <>
Base<int>::member()
{
    std::cout << "This is the int-specialization\n";
}

```

Como o compilador, quando processe a classe `Derived`, estar certo de que nenhuma especialização estará em efeito quando uma instanciação de `Derived` for chamada, não pode decidir agora para que tipo instanciar `member`, já que a chamada a `member()` em `Derived::Derived()` não requer um modelo de tipo de parâmetro. Em casos como estes, onde não se dispõe de um modelo de tipo do parâmetro é lícito determinar que tipo usar, o compilador reterá que pospõe sua decisão sobre o modelo de tipo do parâmetro usar para `member()` até sua instanciação. Isto pode ser feito de duas formas: ou usando `this`, ou mencionando explicitamente a classe de base, instanciada para um modelo de tipo(s) da classe derivada. Na seguinte função `main()` ambas formas são usadas. Note que com o modelo de tipo `int` a especialização de `int` é usada.

```

#include <iostream>

template <typename T>
class Base
{
    public:
        void member()
        {
            std::cout << "This is Base<T>::member()\n";
        }
}

```



```

        }
    };

    template <>
    void Base<int>::member()
    {
        std::cout << "This is the int-specialization\n";
    }

    template <typename T>
    class Derived: public Base<T>
    {
    public:
        Derived()
        {
            this->member();
            Base<T>::member();
        }
    };

    int main()
    {
        Derived<double> d;
        Derived<int> i;
    }

    /*
    Generated output:
    This is Base<T>::member()
    This is Base<T>::member()
    This is the int-specialization
    This is the int-specialization
    */

```

19.10: Modelos de classes e aninhamento

Quando uma classe é aninhada dentro de um modelo de classe, esta automaticamente se torna um modelo de classe.

A classe aninhada pode usar os modelos de parâmetros da classe envolvente, como mostra o seguinte esqueleto de programa. Numa classe 'PtrVector' é definido um iterador. A classe aninhada recebe sua informação da classe envolvente, uma classe 'PtrVector<Type>'. Enquanto esta classe envolvente é a única que pode construir seus iteradores, o construtor dos iteradores é privado e à classe

envolvente é dado acesso aos membros privados de 'reverse_iterator' usando uma declaração de amiga limite.

Eis a seção inicial da interface da classe 'PtrVector':

```
template <typename Type>
class PtrVector: public std::vector<Type *>
```

Isto mostra que a classe de base 'std::vector' guardará apontadores a valores do tipo 'Type', antes que valores propriamente ditos. Claro está que agora é necessário um destrutor, já que a memória (alocada externamente) dos objetos 'Type' necessitam eventualmente serem liberados. Alternativamente a alocação poderia ser parte das tarefas de 'PtrVector', quando guarde novos elementos. Aqui é assumido que os clientes de 'PtrVector' fazem as alocações requeridas e que o destrutor será implantado mais tarde.

A classe aninhada define seu construtor como membro privado e permite a objetos 'PtrVector<Type>' acessarem seus membros privados. Por isso só a objetos de tipo da classe envolvente 'PtrVector<Type>' é permitido construir seus objetos iteradores. Contudo, clientes de 'PtrVector<Type>' podem construir cópias dos objetos 'PtrVector<Type>::iterator' que usem. Aqui está a classe aninhada iteradora, contendo a declaração de amiga requerida. Note o uso da palavra chave 'typename': Como 'std::vector<Type *>::iterator' depende de um modelo de parâmetro, ainda não é uma classe instanciada, assim 'iterator' se torna um 'typename' implícito. Se 'typename' for omitido o compilador lança uma mensagem de atenção correspondente. O aviso é para usar 'typename' nesses casos. Ainda mais, note que o membro da classe de base 'begin()' é chamado para iniciar 'd_begin':

```
class iterator
{
    friend class PtrVector<Type>;
    typename std::vector<Type *>::iterator d_begin;

    iterator(PtrVector<Type> &vector)
    :
        d_begin(vector.std::vector<Type *>::begin())
    {}
public:
    Type &operator*()
    {
        return **d_begin;
    }
};
```

O restante da classe é simples. Omitindo todas as outras funções que devem ser implantadas, a função 'begin()' retorna um 'PtrVector<Type>' recém construído. Pode chamar o construtor, já que a classe do iterador chamou sua classe envolvente de amiga:

```
iterator begin()
{
```

```

        return iterator(*this);
    }

```

Eis aqui o esqueleto de um programa mostrando como usar a classe aninhada 'iterator':

```

int main()
{
    PtrVector<int> vi;

    vi.push_back(new int(1234));

    PtrVector<int>::iterator begin = vi.begin();

    std::cout << *begin << endl;
}

```

Enumerações de 'typedefs' também podem ser definidas em modelos de classes. A classe 'Table', mencionada anteriormente (seção 19.9.3) herda a enumeração 'TableType::FillDirection'. Se 'Table' tivesse sido implantada como um modelo completo de classe, então esta enumeração teria sido definida em 'Table' como:

```

template <typename Iterator>
class Table: public TableType
{
    public:
        enum FillDirection
        {
            Horizontal,
            Vertical
        };
        ...
};

```

Neste caso o valor do modelo de tipo de parâmetro precisa ser especificado quando se refere ao valor de 'FillDirection' ou ao seu tipo. Por exemplo (assumindo que 'iter' e 'nCols' são definidos como na seção 19.9.3):

```

Table<istream_iterator<string> >::FillDirection direction =
    argc == 2 ?
        Table<istream_iterator<string> >::Vertical
    :
        Table<istream_iterator<string> >::Horizontal;

Table<istream_iterator<string> >
    table(iter, istream_iterator<string>(), nCols, direction);

```

19.10.1: Retorno de tipos aninhados em modelos de classes

Na seção [19.1.3](#) a palavra chave 'typename' foi introduzida para permitir ao compilador distinguir

entre membros de um modelo de classe e tipos definidos com modelos de classes. A palavra chave 'typename' é usada como uma ferramenta que permite-nos dizer ao compilador que temos em mente um tipo que está aninhado num modelo de classe.

Considere o seguinte exemplo no qual uma classe aninhada, que não depende de um modelo de parâmetro, é definida dentro de um modelo de classe. Mais ainda, o membro do modelo de classe `nested()` que retorna um objeto da classe aninhada:

```
template <typename T>
class Outer
{
    public:
        class Nested
        {
        };

        Nested nested() const
        {
            return Nested();
        }
};
```

O exemplo acima compila perfeitamente: na classe `Outer` não há ambigüidade respeito ao significado do tipo de retorno de `nested()`. Contudo a situação muda quando tentamos implantar `nested()` fora de sua classe. A seguinte tentativa de o fazer falha:

```
template <typename T>
class Outer
{
    public:
        class Nested
        {
        };

        Nested nested() const;
};

template <typename T>
Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}
```

Este intento produz uma mensagem de erro como:

error: expected constructor, destructor, or type conversion before 'Outer'.

Neste caso um tipo de conversão é requerido como `Outer<T>::Nested` se refere a um *tipo*, aninhado em `Outer<T>` antes que a um membro de `Outer<T>`. Conseqüentemente, uma palavra

chava `typename` precisa ser usada para obrigar o compilador interpretar `Outer<T>::Nested` como um nome de tipo. Escrevendo-se `typename` na frente de `Outer<T>::Nested` se remove o erro de compilação. A implementação correta da função `nested()` fica:

```
template <typename T>
typename Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}
```

19.11: Construindo iteradores

Na seção 17.2 os iteradores usados com algoritmos genéricos foram introduzidos. Vimos os diversos tipos de iteradores que se distinguem como: `InputIterators`, `ForwardIterators`, `OutputIterators`, `BidirectionalIterators` e `RandomAccessIterators`.

Contudo, quando os iteradores têm que ser usados no contexto dos algoritmos genéricos, necessitam requerimentos adicionais. Isto devido ao fato que os algoritmos genéricos examinam os tipos de iteradores que recebem. Simples apontadores usualmente são aceitos, mas se um objeto iterador é usado precisa ser capaz de especificar que natureza de iterador representa.

Para assegurar que um objeto da classe seja interpretado como um tipo particular de iterador, a classe precisa ser derivada da classe 'iterator'. O tipo particular de iterador é definido pelo primeiro parâmetro do modelo de classe e o tipo particular de dados para o tipo de dados que aponta é definido como segundo parâmetro do modelo de classe. Antes de uma classe poder herdar da classe 'iterator', o seguinte arquivo cabeçalho tem que ser incluído:

```
#include <iterator>
```

O tipo particular de iterador implantado pela classe derivada é especificado usando-se o assim chamado 'iterator_tag', como primeiro modelo de argumento da classe 'iterator'. Para os cinco tipos básicos de iteradores essas citações são:

- 'std::input_iterator_tag': Esta citação define um iterador de entrada. Os iteradores deste tipo permitem operações de leitura, iterando do primeiro ao último elemento da série à que o iterador se refere;
- 'std::output_iterator_tag': Define um iterador de saída. Este tipo de iteradores permite operações de adjudicação, iterando do primeiro ao último elemento da série a que se refere;
- 'std::forward_iterator_tag': Define um iterador de avanço. Este tipo permite operações de

leitura e adjudicação, iterando do primeiro ao último dos elementos da série a que se refere;

- `'std::bidirectional_iterator_tag'`: Define um iterador bidirecional. Permite operações de leitura e adjudicação, iterando passo a passo, possivelmente em direções alternadas, sobre todos os elementos da série a que se refere.
- `'std::random_access_iterator_tag'`: Define um iterador de acesso aleatório. Permite operações de leitura e adjudicação, iterando, possivelmente em ambas direções, sobre todos os elementos da série a que se refere, usando qualquer passo possível (aleatório).

Cada citação de iterador assume que um certo conjunto de operações são possíveis. O iterador `'random_access_iterator'` é o mais complexo e implica todos os outros tipos.

Note que os iteradores sempre são definidos sobre um espaço, p.ex., `'[begin, end)`. As operações de incremento e decremento podem resultar em comportamento indefinido do iterador se o valor resultante se referir a uma localização fora do espaço definido.

Freqüentemente os iteradores acessam os elementos da série a que se referem. Internamente um iterador usa um ponteiro normal, mas é desnecessário um iterador alocar sua própria memória. Por isso, como o operador sobrecarregado de adjudicação e o construtor de cópias não necessitam alocar memória, a implantação padrão do operador sobrecarregado de adjudicação e construtor de cópias, usualmente, é suficiente. I.e., em geral estes membros não têm que ser implantados. Como consequência também não tem destrutor.

A maioria das classes que oferecem membros que retornam iteradores possuem membros que constroem o iterador requerido, que é retornado como um objeto por essas funções membro. Como o visitante dessas funções membro só tem que usar ou às vezes copiar os objetos iteradores, normalmente não há necessidade de existir construtores disponíveis publicamente, exceto para o construtor de cópias. Por isso esses construtores podem, usualmente, serem definidos como membros privados ou protegidos. Para permitir uma classe externa criar objetos iteradores, a classe `'iterator'` declarará a classe externa como amiga.

Nas seguintes seções a construção de um `'RandomAccessIterator'`, o mais complexo de todos os iteradores, e a construção de um `'RandomAccessIterator'` reverso é discutida. A classe recipiente onde o iterador de acesso aleatório tem que ser desenvolvido deve guardar seus elementos de dados de diferentes maneiras, p.ex., usando vários recipientes ou usando ponteiros a ponteiros. Por isso é difícil construir um modelo de classe iterador apropriado para uma grande variedade de classes concretas (recipientes).

Nas seções seguintes a classe disponível 'std::iterator' será usada para construir uma classe de entrada que representa um iterador de acesso aleatório. Esta solução mostra claramente como construir uma classe de iteradores. O leitor seguirá esta solução ao construir classes iteradoras em outro contexto ou uma classe completa de iteradores pode ser projetada. Um exemplo de tal modelo de classe iteradora é dado na seção 20.5.

A construção de um iterador de acesso aleatório, como mostram as seções seguintes, visam a realização de um iterador que acesse elementos de uma série de elementos só acessíveis através de ponteiros. A classe de iteradores é projetada como classe de entrada a uma classe derivada de um vetor de ponteiros a 'strings'.

19.11.1: Implantando um 'RandomAccessIterator'

Quando discutimos os recipientes (Capítulo 12) notamos que os recipientes contém sua própria informação. Se contém objetos, então esses objetos são destruídos uma vez destruído o recipiente. Como os apontadores não são objetos e como 'auto_ptr' não pode ser armazenado em recipientes, usando o tipo de dados ponteiro para recipientes é desencorajador. Contudo, poderíamos usar o tipo de dados ponteiro em recipientes em contextos específicos. Na seguinte classe 'StringPtr', uma classe concreta, derivada do recipiente 'std::vector', usando 'std::string *' como tipo de dados:

```
#ifndef __INCLUDED_STRINGPTR_H_
#define __INCLUDED_STRINGPTR_H_

#include <string>
#include <vector>

class StringPtr: public std::vector<std::string *>
{
    public:
        StringPtr(StringPtr const &other);
        ~StringPtr();

        StringPtr &operator=(StringPtr const &other);
};

#endif
```

Note a declaração do destrutor: Como o objeto guarda ponteiros a 'strings', se requer um destrutor para destruir as 'strings' quando o objeto 'StringPtr' for destruído. Analogamente, um construtor de cópias e um operador sobrecarregado de adjudicação são requeridos. Outros membros (em particular construtores) não são explicitamente declarados, já que não são relevantes para o tópico desta seção.

Assumamos que queremos usar o algoritmo genérico 'sort()' em objetos 'StringPtr'. Este

algoritmo (veja seção 17.4.58) requer dois 'RandomAccessIterators'. Porém estes iteradores estão disponíveis (via os membros 'begin()' e 'end()' de `std::vector`), retornam iteradores a `'std::string *s'`, que não podem ser comparados.

Para remediar isto, assuma que temos definido um tipo interno `'StringPtr::iterator'`, não retornando iteradores a ponteiros, mas iteradores aos objetos que esses apontadores apontam. Uma vez que este tipo de iterador esteja disponível, podemos agregar os seguintes membros à interface de classe de `'StringPtr'`, anulando os membros de sua classe de base de idênticos nomes, mas sem serventia:

```
StringPtr::iterator begin();    // retorna iterador ao primeiro elemento
StringPtr::iterator end();     // retorna iterador além do último
                              // elemento
```

Como esses dois membros retornam os iteradores (apropriados), os elementos de um objeto `'StringPtr'` podem facilmente serem ordenados:

```
in main()
{
    StringPtr sp;                // se assume que sp está preenchida

    sort(sp.begin(), sp.end()); // sp agora é ordenada
    return 0;
}
```

Para fazer isto tudo funcionar, o tipo `'StringPtr::iterator'` deve ser definido. Como sugerido pelo seu nome de tipo, `'iterator'` é um tipo de `'StringPtr'`, sugerindo que `'iterator'` pode ser implantado como uma classe aninhada de `'StringPtr'`. Contudo, para usar um `'StringPtr::iterator'` em combinação com o algoritmo genérico `'sort()'`, precisa também ser um `'RandomAccessIterator'`. Para isso, `'StringPtr::iterator'` precisa ser derivado da classe existente `'std::iterator'`, disponível uma vez que a diretiva ao pré-processador seja especificada:

```
#include <iterator>
```

Para derivar a classe de `'std::iterator'`, precisamos especificar ambos o tipo do iterador e o tipo de dados para o qual aponta. Tenha cuidado: Nosso iterador cuidará da referência de `'string *'`, assim o tipo de dado requerido será `'std::string'` e não `'std::string *'`. Dessa maneira a interface da classe `'iterator'` começa como:

```
class iterator:
    public std::iterator<std::random_access_iterator_tag, std::string>
```

Como a especificação de sua classe de base é bastante complexa, associamos este tipo a um nome menor, usando a seguinte definição de tipo:

```
typedef std::iterator<std::random_access_iterator_tag, std::string>
    Iterator;
```

Agora estamos prontos para re-projetar a interface de classe de `'StringPtr'`, até sua classe

aninhada 'iterator':

```
class StringPtr: public std::vector<std::string *>
{
    typedef std::iterator<std::random_access_iterator_tag, std::string>
        Iterator;

public:
    class iterator: public Iterator
```

Vejamos as características de 'StringPtr::iterator':

- A classe 'iterator' define 'StringPtr' como sua amiga, assim o construtor de 'iterator' pode permanecer privado: Só a classe 'StringPtr' está habilitada a construir 'iterators', o que parece importante. Ainda mais, como um 'iterator' já é fornecido pela classe de base 'StringPtr', podemos usar esse iterador para acessar a informação guardada no objeto 'StringPtr'. Dessa forma a seção privada de 'iterator' fica:

```
friend class StringPtr;

std::vector<std::string *>::iterator d_current;

iterator(std::vector<std::string *>::iterator const &current)
:
    d_current(current)
{ }
```

- Os membros 'StringPtr::begin()' e 'StringPtr::end()' podem agora retornar objetos 'iterator'. Definido em linha na classe 'StringPtr', a implantação fica:

```
iterator begin()
{
    return iterator(this->std::vector<std::string *>::begin() );
}
iterator end()
{
    return iterator(this->std::vector<std::string *>::end() );
}
```

- Todos os outros membros restantes de 'iterator' são públicos. É bem fácil implantá-los, principalmente manipulam e referenciam o iterador disponível 'd_current'. Para definir um 'RandomAccessIterator', pelo menos os seguintes operadores devem ser implantados:

- Iterador '&operator++()': Operador de pré-incremento:

```
iterator &operator++()
{
    ++d_current;
    return *this;
}
```

```
}
```

- Iterador '&operator--()': Operador de pré-decremento:

```
iterator &operator--()
{
    --d_current;
    return *this;
}
```

- Iterador 'operator--()': Operador de pós-decremento:

```
iterator const operator--(int)
{
    return iterator(d_current--);
}
```

- Iterador '&operator==(iterator const &other)': Operador sobrecarregado de adjudicação. Como os objetos 'iterator' não alocam memória por si sós, o operador padrão de adjudicação o fará.

```
bool operator==(iterator const &other) const
{
    return d_current == other.d_current;
}
```

- 'bool operator<(iterator const &rhv) const': Testa se o iterador à esquerda aponta a um elemento da série anterior ao elemento apontado pelo iterador à direita:

```
bool operator<(iterator const &other) const
{
    return **d_current < **other.d_current;
}
```

- 'int operator-(iterator const &rhv) const': Retorna o número de elementos entre o elemento apontado pelo iterador à esquerda e o iterador à direita (i.e., o valor a ser adicionado ao iterador à esquerda para fazê-lo igual ao iterador à direita):

```
int operator-(iterator const &rhs) const
{
    return d_current - rhs.d_current;
}
```

- 'Type &operator*() const': Retorna uma referência ao objeto para o qual o iterador aponta. Com um 'InputIterator' e com todos os iteradores constantes o tipo de retorno deste operador sobrecarregado deve ser 'Type const &'. Este operador retorna uma referência a uma 'string'. Esta 'string' é obtida de-referenciando o valor de-referenciado de d_current. Como 'd_current' é um iterador a elementos 'string *', duas operações de de-referência são requeridas para chegar à 'string':

```
std::string &operator*() const
```

```

{
    return **d_current;
}

```

- Iterador 'const operator+(int stepsize) const': Este operador avança o iterador com um tamanho de passo dado por 'stepsize':

```

iterator const operator+(int step) const
{
    return iterator(d_current + step);
}

```

- Iterador 'const operator-(int stepsize) const': Este operador decrementa o iterador com um tamanho de passo dado por 'stepsize':

```

iterator const operator-(int step) const
{
    return iterator(d_current - step);
}

```

Os iteradores podem ser construídos dos iteradores existentes. Este construtor não precisa ser implantado, pois o construtor de cópias pode ser usado.

- 'std::string *operator->() const': É um operador adicional. Aqui só uma operação de de-referência é requerida, retornando um ponteiro a uma 'string', permitindo-nos acessar os membros de uma 'string' via seu apontador:

```

std::string *operator->() const // access the fields of the
{                               // struct an iterator points
    return *d_current;          // to. E.g., it->length()
}

```

- Mais dois operadores adicionais são 'operator+=(int step)' e 'operator-=(int step)'. Não são formalmente requeridos por 'RandomAccessIterators', mas são úteis de todas formas:

```

iterator &operator+=(int step) // increment the iterator
{                               // over `n' steps
    d_current += step;
    return *this;
}
iterator &operator-=(int step) // decrement the iterator
{                               // over `n' steps
    d_current -= step;
    return *this;
}

```

As interfaces requeridas para outros tipos de iteradores são mais simples, requerendo só um subconjunto da interface requerida pelo iterador de acesso aleatório. P.ex., o iterador de avanço nunca é decrementado ou incrementado em passos de tamanho arbitrário. Conseqüentemente nesse caso todos os operadores de decrementação e incrementação podem ser omitidos da interface. Claro que a citação a

usar então seria 'std::forward_iterator_tag'.

19.11.2: Implantando um `iterador reverso`

Uma vez que implantamos um iterador, o iterador reverso que combine com esse pode ser implantado. Comparativamente a 'std::iterator' um 'std::reverse_iterator' existe, que perfeitamente implantará um iterador reverso para nós, uma vez que definamos uma classe iterador. Seu construtor requer somente um objeto do tipo iterador para o qual queremos construir um iterador reverso.

Para implantar um iterador reverso para 'StringPtr', só necessitamos definir o tipo de iterador reverso em sua interface. Isto requer que especifiquemos só uma linha de código, que deve ser inserida depois da interface da classe iterador:

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Finalmente os membros bem conhecidos 'rbegin()' e 'rend()' são agregados à interface de 'StringPtr'. Podem, muito bem, serem implantados em linha:

```
reverse_iterator rbegin()
{
    return reverse_iterator(end());
}
reverse_iterator rend()
{
    return reverse_iterator(begin());
}
```

Note os argumentos que os construtores de 'reverse_iterator' recebem: O ponto de início é obtido provendo o construtor do iterador reverso com 'end()', o ponto final do iterador normal; o ponto final do iterador reverso é obtido provendo o construtor do iterador reverso com 'begin()', o ponto inicial do iterador normal.

O pequeno programa a seguir ilustra o uso do RandomAccessIterator de 'StringPtr':

```
#include <iostream>
#include <algorithm>
#include "stringptr.h"
using namespace std;

int main(int argc, char **argv)
{
    StringPtr sp;

    while (*argv)
        sp.push_back(new string(*argv++));
}
```

```

    sort(sp.begin(), sp.end());
    copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));

    cout << "\n=====\n";

    sort(sp.rbegin(), sp.rend());
    copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));

    cout << endl;
}
/*
    Quando chamado como:
    a.out bravo mike charlie zulu quebec

    Gera a saída:
    a.out bravo charlie mike quebec zulu
    =====
    zulu quebec mike charlie bravo a.out
*/

```

Apesar de que é possível construir um iterador reverso de um normal, o oposto não é verdade: Não é possível iniciar um iterador normal a partir de um reverso.

Assumamos que queremos processar todas as linhas guardadas num 'vector<string>' até qualquer linha seguida por linhas vazias (ou linhas contendo espaços somente) que contenha. Como proceder? Uma solução é começar o processamento da primeira linha do vetor, continuando até a primeira linha seguida por linhas vazias. Contudo, ao encontrarmos uma linha vazia não necessariamente tem que ser a primeira linha do conjunto que seguem linha vazias. Nesse caso, devemos usar o seguinte algoritmo:

- Primeiro use:

```
rit = find_if(lines.rbegin(), lines.rend(), NonEmpty());
```

Para obter um iterador reverso 'rit' que aponte para a última linha não vazia.

- Em seguida use:

```
for_each(lines.begin(), --rit, Process());
```

Para processar todas as linhas até a primeira linha vazia.

Contudo, não podemos misturar iteradores com iteradores reversos ao usarmos o algoritmo genérico. Como podemos iniciar o segundo iterador usando o iterador reverso disponível? A solução não é difícil, já que um iterador pode ser iniciado por um apontador. O iterador reverso 'rit' não é um ponteiro, mas '&*(rit - 1)' ou '&*--rit' é. Portanto podemos usar:

```
'for_each(lines.begin(), &*--rit, Process());
```

Para processar todas as linhas até à primeira do conjunto seguido por linhas vazias. Em geral, se 'rit' é um iterador reverso, que aponta para algum elemento, mas precisamos um iterador que aponte a esse elemento, podemos usar '&*rit' para iniciar o iterador. Aqui, o operador de de-referência é aplicado para se chegar ao elemento a que o iterador reverso se refere. Então o operador de endereço é aplicado para obter seu endereço.

Capítulo 20: Aplicações de Modelos Avançados

A finalidade principal dos modelos é fornecer uma definição genérica das classes e das funções que podem então ser adaptadas aos tipos específicos quando requeridas.

Entretanto, os moldes permitem que nós façam mais que isso. Se não for por limitações do compilador, os modelos podem ser usados para programar em tempo de compilação qualquer coisa que para as quais usamos os computadores. Este fato notável, oferecido por nenhuma outra linguagem de programação atual, vem do fato que os modelos permitem que façamos três coisas em tempo de compilação:

- Os modelos permitem-nos executar aritmética de números inteiros e salvar aos valores computados como símbolos;
- Os modelos nos permitem tomar decisões em tempo de compilação;
- Os modelos nos permitem realizar coisas repetidamente.

Naturalmente, pedir que o compilador compute, por exemplo, números primos, é uma coisa. É uma coisa completamente diferente fazê-lo de maneira que ganhe em velocidade. Não espere quebrar recordes de velocidade quando o compilador executar cálculos complexos para nós. Mas isto está além do ponto do que podemos pedir que o compilador compute virtualmente qualquer coisa que usa a linguagem de modelação da C++.

Neste capítulo estas características notáveis dos modelos são discutidas. Depois de uma vista geral curta das sutilezas relacionados aos modelos, as características principais da meta programação de modelos são introduzidas.

Seguindo essa discussão um terceiro tipo de parâmetro de modelos, o parâmetro modelo de modelo é introduzido, colocando o traço básico para a discussão de classes e da política de classes.

Este capítulo termina com a discussão de diversas aplicações adicionais e interessantes de modelos: adaptando mensagens de erro do compilador, conversões aos tipos da classe e um exemplo elaborado que discute o processamento de lista em tempo de compilação.

Muita da inspiração para este capítulo resultou de dois livros altamente recomendados:

Andrei Alexandrescu 2001, Modern C++ design (Addison-Wesley)

Nicolai Josutis e David Vandevoorde 2003, *Templates* (Addison-Wesley)

20.1: *Subtleties (sutilezas)*

Nesta seção os seguintes tópicos são cobertos:

- Em 20.1.1 o uso da palavra chave ``typename`` é discutido. É usada para distinguir os tipos definidos por modelos da classe dos membros definidos por modelos da classe;
- Em 20.1.2 aplica ``typename`` às situações onde os tipos aninhados nos modelos são retornados das funções de membros de modelos da classe;
- Em 20.1.3 se cobre o problema de como consultar aos modelos da classe de base desde os modelos de classes derivadas;
- E em 20.1.4 se cobre alguma sintaxe nova: ``::template`` e variantes, usadas para informar o compilador que um nome usado dentro de um modelo é ele próprio um modelo da classe.

20.1.1: A palavra chave ``typename``

A palavra chave ``typename`` foi usada até agora para indicar um tipo de parâmetro do modelo. Entretanto, é usada também para evitar ambigüidades no código dentro dos modelos. Considere o seguinte código:

```
template <typename Type>
Type function(Type t)
{
    Type::Ambiguous *ptr;

    return t + *ptr;
}
```

Quando este código é mostrado ao compilador, este queixar-se-á com uma mensagem de erro confundindo a primeira vista como:

```
demo.cc:4: error: 'ptr' was not declared in this scope
```

A natureza confusa desta mensagem de erro é que a intenção do programador era realmente declarar

um ponteiro a um tipo ``ambiguous`` (ambíguo) definido dentro do tipo do modelo da classe. Entretanto, o compilador, quando confrontado com ``Type::Ambiguous`` tem que fazer uma decisão sobre a natureza desta construção. Claramente não pode inspecionar o tipo para encontrar sua natureza verdadeira, desde que o tipo é um tipo do modelo, e daqui sua definição real não está disponível ainda. O compilador é confrontado agora com as duas possibilidades: ou ``Type::Ambiguous`` é um membro estático do tipo misterioso do modelo, ou é um subtipo definido por ``Type``. Porque o padrão do compilador especifica que deve supor o anterior, a indicação:

```
Type::Ambiguous *ptr;
```

que é interpretada, eventualmente, como uma multiplicação do membro estático ``Type::Ambiguous`` e (agora o ``ptr``) da entidade não declarada. A razão para a mensagem de erro deve agora estar aclarada: neste contexto o ``ptr`` é desconhecido.

Para evitar ambigüidades num código em que um identificador consulta a um tipo que seja ele próprio um subtipo de um tipo de parâmetro do modelo a palavra chave ``typename`` deve ser usada. Assim, o código acima é alterado em:

```
template <typename Type>
Type function(Type t)
{
    typename Type::Ambiguous *ptr;

    return t + *ptr;
}
```

As classes definem com razoável frequência subtipos. Quando tais classes forem pensadas no projeto dos modelos, estes subtipos podem aparecer dentro das definições dos modelos como subtipos do tipo de parâmetros do modelo, requerem o uso da palavra chave ``template``. Por exemplo, suponha que o modelo da classe ``Handler`` define um ``typename`` ``Container`` como seu tipo de parâmetro, bem como um membro de dados que armazena o iterador ``begin()`` do ``container``. Além disso, o modelo da classe ``Handler`` pode oferecer um construtor que aceita qualquer recipiente que suporta um membro ``begin()``. O esqueleto do alimentador da classe poderia então ser:

```
template <typename Container>
class Handler
{
    Container::const_iterator d_it;

public:
    Handler(Container const &container)
    :
        d_it(container.begin())
    {}
};
```

Quais eram as considerações que tínhamos em mente ao projetar esta classe?

- O recipiente de `typename` representa todos os iteradores suportados pelo recipiente.
- O recipiente suporta presumivelmente `begin()` do membro. A iniciação `d_it(container.begin())` depende claramente do tipo de parâmetro do modelo, assim que só é verificado para ver se há a exatidão sintática básica.
- Do mesmo modo, o recipiente suporta, presumivelmente, um `const_iterator` do subtipo, definido no recipiente da classe.

A consideração final é uma indicação de é requerido `typename`. Se isto for omitido, e um `Handler` for instanciado, por causa da função main() seguinte, tal erro peculiar de compilação é gerado outra vez:

```
#include "handler.h"
#include <vector>
using namespace std;

int main()
{
    vector<int> vi;
    Handler<vector<int> > ph(vi);
}
/*
    Erro Reportado:

handler.h:4: error: syntax error before `;' token
*/
```

Claramente a linha

```
Container::const_iterator d_it;
```

da classe `Handler` causa um problema: é interpretada pelo compilador como um membro estático em vez de um subtipo. Outra vez, o problema é resolvido usando o `typename`:

```
template <typename Container>
class Handler
{
    typename Container::const_iterator d_it;
    ...
};
```

Uma ilustração interessante que o compilador supõe certamente que `X::a` é um membro `a` da classe X é dado pela mensagem de erro quando tentamos compilar main() usando a seguinte execução do construtor de `Handler`:

```
Handler(Container const &container)
:
    d_it(container.begin())
```

```

{
    size_t x = Container::ios_end;
}
/*
    Erro Reportado:

    error: `ios_end' is not a member of type `std::vector<int,
        std::allocator<int> >'
*/

```

Como uma ilustração final considere o que acontece se o modelo da função introduzido no começo desta seção não retorna um valor `Type`, mas um valor `Type::Ambiguous`. Outra vez, um subtipo de um tipo do modelo é consultado, e o `typename` é requerido:

```

template <typename Type>
typename Type::Ambiguous function(Type t)
{
    return t.ambiguous();
}

```

Using `typename` in the specification of a return type is further discussed in section (20.1.2).

In some cases `typename` can be avoided by resorting to a

`typedef`. E.g., `Iterator`, defined using `typedef`, can be used to indicate the specific type:

```

template <typename Container>
class Handler
{
    typedef typename Container::const_iterator Iterator;

    Iterator d_it;
    ...
};

```

20.1.2: Retornando tipos aninhados sob modelos de classes

Considere o seguinte exemplo em que uma classe aninhada, que não depende de um parâmetro do modelo, é definida dentro de uma classe do modelo. Além disso, o membro `Nested()` do modelo da classe retorna um objeto da classe aninhada. Note que a execução do membro da classe (desprezado) está usado. A razão disto é para tornar-se logo evidente.

```

template <typename T>
class Outer
{
    public:
        class Nested

```

```

        {
        };

    Nested nested() const
    {
        return Nested();
    }
};

```

O exemplo acima compila fluentemente: dentro da classe `Outer` não há nenhuma ambigüidade com respeito ao significado do tipo de retorno, `Nested()`s'.

Entretanto, desde que se recomenda para implementar membros em linha e do modelo abaixo de sua interface de classe (veja a seção 6.3.1), agora removemos a implementação da própria interface, e a pusemos abaixo da interface. De repente o compilador recusa compilar nosso membro `Nested()`:

```

template <typename T>
class Outer
{
    public:
        class Nested
        {
        };

        Nested nested() const;
};

template <typename T>
Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}

```

A implementação acima de `Nested()` produz uma mensagem de erro como:

```

error:
    expected constructor, destructor, or type conversion before
    'Outer'.

```

Nos casos como estes o tipo de retorno (`Outer<T>::Nested`) se refere a um subtipo de `Outer<T>` antes que a um membro de `Outer<T>`.

Como regra geral o seguinte permanece verdadeiro: a palavra chave `typename` deve ser usada sempre que um tipo é referenciado para isso é um subtipo de um tipo que seja ele próprio dependente de um tipo de parâmetro de um modelo.

O `typename` diante de `Outer<T>::Nested` remove o erro de compilação e conseqüentemente a execução correta da função `Nested()` que torna-se:

```

template <typename T>
typename Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}

```

20.1.3: Resolução de tipos em membros de classes básicas

Considere o seguinte exemplo de um modelo básico e de uma classe derivada:

```

#include <iostream>

template <typename T>
class Base
{
public:
    void member();
};

template <typename T>
void Base<T>::member()
{
    std::cout << "This is Base<T>::member()\n";
}

template <typename T>
class Derived: public Base<T>
{
public:
    Derived();
};

template <typename T>
Derived<T>::Derived()
{
    member();
}

```

Este exemplo não compilará, e o compilador nos dirá algo como:

```

error: there are no arguments to 'member' that depend on a template
parameter, so a declaration of 'member' must be available

```

A primeira vista, este erro pode causar alguma confusão, desde que com modelos públicos sem classe e membros protegidos de classe básica estão imediatamente disponíveis. Isto também é verdadeiro para modelos da classe, mas somente se o compilador pode compreender o que queremos. Na situação acima, o compilador não pode, já que não sabe o tipo de T, o membro da função membro deve ser iniciado.

Para apreciar porque isto é verdade, considere a situação onde nós definimos uma especialização:

```

template <>
Base<int>::member()
{
    std::cout << "This is the int-specialization\n";
}

```

Since the compiler, when processing the class `Derived`, can't be sure that no specialization will be in effect once an instantiation of `Derived` is called for, it can't decide yet for what type to instantiate `member`, since `member()`'s call in `Derived::Derived()` doesn't require a template type parameter. In cases like these, where no template type parameter is available to determine which type to use, the compiler must be told that it should postpone its decision about the template type parameter to use for `member()` until instantiation time. This can be realized in two ways: either by using this, or by explicitly mentioning the base class, instantiated for the derived class's template type(s). In the following `main()` function both forms are used. Note that with the `int` template type the `int` specialization is used.

```

#include <iostream>

template <typename T>
class Base
{
public:
    void member();
};

template <typename T>
void Base<T>::member()
{
    std::cout << "This is Base<T>::member()\n";
}

template <>
void Base<int>::member()
{
    std::cout << "This is the int-specialization\n";
}

template <typename T>
class Derived: public Base<T>
{
public:
    Derived();
};

template <typename T>
Derived<T>::Derived()
{

```

```

        this->member();
        Base<T>::member();
    }

int main()
{
    Derived<double> d;
    Derived<int> i;
}

/*
    Generated output:
    This is Base<T>::member()
    This is Base<T>::member()
    This is the int-specialization
    This is the int-specialization
*/

```

20.1.4: ::template, .template and ->template

In general, the compiler is able to determine the true nature of a name. As discussed in the previous sections, this is not always the case and the software engineer sometimes has to advise the compiler. The `typename` keyword can often be used to that purpose.

However, `typename` cannot always come to the rescue. While parsing a source, the compiler receives a series of tokens, representing meaningful units of text encountered in the program's source. A token represents, e.g., an identifier or a number. Other tokens represent operators, like `=`, `+` or `<`. It is precisely the last token that may cause problems, as it is used in multiple ways, which cannot always be determined from the context in which the compiler encounters `<`. Sometimes, however, the compiler will know that `<` does not represent the less than operator, as in the situation where a template parameter list follows the keyword `template`, e.g.,

```
template <typename T, int N>
```

Clearly, in this case `<` does not represent a 'less than' operator.

The special meaning of `<` if preceded by `template` forms the basis for the syntactical constructs discussed in this section.

Assume the following class has been defined:

```

template <typename Type>
class Outer
{
    public:
        template <typename InType>

```

```

class Inner
{
    public:
        template <typename X>
        void nested();
};

```

Here a class template Outer defines a nested class template Inner, which in turn defines a template member function.

Next, a class template Usage is defined, offering a member function caller() expecting an object of the above Inner type. E.g., Usage an initial setup for Usage could now have been written as follows:

```

template <typename T1, typename T2>
class Usage
{
    public:
        void fun(Outer<T1>::Inner<T2> &obj);
        ...
};

```

The compiler, however, won't accept this. It interprets Outer<T1>::Inner as a class type, which of course doesn't exist. In this situation the compiler generates an error like:

```
error: 'class Outer<T1>::Inner' is not a type
```

To inform the compiler that in this case Inner itself is a template, using the template type parameter <T2>, the ::template construction is required. This tells the compiler that the next < should not be interpreted as a 'less than' token, but rather as a template type argument. So, the declaration is modified to:

```
void fun(Outer<T1>::template Inner<T2> &obj);
```

But this still doesn't get us where we want to be: after all Inner<T2> is a type, nested under a class template, depending on a template type parameter. Actually, the compiler produces a series of error messages here, one of them being like:

```
error: expected type-name before '&' token
```

which nicely indicates what should be done to get it right: add typename:

```
void fun(typename Outer<T1>::template Inner<T2> &obj);
```

Next, fun() itself is not only just declared, it is implemented as

well. The implementation should call Inner's member nested() function, instantiated for yet another type X. The class template Usage should now receive a third template type parameter, which can be called T3: let's assume it has been defined. To implement fun(), we start out with:

```
void fun(typename Outer<T1>::template Inner<T2> &obj)
{
    obj.nested<T3>();
}
```

However, once again we run into a problem. The compiler once again interprets < as 'less than', and expects a logical expression, having as its right-hand side a primary expression instead of a formal template type.

To tell the compiler in situations like these that <T3> should be interpreted as a type to instantiate nested with, the template keyword is used once more. This time it is used in the context of the member selection operator: by writing .template the compiler is informed that what follows is not a 'less than' operator, but rather a type specification. The function's final implementation becomes:

```
void fun(typename Outer<T1>::template Inner<T2> &obj)
{
    obj.template nested<T3>();
}
```

Instead of value or reference parameters functions may define pointer parameters. If obj would have been defined as a pointer parameter the implementation would use the ->template construction, rather than the .template construction. E.g.,

```
void fun(typename Outer<T1>::template Inner<T2> *ptr)
{
    ptr->template nested<T3>();
}
```

20.2: Template Meta Programming

20.2.1: Values according to templates

In template programming values are preferably represented by enum values. Enums are preferred over, e.g., int const values since enums never have any linkage: they are pure symbolic values with no memory representation.

Consider the situation where a programmer must use a cast, say a

`reinterpret_cast`. A problem with a `reinterpret_cast` is that it is the ultimate way to turn off all compiler checks. All bets are off, and we can write extreme but absolutely pointless `reinterpret_cast` statements, like

```
int value = 12;
ostream &ostr = reinterpret_cast<ostream &>(value);
```

Wouldn't it be nice if the compiler would warn us against such oddities by generating an error message? If that's what we'd like the compiler to do, there must be some way to distinguish madness from weirdness. Let's assume we agree on the following distinction: `reinterpret` casts are never acceptable if the target type represents a larger type than the expression (source) type, since that would immediately result in abusing the amount of memory that's actually available to the target type. In this way we can't allow `reinterpret` cast from `int` to `double` since a `double` is a larger type than an `int`.

The intent is now to create a new kind of cast, let's call it `reinterpret_to_smaller_cast`, which can only be performed if the target type is a smaller type than the source type (note that this exactly the opposite reasoning as used by Alexandrescu (2001), section 2.1).

The following template is constructed:

```
template<typename Target, typename Source>
Target &reinterpret_to_smaller_cast(Source &source)
{
    // determine whether Target is smaller than source
    return reinterpret_cast<Target &>(source);
}
```

At the comment an enum-definition is inserted with a suggestive name, resulting in a compile-time error if the condition is not met. A division by zero is clearly not allowed, and noting that a false value represents a zero value, the condition could be:

```
1 / (sizeof(Target) <= sizeof(Source));
```

The interesting part is that this condition doesn't result in any code at all: it's a mere value that's computed by the compiler while compiling the expression. To transform this into a useful error message the expression is assigned to a descriptive enum value, resulting in, e.g.,

```
template<typename Target, typename Source>
Target &reinterpret_to_smaller_cast(Source &source)
{
    enum
    {
        the_Target_size_exceeds_the_Source_size =
```

```

        1 / (sizeof(Target) <= sizeof(Source))
    };
    return reinterpret_cast<Target &>(source);
}

```

When `reinterpret_to_smaller_cast` is used to cast from `int` to `ostream` an error is produced by the compiler, like:

```

error: enumerator value for 'the_Target_size_exceeds_the_Source_size'
      not integer constant

```

whereas no error is reported if, e.g.,
`reinterpret_to_smaller_cast<int>(cout)` is requested.

In the above example a `enum` was used to compute, compile time, a value that is illegal if an assumption is not met. The creative part is finding an appropriate expression.

`Enum` values are well suited for these situations as they do not consume any memory and their evaluation does not produce any executable code. They can be used to accumulate values too: the resulting `enum` value will then contain a final value, computed by the compiler rather than by code as the next sections illustrate. In general, programs shouldn't do run-time what they can do

compile-time and computing complex calculations resulting in constant values is a clear example of this principle.

20.2.1.1: Converting integral types to types

Another use of values buried inside templates is to 'templatize' simple scalar

`int` values. This is primarily useful in situations where a scalar value (often a `bool` value) is available to select an appropriate member specialization, a situation that will be encountered shortly (section 20.2.2).

Templatizing

integral values is based on the fact that a

class template together with its template arguments represent a type. E.g., `vector<int>` and `vector<double>` are different types.

Templatizing integral values is simply realized: just define a template, it does not have to have any contents at all, but it customarily has the integral values stored as an `enum` value:

```

template <int x>
struct IntType

```

```
{
    enum { value = x };
};
```

Since `IntType` does not have any members, but just the enum value, the `'class'` can be defined as a `'struct'`, saving us from typing `public:..`. Defining the enum value `'value'` allows us to retrieve the value used at the instantiation at no cost in memory: enum values are not variables or data members, and thus have no address. They are mere values.

Using the struct `IntType` is easy: just define an anonymous or named object by specifying a value for its int non-type parameter:

```
int main()
{
    IntType<1> it;
    cout << "IntType<1> objects have value: " << it.value << "\n" <<
        "IntType<2> objects are of a different type "
        "and have values " << IntType<2>().value << endl;
}
```

20.2.2: Selecting alternatives using templates

Being able to make choices is an essential feature of programming languages. If we want to be able to `'program the compiler'` this feature must be present in templates as well. Once again, realize that being able to make choices in templates has nothing to do with run-time execution of programs. The essence of template meta programming is that we're not using or relying on any executable code in our template meta program. Of course, the result will usually be executable code, but the particular code that is produced must be a function of decisions the compiler can make by itself.

Since template (member) functions are only instantiated when they are actually used, we can even define specializations of functions which are mutually exclusive. I.e., it is possible to define a specialization which may be compilable in one situation, but not in another, and a second specialization which is compilable in the other situation, but not in the first situation. This way code can be tailored to the demands of a concrete situation.

A feature like this cannot be realized in code. For example, when designing a generic storage class the software engineer may have the intention to store both value class type objects and objects of polymorphic class types in the storage class. From this point of departure the engineer may conclude that the storage class should contain pointers to objects, rather than objects themselves, and the following code may be conceived of:

```
template <typename Type>
void Storage::add(Type const &obj)
```

```

{
    d_data.push_back(
        d_ispolymorphic ?
            obj.clone()
        :
            new Type(obj)
    );
}

```

The intent is to use the `clone()` member function of the `Type` class if `Type` is a polymorphic class and the standard copy constructor if `Type` is a value class.

Unfortunately, this scheme will normally fail to compile as value classes do not define `clone()` member functions and polymorphic base classes should define

their copy constructor in the class's private section. It doesn't matter to the compiler that `clone()` is never called for value classes and the copy constructor is never called for value type classes: it has some code to compile, and can't do that because of missing members. It's as simple as that.

Template meta programming comes to the rescue. Knowing that template functions are only instantiated when used, we design specializations of our `add()` function, and provide our class `Storage` with an additional (in addition to `Type` itself) template non-type parameter indicating whether we'll use `Storage` for polymorphic or non-polymorphic classes:

```

template <typename Type, bool isPolymorphic>
class Storage
    ...

```

and we simply define overloaded versions of our `add()` member: the one implementing the polymorphic class variant expecting `true` as its argument, the one implementing the value class variant accepting `false` as its argument.

Again we run into a problem: overloading members cannot be based on argument values, only on types. Fortunately there is a way out: in section 20.2.1.1 it was discussed how to convert integral values to types, and knowledge of how to do this now comes in handy. The strategy is to define two overloaded versions: one defining an `IntType<true>` parameter, implementing the polymorphic class and one defining an `IntType<false>` parameter, implementing the polymorphic class. In addition to these overloaded versions of the member function `add()` the member `add()` itself calls the appropriate overloaded member by providing an `IntType` argument, constructed from `Storage`'s template non-type parameter. Here are the implementations:

Declared in `Storage`'s private section:

```

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<true>)
{
    d_data.push_back(obj.clone());
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<false>)
{
    d_data.push_back(new Type(obj));
}

```

Declared in Storage's public section:

```

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj)
{
    add(obj, IntType<isPolymorphic>());
}

```

In the above example making a selection was made possible by converting a primitive value to a type and then (since each concrete primitive value may be used to construct a different type) using these types to define overloaded versions of template member functions one of which is then called from a (public) member using `IntType` to construct the appropriate selector type.

Since template members are only instantiated when used, only one of the overloaded private `add()` members is instantiated. Since the other one is never called compilation errors are prevented.

Some software engineers may have second thoughts when thinking about the `Storage` class using pointers to store copies of value classes. Their argument could be that value class objects can very well be stored by value, rather than by pointer. In those cases we'd like to define the actual type used for storing the values as either value types or pointer types. Situations like these frequently occur in template meta programming and the following struct `IfElse`

may be used to obtain one of two types, depending on a bool selector value:

```

template<bool selector, typename FirstType, typename SecondType>
struct IfElse
{
    typedef FirstType TypeToUse;
};
template<typename FirstType, typename SecondType>
struct IfElse<false, FirstType, SecondType>

```

```
{
    typedef SecondType TypeToUse;
};
```

The IfElse struct uses in its second definition a partial specialization to select the FalseType if the selector is false. In all other cases (i.e., selector == true) the less specific generic case is instantiated by the compiler, defining FirstType as the TypeToUse.

The IfElse struct allows us to templatize structural types: our Storage class may use pointers to store copies of polymorphic class type objects, but values to store value class type objects.

```
template <typename Type, bool isPolymorphic>
class Storage
{
    typedef typename IfElse<isPolymorphic, Type *, Type>::TypeToUse
        DataType;

    std::vector<DataType> d_data;

private:
    void add(Type const &obj, IntType<true>);
    void add(Type const &obj, IntType<false>);
public:
    void add(Type const &obj);
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<true>)
{
    d_data.push_back(obj.clone());
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<false>)
{
    d_data.push_back(obj);
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj)
{
    add(obj, IntType<isPolymorphic>());
}
```

The above example uses IfElse's TypeToUse, which is a type defined by IfElse as either FirstType or SecondType to define the actual data type to be used for Storage's std::vector data type. To prevent long data type definitions Storage defines its own type DataType.

The remarkable result in this example is that the structure of the Storage class's data is now depending on its template parameters. Since the `isPolymorphic == true` situation uses different data types than `t(isPolymorphic == false)` situation, the overloaded private `add()` members can utilize this difference immediately. E.g., `add(Type const &obj, IntType<false>)` now uses direct copy construction to store a copy of `obj` in `d_vector`.

It is also possible to select a type from more than two alternatives. In that case, `IfElse` structs can be nested. Remember that these structs never have any effect on the run-time program, which simply is confronted with the appropriate type, conditional to the type that's associated with the selector value. The following example, defining `MapType` as a map having plain types or pointers for either its key or its value type, illustrates this approach:

```
template <typename Key, typename Value, int selector>
class Storage
{
    typedef typename IfElse<
        selector == 1,                // if selector == 1:
        map<Key, Value>,              // use map<Key, Value>

        typename IfElse<
            selector == 2,            // if selector == 2:
            map<Key, Value *>,        // use map<Key, Value *>

            typename IfElse<
                selector == 3,        // if selector == 3:
                map<Key *, Value>,    // use map<Key *, Value>
                // otherwise:
                map<Key *, Value *> // use map<Key *, Value *>

            >::TypeToUse
        >::TypeToUse
    >::TypeToUse
    MapType;

    MapType d_map;

public:
    void add(Key const &key, Value const &value);
private:
    void add(Key const &key, Value const &value, IntType<1>);
    ...
};

template <typename Key, typename Value, int selector>
inline void Storage<selector, Key, Value>::add(Key const &key,
                                                Value const &value)
{
    add(key, value, IntType<selector>());
}
```



```
}
```

The principle used in the above examples is: if different data types are to be used in class templates, depending on template non-type parameters, an `IfExists` struct can be used to define the appropriate type, and overloaded member functions may utilize knowledge about the appropriate types to optimize their implementations.

Note that the overloaded functions have identical parameter lists as the matching public wrapper function, but add to this parameterlist a specific `IntType` type, allowing the compiler to select the appropriate overloaded version, based on the template's non-type selector parameter.

20.2.3: Templates: Iterations by Recursion

Since there are no variables

in template meta programming, there is no way to implement iteration using templates. However, iterations can always be rewritten as recursions, and since recursions are supported by templates iterations can always be rewritten as (tail) recursions.

The principle to follow here is:

- o Define a specialization implementing the end-condition;
- o Define all other steps using recursion.
- o Store intermediate values as enum values.

Since the compiler will select a more specialized implementation over a more generic one, by the time it reaches the final recursion it will stop the recursion since the specialization will not rely on recursion anymore.

Most readers will be familiar with the recursive implementation of the mathematical 'faculty' operator, indicated by the exclamation mark (!). The faculty operator of value n (so: $n!$) returns the successive products $n * (n - 1) * (n - 2) * \dots * 1$, representing the number of ways n objects can be permuted. Interestingly, the faculty operator is usually defined by a recursive definition:

```
n! = (n == 0) ?  
    1  
    :  
    n * (n - 1)!
```

To compute $n!$ from a template, a template `Faculty` can be defined using a `int n` template non-type parameter, and defining a specialization for the case $n == 0$. The generic implementation uses recursion according to the faculty definition. Furthermore, the `Faculty` template defines an enum value 'value' to contain the its faculty value. Here is the

generic definition:

```
template <int n>
struct Faculty
{
    enum { value = n * Faculty<n - 1>::value };
};
```

Note how the expression assigning a value to `value' uses constant, compiler determinable values: `n` is provided, and `Faculty<n - 1>()` is computed by template meta programming, also resulting in a compiler determinable value. Also note the interpretation of `Faculty<n - 1>::value`: it is the value defined by the type `Faculty<n - 1>`; it's not, e.g., the value returned by an object of that type. There are no objects here, simply values defined by types.

To end the recursion a specialization is required, which will be preferred by the compiler over the generic implementation when its template arguments are present. The specialization can be provided for the value 0:

```
template <>
struct Faculty<0>
{
    enum { value = 1 };
};
```

The `Faculty` template can be used to determine, compile time, the number of permutations of a fixed number of objects. E.g.,

```
int main()
{
    cout << "The number of permutations of 5 objects = " <<
        Faculty<5>::value << "\n";
}
```

Once again, `Faculty<5>::value` is not evaluated run-time, but compile-time. The above statement is therefore run-time equivalent to:

```
int main()
{
    cout << "The number of permutations of 5 objects = " <<
        120 << "\n";
}
```

20.3: Template template parameters

Consider the following situation: a software engineer is asked to design a storage class `Storage` which is able to store data, which may either make and store copies of the data or store the data as received, and which may

either use a vector or a linked list as its underlying storage medium. How should the engineer tackle this request?

The engineer's first reaction could be to develop `Storage` as a class having two data members, one being a list, another being a vector, and to provide the constructor with maybe an enum value indicating whether the data itself or new copies should be stored, using that enum value to set a series of pointers to member functions to activate the appropriate subset of its private member functions, providing public wrapped functions to hide the use of the pointers to members.

Complex, but doable, until the engineer is confronted with a modification of the original question: now the request states that it should also be possible to use -in the case of new copies- a custom-made allocation scheme, rather than the standard `new` operator, and it should also be possible to use yet another type of container, in addition to the vector and list that were already part of the design. E.g., a `t(queue)` could be preferred or maybe even a stack.

It's clear that the approach suggesting to have all functionality provided by the class doesn't scale. The class `Storage` would soon become a monolithic giant which

is hard to understand, maintain, test, and deploy.

One of the reasons why the big, all-encompassing class is hard to deploy and understand is that a well-designed class should

enforce constraints: the design of the class should, by itself, disallow certain operations, violations of which should be detected by the compiler, rather than by a program, crashing or terminating with a fatal error.

Consider the above request. If the class offers both an interface to access the vector data storage and an interface to access the list data storage, then it's likely that the class will offer an overloaded `operator[]` member to access elements in the vector. This member, however, will be syntactically present, but semantically invalid when the list data storage is selected, which doesn't support `operator[]`.

Sooner or later, users of the monolithic all-encompassing class `Storage` will fall into the trap of using `operator[]` even though they've selected the list as the underlying data storage. The compiler won't be able to detect the error, which will only appear once the program is running, leaving users rather than the engineer flabbergasted.

The question remains: how should the engineer proceed, when confronted with the above questions? It's time to introduce policies.

20.3.1: Policy classes - I

A policy defines (in some contexts: prescribes) a particular kind of behavior. In our context a policy class defines a certain part of the class interface, and it may define inner types, member functions and data members.

In the previous section a problem of how to create a class which might use a series of allocation schemes was introduced. These allocation schemes all depend on the actual data type to be used, and so the 'template' reflex should kick in: the allocation schemes should probably be defined as template classes, applying the appropriate allocation procedures to the data type at hand. E.g. (using in-class implementations to save some space), the following three allocation classes could be defined:

- o No special allocation takes place, data is used 'as is':

```
template <typename Data>
class PlainAlloc
{
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out,
                                   PlainAlloc<IData> const &alloc);

    Data d_data;

public:
    PlainAlloc()
    {}
    PlainAlloc(Data data)
    :
        d_data(data)
    {}
    PlainAlloc(PlainAlloc<Data> const &other)
    :
        d_data(other.d_data)
    {}
};
```

- o The second allocation scheme uses the standard new operator to allocate a new copy of the data:

```
template <typename Data>
class NewAlloc
{
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out,
                                   NewAlloc<IData> const &alloc);

    Data *d_data;

public:
    NewAlloc()
    :

```

```

        d_data(0)
    {}
    NewAlloc(Data const &data)
    :
        d_data(new Data(data))
    {}
    NewAlloc(NewAlloc<Data> const &other)
    :
        d_data(new Data(*other.d_data))
    {}
    ~NewAlloc()
    {
        delete d_data;
    }
};

```

- o The third allocation scheme uses the placement new operator (see section 7.1.4), requesting memory from a common pool of bytes (the implementation of the member request(), obtaining the required amount of memory, is left as an exercise to the reader):

```

template<typename Data>
class PlacementAlloc
{
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out,
                                    PlacementAlloc<IData> const &alloc);

    Data *d_data;

    static char s_commonPool[];
    static char *s_free;

public:
    PlacementAlloc()
    :
        d_data(0)
    {}
    PlacementAlloc(Data const &data)
    :
        d_data(new(request()) Data(data))
    {}
    PlacementAlloc(PlacementAlloc<Data> const &other)
    :
        d_data(new(request()) Data(*other.d_data))
    {}
    ~PlacementAlloc()
    {
        d_data->~Data();
    }
private:

```

```

        static char *request();
};

```

The above three classes define policies that may be selected by the user of the class `Storage`, introduced in the previous section. In addition to this, additional allocation schemes could be implemented by the user as well.

In order to be able to apply the proper allocation scheme to the class `Storage` it should also be designed as a class template. The class will also need a template type parameter allowing users to specify the data type.

It would be possible to specify the data type with the specification of the allocation scheme, resulting in code like:

```

template <typename Data, typename Scheme>
class Storage ...

```

and then use `Storage`, e.g., as follows:

```

Storage<string, NewAlloc<string> > storage;

```

However, this implementation is unnecessarily complex, as it requires the user to specify the data type twice. Instead, the allocation scheme should be specified using a third type of template parameter, not requiring the user to specify the data type with the allocation scheme to use. This third type of template parameter (in addition to the well-known template type parameter and template non-type parameter) is the template template parameter.

20.3.2: Policy classes - II: template template parameters

Template

template parameters allow us to specify a class template as a template parameter. By specifying a class template, it is possible to add a certain kind of behavior, called policy to an existing class template.

Consider the class `Storage`, introduced at the beginning of this section, and consider the allocation classes discussed in the previous section. To specify an allocation policy the class `Storage` starts its definition as follows:

```

template <typename Data, template <typename> class Policy>
class Storage ...

```

The second template parameter is the

template template parameter. It contains the following elements:

- o The keyword `template` starts the definition of a template parameter;
- o It is followed, between pointed brackets, a list of template parameters that must be specified for the template parameter. These parameters may be given names, but these names cannot be used in the subsequent template definition, and are therefore usually omitted. On the other hand, providing formal names may help the reader of the template to understand the kinds of templates that may be specified as template parameters.
- o Template parameters must match, in numbers and types (template type parameter, template non-type parameter, template parameter) the template parameters that must be specified for the policy.
- o Following the bracketed list the keyword `class` is provided. In this case, `typename` cannot be used.
- o All parameter values may be provided with default values, as shown by the following example of a hypothetical class template:

```
template
<
    template
    <
        typename = std::string,
        int = 12,
        template
        <
            typename = int
        >
        class Inner = std::vector
    >
    class Policy
>
class Demo
{
    ...
};
```

Since the policy class should be an inherent part of the class under consideration, it is often deployed as a base class. So, `Policy` becomes a base class of `Storage`. Moreover, the policy should operate on the data type to be used with the class `Storage`. Therefore the policy is handed that data type as well. From this we obtain the following setup:

```
template <typename Data, template <typename> class Policy>
class Storage: public Policy<Data>
```

This scheme allows us to use the policy's members when implementing the members of the class `Storage`.

Now the allocation classes do not really offer many useful members: apart from the extraction operator, no immediate access to the data is offered. This can easily be repaired by providing some additional members. E.g., the class `NewAlloc` could be extended with the following operators, allowing access to and modification of stored data:

```
operator Data &()    // optionally add a `const' member too
{
    return *d_data;
}
NewAlloc &operator=(Data const &data)
{
    *d_data = data;
}
```

Other allocation classes can be provided with comparable members.

The next step is to use the allocation schemes in some real code. The following example shows how a storage can be constructed for a data type to be specified and an allocation scheme to be specified. First, define a class `Storage`:

```
template <typename Data, template <typename> class Policy>
class Storage: public std::vector<Policy<Data> >
{
};
```

That's all there is. All required functionality is offered by the vector base class, while the policy is 'factored into the equation' via the template template parameter. Here's an example of its use:

```
Storage<std::string, NewAlloc> storage;

copy(istream_iterator<std::string>(cin), istream_iterator<std::string>(),
     back_inserter(storage));

cout << "Element index 1 is " << storage[1] << endl;
storage[1] = "hello";

copy(storage.begin(), storage.end(),
     ostream_iterator<NewAlloc<std::string> >(cout, "\n"));
```

Following the construction of a `Storage` object, the STL `copy()` function can be used in combination with the `back_inserter` iterator to add some data to storage. Its elements can be both accessed and modified directly using the index operator, and then `NewAlloc<std::string>` objects are inserted into `cout`, again using the STL `copy()` algorithm.

Interestingly, this is not the end of the story. After all, the intention was to create a class allowing us to specify the storage type as well. What if we don't want to use a vector, but instead would like to use a list?

It's easy to change Storage's setup so that a completely different storage type can be used on request, say a list or a deque. To realize this, the storage class is parameterized as well, again using a template template parameter, that could be given a default value too, as shown in the following redefinition of Storage:

```
template <typename Data, template <typename> class Policy,
          template <typename> class Container =
          std::vector>
class Storage: public Container< Policy<Data> >
{
};
```

The earlier example in which a Storage object was used can be used again, without any modifications, for the above redefinition. It clearly can't be used with a list container, as the list lacks operator[]. But that's immediately recognized by the compiler, producing an error if an attempt is made to use operator[] on, e.g., a

list\ (A complete example showing the definition of the allocation classes and the class Storage as well as its use is provided in the Annotation's distribution in the file yo/templateapp/examples/storage.cc.).

20.3.2.1: The destructor of Policy classes

In the previous section policy classes are used as base classes of template classes resulting in the interesting construction that a class template actually serves as a base class of a derived class

Since a policy class may act as a base class, it is thinkable that a pointer or reference to a policy class is used to point or refer to the derived class using the policy.

This situation, although legal, should be avoided for various reasons:

- o Destruction of a derived class object using the base class's destructor requires the implementation of a virtual destructor;
- o A virtual destructor introduces overhead to a class that normally has no data members, but merely defines behavior: suddenly a vtable is required as well as a data member: a pointer to the vtable;
- o Virtual member functions somewhat reduce the efficiency of code; thus virtual member functions, using dynamic polymorphism, somewhat

counteract the static polymorphism offered by templates;

- o Virtual member functions in templates may result in code bloat: once an instantiation of a class's member is required, the class's vtable and all its virtual members must be implemented too.

To avoid these drawbacks, it is good practice to prevent using references or pointers to policy classes to refer or point to derived class objects.

This is accomplished by providing policy classes with

nonvirtual protected destructors. Since the destructor is non-virtual there is no implementation penalty in reduced efficiency or memory overhead, and since it is protected users cannot refer to classes derived from the policy class using a pointer or reference to the policy class.

20.3.3: Structure by Policy

Policy classes usually define behavior, not structure. I.e., policy classes are used to parameterize some aspect of the behavior of classes that are derived from them. However, different policies may very well imply the use of different data members. Thus a policy class may be used to define both behavior and structure.

By providing a well-defined interface a class derived from a policy class may define member specializations using the different structures of policy classes to their advantage. For example, a plain pointer-based policy class could offer its functionality by resorting to C-style pointer juggling, whereas a vector-based policy class could use the vector's members directly.

In this situation a generic class template `Size` could be designed expecting a container-like policy, using features commonly found in containers, defining the data (and hence the structure) of the container specified in the policy. E.g.:

```
template <typename Data, template <typename> class Container>
struct Size: public Container<Data>
{
    size_t size()
    {
        // relies on the container's `size()'
        // note: can't use `this->size()'
        return Container<Data>::size();
    }
};
```

Next, a specialization can be defined to accomodate the specifics of a much simpler storage class using, e.g., plain pointers (the implementation capitalizes on first and second, data members of `std::pair`).

Cf. the end of this section):

```
template <typename Data>
struct Size<Data, Plain>: public Plain<Data>
{
    size_t size()
    {
        // relies on pointer data members
        return this->second - this->first;
    }
};
```

Depending on the intentions of the template's author other members could be implemented as well.

To use the above templates for real, a generic wrapper class can now be constructed: depending on the actual storage type that is used (e.g., a `std::vector` or some plain storage class) it will use the matching `Size` template to define its structure:

```
template <typename Data, template <typename> class Store>
class Wrapper: public Size<Data, Store>
{};
```

The above classes could now be used as follows (en passant showing an extremely basic `Plain` class):

```
#include <iostream>
#include <vector>

template <typename Data>
struct Plain: public std::pair<Data *, Data *>
{};

int main()
{
    Wrapper<int, std::vector> wiv;
    std::cout << wiv.size() << "\n";

    Wrapper<int, Plain> wis;
    std::cout << wis.size() << "\n";
}
```

The `wiv` object now defines vector-data, the `wis` object merely defines a `std::pair` object's data members.

20.4: Trait classes

Scattered over the `std` namespace

trait classes are found. E.g., most C++ programmers will have seen the compiler mentioning ``std::char_traits<char>'` when performing an illegal operation on `std::string` objects, as in `std::string s(1)`.

Trait classes are used to make compile-time decisions about types. Traits classes allow the software engineer to apply the proper code to the proper data type, be it a pointer, a reference, or a plain value, all maybe in combination with `const`. Moreover, the specification of the particular type of data to be used does not have to be made by the template writer, but can be inferred from the actual type that is specified (or implied) when the template is used.

Trait classes allow the software engineer to develop a template `<typename Type1, typename Type2, ...>` without the need to specify many specializations covering all combinations of, e.g., values, (const) pointers, or (const) references, which would soon result in an unmaintainable exponential explosion of template specializations (e.g., allowing these five different actual types for each template parameter already results in 25 combinations when two template type parameters are used: each must be covered by potentially different specializations).

Having available a trait class, the actual type can be inferred compile time, allowing the compiler to deduct whether or not the actual type is a pointer, a pointer to a member, a const pointer, and make comparable deductions in case the actual type is, e.g., a reference type. This in turn allows us to write templates that define types like `argument_type`, `first_argument_type`, `second_argument_type` and `result_type`, which are required by several generic algorithms (e.g., `count_if()`).

A trait class usually performs no behavior. I.e., it has no constructor and no members that can be called. Instead, it defines a series of types and enum values that have certain values depending on the actual type that is passed to the trait class template. The compiler uses one of a set of available specializations to select the one appropriate for an actual template type parameter.

The generic point of departure when defining a trait template is a plain vanilla struct, defining the characteristics of a plain value type, e.g., an `int`. This sets the stage for specific specializations, modifying the characteristics for any other type that could be specified for the template.

To make matters concrete, assume the intent is to create a trait class `BasicTraits` telling us whether a type is a plain value type, a pointer type, or a reference type (all of which may or may not be `const` types).

Moreover, whatever the actual type that's provided, we want to be able to determine the 'plain' type (i.e., the type without any modifiers, pointers or references), the 'pointer type' and the 'reference type', allowing us to define in all cases, e.g., a reference to its basic type, even though we passed a `const` pointer to that type.

Our point of departure, as mentioned, is a plain struct defining the required parameter. E.g., something like:

```
template <typename T>
struct Basic
{
    typedef T Type;
    enum
    {
        isValue = true,
        isPointer = false,
        isConst = false
    };
};
```

However, often decisions about types can be made using constant logical expressions. Note that the above definition does not contain a `'isReference'` enumeration value. Such a value is not required as it is implied by the expression `not isPointer and not isValue`.

Although some conclusions can be drawn by combining various enum values, it is good practice to provide a full implementation of trait classes, not requiring its users to construct these logical expressions themselves. Therefore, the basic decisions in a trait class are usually made by a

nested trait class,
leaving the task of creating appropriate logical expressions to a
surrounding trait class.

So, the struct `Basic` defines the generic form of our inner trait class. Specializations handle specific details. E.g., a pointer type is recognized by the following specialization:

```
template <typename T>
struct Basic<T *>
{
    typedef T Type;
    enum
    {
        isValue = false,
        isPointer = true,
        isConst = false
    };
};
```

whereas a pointer to a const type is matched with the next specialization:

```
template <typename T>
struct Basic<T const *>
```

```

{
    typedef T Type;
    enum
    {
        isValue = false,
        isPointer = true,
        isConst = true
    };
};

```

Several other specializations should be defined: e.g., recognizing const value types or reference types. Eventually all these specializations wind up being nested structs of an outer class `BasicTraits`, offering the public traits class interface. The outline of the outer trait class is:

```

template <typename TypeParam>
class BasicTraits
{
    // Define (specializations) of the template 'Base' here

public:
    typedef typename Basic<TypeParam>::Type ValueType;
    typedef ValueType *PtrType;
    typedef ValueType &RefType;

    enum
    {
        isValueType = Basic<TypeParam>::isValue,
        isPointerType = Basic<TypeParam>::isPointer,
        isReferenceType = not Basic<TypeParam>::isPointer and
                           not Basic<TypeParam>::isValue,
        isConst = Basic<TypeParam>::isConst
    };
};

```

A trait class template can be used to obtain the proper type, irrespective of the template type argument provided, or it can be used to select the proper specialization, depending on, e.g., the const-ness of a template type. The following statements serve as an illustration:

```

cout << BasicTraits<int>::isPointerType << " " <<
      BasicTraits<int *>::isPointerType << " " <<
      BasicTraits<int>::isConst << " " <<
      BasicTraits<int>::isReferenceType << " " <<
      BasicTraits<int &>::isReferenceType << " " <<
      BasicTraits<int const>::isConst << " " <<
      BasicTraits<int const *>::isPointerType << " " <<
      BasicTraits<int const *>::isConst << " " <<

```

```

endl;

BasicTraits<int *>::ValueType value = 12;
int *otherValue = &value;
cout << *otherValue << endl;

```

20.4.1: Distinguishing class from non-class types

In the previous section the `TypeTrait` trait class was developed. Using specialized versions of a nested struct `Type` modifiers, pointers, references and values could be distinguished.

Knowing whether a type is a class type or not (e.g., the type represents a primitive type) could also be a useful bit of knowledge to a template developer. E.g, the class template developer might define a specialization for a member knowing the template's type parameter is a class type (maybe using some member function that should be available) and another specialization for non-class types.

This section addresses the question how a trait class can distinguish class types from non-class types.

In order to distinguish classes from non-class types a distinguishing feature that can be used compile-time must be found. It may take some thinking to find such a distinguishing characteristic, but a good candidate eventually is found in the pointer to member syntactical construct, which is available only for classes. Using the pointer to member construct as the distinguishing characteristic, we now look for a construction which uses the pointer to member if available, and does something else if the pointer to member construction is not available.

Note again the rule of thumb that works so well for template meta programming: define a generic situation, and then specialize for the situations you're interested in. It's not a trivial task to apply this rule of thumb here: how can we distinguish a pointer to a member from 'a generic situation', not being a pointer to a member? Fortunately, such a distinction is possible: a function template can be provided with a parameter which is a pointer to a member function (defining the 'specialization' case), and another function template can be defined so that it accepts any argument. The compiler will then select the latter function in all situations but those in which the provided type is actually a class type, and thus a type which may support a pointer to a member.

Realize that the compiler will not call the functions: we're talking compile-time here, and all the compiler does is to select the appropriate function, in order to be able to evaluate a constant expression (defining the value of, e.g, the enum value

```
isClass).
```

So, one function template will be something like:

```
template <typename ClassType>
static (returntype) fun(void (ClassType::*)());
```

The question about what the return type should be will be answered shortly. Arbitrarily the function's parameter defines a pointer to a member returning void. Note that there's no need for such a function to exist for the concrete class-type that's specified with the traits class, since all the compiler will do is select this function if a class-type was provided to the trait class in which fun() will be nested. In line with this: fun() is only declared, not defined. Furthermore note that fun() is declared as a static member of the trait class, so that there's no need for an actual object when fun() is called.

So far for the class-types. What about the non-class types? For those types a (generic) alternative must be found, one the compiler will select when the actual type is not a class type. Again, the language offers a 'worst case' solution in the

ellipsis parameter list. The ellipsis is a final resort the compiler may turn to if everything else fails. It's not only used to define (the in

C++ deprecated) functions with variable number of arguments, but it's also used to define the catch-all exception catch clause. Therefore, the 'generic case' can be defined as follows:

```
template <typename NonClassType>
static (returntype) fun(...);
```

Note that it would be an error to define the generic alternative as a function expecting an int. The compiler, when confronted with alternatives, will favor the simplest, most specified alternative over a more complex, generic one. So, when providing fun() with an argument it will select int when possible, given the nature of the used argument.

The question now becomes: what argument can be used for both a pointer to a member and the generic situation? Actually, there is such a 'one size fits all' argument: 0. The value 0 can be used as argument value to initialize not only primitive types, but also to initialize pointers and pointers to members. Therefore, fun will be called as fun<Type>(0), with Type being the template type parameter of the trait class. Here, Type must be specified since the compiler will not be able to determine fun's template type parameter when fun(...) is selected.

Now for the return type: the return type cannot be a simple value (like true or false). When using a simple value the isClass enum value cannot be defined, since

```
vern(
```



```
enum { isClass = fun<Type>(0) } ;
    )
```

needs to be evaluated to obtain fun's return value, which is clearly not possible as enum values must be determined compile-time.

To allow a compile-time definition of isClass's value the solution must be sought in an expression that discriminates between fun<Type>(...) and fun<Type>(void (Type::*)()). In situations like these sizeof is our tool of choice. The sizeof operator is evaluated compile-time, and so by defining return types that differ in their sizes it is possible to discriminate compile-time among the two fun() alternatives.

The char type is by definition a type having size 1. By defining another type containing two consecutive char values a bigger type is obtained. Now char [2] is not a type, but char[2] can be defined as a data member of a struct which will thus have a size exceeding 1. E.g.,

```
struct Char2
{
    char data[2];
};
```

Char2 can be defined as a nested type within our traits class, and the two fun declarations become:

```
template <typename ClassType>
static Char2 fun(void (ClassType::*)());

template <typename NonClassType>
static char fun(...);
```

This, in turn enables us to specify an expression that can be evaluated compile time, allowing the compiler to determine isClass's value:

```
enum { isClass = sizeof(fun<Type>(0)) == sizeof(Char2) };
```

It's interesting to realize that no fun() function template ever makes it to the initialization stage, but that the compiler nonetheless is able to infer what fun's return type will be, given a concrete template type argument. This inference is then used by the compiler to determine the truth of an expression, in turn enabling the compiler to compute the required compile-time constant value isClass, allowing us to determine whether a certain type is or is not a class type. Marvelous!

20.5: More conversions to class types

20.5.1: Types to types

Although class templates may be partially specialized, function templates may not. At times that can be annoying. Assume a function template is available implementing a certain unary operator that could be used with the transform (cf. section 17.4.63) generic algorithm:

```
template <typename Return, typename Argument>
Return chop(Argument const &arg)
{
    return Return(arg);
}
```

Furthermore assume that if Return is `std::string` then the specified implementation should not be used. Rather, with `std::string` a second argument `l` should always be provided (e.g., if Argument is a C++ string, a `std::string` is returned holding a copy of the function's argument, except for the argument's first character, which is chopped off).

Since `chop()` is a function, it is not possible to use a partial specialization. So it is not possible to specialize for `std::string` as attempted in the following erroneous implementation:

```
template <typename Argument>
std::string chop<std::string, Argument>(Argument const &arg)
{
    return string(arg, 1);
}
```

it is possible to use overloading, though. Instead of using partial specializations overloaded function templates could be designed:

```
template <typename Return, typename Argument>
Return chop(Argument const &arg, Argument )
{
    return Return(arg);
}

template <typename Argument>
std::string chop(Argument const &arg, std::string )
{
    return string(arg, 1);
}
```

This way it is possible to distinguish the two cases, but at the expense of a more complex function call (e.g., maybe requiring the use of the `bind2nd()` binder (cf. section 17.1.4)

to bind the second argument to a fixed value) as well as the need to provide a (possibly expensive to construct) dummy argument to allow the compiler to choose among the two overloaded function templates.

Alternatively, overloaded versions could use the `IntType` template (cf. section 20.2.1.1) to select the proper overloaded version. E.g., `IntType<0>` could be defined as the type of the second argument of the first overloaded `chop()` function, and `IntType<1>` could be used for the second overloaded function. From the point of view of program efficiency this is an attractive option, as the provided `IntType` objects are extremely lightweight: they contain no data at all. But there's also an obvious disadvantage: there is no intuitively clear association between on the one hand the `int` value used and on the other hand the intended type.

In situations like these it is attractive to use another lightweight solution. Instead of using an arbitrary `int`-to-type association, an intuitively clear and automatic type-to-type association is used. The struct `TypeType` is a lightweight type wrapper, much like `IntType` is a lightweight wrapper around an `int`. Here is its definition:

```
template <typename T>
struct TypeType
{
    typedef T Type;
};
```

This too is a lightweight type as it doesn't have any data fields either. `TypeType` allows us to use a natural type association for `chop()`'s second argument. E.g, the overloaded functions can now be defined as follows:

```
template <typename Return, typename Argument>
Return chop(Argument const &arg, TypeType<Argument> )
{
    return Return(arg);
}

template <typename Argument>
std::string chop(Argument const &arg, TypeType<std::string> )
{
    return std::string(arg, 1);
}
```

Using the above implementations any type can be specified for `Result`. If it happens to be a `std::string` the correct overloaded version is automatically selected. E.g.,

```
template <typename Result>
Result chopper(char const *txt)
{
    return chop(std::string(txt), TypeType<Result>());
}
```

Using `chopper()`, the following statement will produce the text ``ello world'`:

```
cout << chopper<string>("hello world") << endl;
```

20.5.2: An empty type

At times (cf. section 20.6) an empty

`struct` is a useful little tool. It can be used as a type acting analogously to the ASCII-Z (final 0-byte) in C-strings or as a 0-pointer to indicate the end of a linked list. Its definition is simply:

```
struct NullType
{};
```

20.5.3: Type convertability

In what situations can a type `T` be used as a ``stand in'` for another type `U`? Since C++ is a strongly typed language the answer is surprisingly simple: `T`s can be used instead of `U`s if a `T` is accepted as argument in cases where `U`s are requested.

This reasoning is behind the following class which can be used to determine whether a type `T` can be used where a type `U` is expected. The interesting part, however, is that no code is actually generated or executed: all decisions can be made by the compiler.

In the second part of this section it will be shown how the code developed in the first part can be used to detect whether a class `B` is a base class of another class `D`. The code developed here closely follows the example provided by Alexandrescu (2001, p. 35).

First, a function is conceived of that will accept a type `U` to which an alternative type `T` will be compared. This function returns a value of the as yet unspecified type `Convertible`:

```
Convertible test(U const &);
```

The function `test()` is not implemented; it is merely declared. The idea is that if a type `T` can be used instead of a type `U` it can be passed as argument to the above `test()` function.

If `T` cannot be used where a `U` is expected, then the above function will not be used by the compiler. Of course, getting a compiler error is not the kind of ``answer'` we're looking for and so the next question is what

alternative we can offer to the compiler in cases where a T cannot be used as argument to the above function.

C (and C++) offer a very general parameter list, a parameter list that will always be considered acceptable. This parameter list consists of the ellipsis

which actually is the worst situation the compiler may encounter: if everything else fails, then the function defining an ellipsis as its parameter list is selected. Normally that's not a productive and type-safe alternative, but in the current situation it is exactly what is needed. When confronted with two alternative function calls, one of which defines the ellipsis parameter list, the compiler will only select the function defining the ellipsis if the alternative(s) can't be used. So we declare an alternative function test(), having the ellipsis as its parameter list, and returning another type, e.g., NotConvertible:

```
NotConvertible test(...);
```

Now, if code passes a value of type T to the test function, the return type will be Convertible if T can be converted to U, and NotConvertible if conversion is not possible.

Two problems still need to be solved: how do we obtain a T argument? The problems here are firstly, that it might not be possible to define a T, as a type T might hide all its constructors and secondly, how can the two return values be distinguished?

Although it might be impossible to construct a T object, it is fortunately not necessary to construct a T. After all, the intent is to decide compile time

whether a type is convertible to another type and not actually to construct such a T value. So another function is declared:

```
T makeT();
```

This mysterious function has the magical power of enticing the compiler into thinking that a T object will come out of it. So, what will happen when the compiler is shown the following code:

```
test(makeT())
```

If T can be converted to U the first function Convertible test(U const &) will be selected by the compiler; otherwise the function NotConvertible test(...) will be selected.

If it is now possible to distinguish Convertible from NotConvertible compile-time then it is possible to determine whether T is convertible to U.

Since Convertible and NotConvertible are values, their sizes are known. If these sizes differ, then the sizeof operator can be used to distinguish the two types; hence it is possible to determine which test() function was selected and hence it is known whether T can be converted to U or not. E.g., if the following expression evaluates as true T is convertible to U:

```
sizeof(test(makeT())) == sizeof(Convertible);
```

The size of a char is well known. By definition it is 1. Using a typedef Convertible can be defined as a synonym of char, thus having size 1. Now NotConvertible must be defined so that it has a different type. E.g.,

```
struct NotConvertible
{
    char array[2];
};
```

Note there that a simple typedef char NotConvertible[2] does not work: functions cannot return arrays, but they can return arrays embedded in a structs.

The above can be wrapped up in a template class, having two template type parameters:

```
template <typename T, typename U>
class Conversion
{
    typedef char Convertible;
    struct NotConvertible
    {
        char array[2];
    };

    static T makeT();
    static Convertible test(U const &);
    static NotConvertible test(...);

    public:
        enum { exists = sizeof(test(makeT())) == sizeof(Convertible) };
        enum { sameType = 0 };
};

template <typename T>
class Conversion<T, T>
{
    public:
        enum { exists = 1 };
        enum { sameType = 1 };
};
```

```
};
```

The above class never results in any run-time execution of code. When used, it merely defines the values 1 or 0 for its exist enum value, depending whether the conversion exists or not. The following example writes 1 0 1 0 when run from a main() function:

```
cout <<
    Conversion<ofstream, ostream>::exists << " " <<
    Conversion<ostream, ofstream>::exists << " " <<
    Conversion<int, double>::exists << " " <<
    Conversion<int, string>::exists << " " <<
    endl;
```

20.5.3.1: Determining inheritance

Now that Conversion has been defined it's easy to determine whether a type Base is a (public) base class of a type Derived. To determine inheritance convertability of (const) pointers is examined. Derived const * can be converted to Base const * if:

- o Both types are identical;
- o Base is a public and unambiguous base class of Derived;
- o and also, but usually not intended: if Base is void.

Preventing the latter, inheritance is determined by inspecting Conversion<Derived const *, Base const *>::exists:

```
#define BASE_1st_DERIVED_2nd(Base, Derived) \
    (Conversion<Derived const *, Base const *>::exists && \
     not Conversion<Base const *, void const *>::sameType)
```

If code should not consider a class to be its own base class, then the following strict test is possible, which adds a test for type-equality:

```
#define BASE_1st_DERIVED_2nd_STRICT(Base, Derived) \
    (BASE_1st_DERIVED_2nd(Base, Derived) && \
     not Conversion<Base const *, Derived const *>::sameType)
```

The following example writes 1: 0, 2: 1, 3: 0, 4: 1, 5: 0 when run from a main() function:

```
cout << "\n" <<
    "1: " << BASE_1st_DERIVED_2nd(ofstream, ostream) << ", " <<
    "2: " << BASE_1st_DERIVED_2nd(ostream, ofstream) << ", " <<
```

```

"3: " << BASE_1st_DERIVED_2nd(void, ostream) << ", " <<
"4: " << BASE_1st_DERIVED_2nd(ostream, ostream) << ", " <<
"5: " << BASE_1st_DERIVED_2nd_STRICT(ostream, ostream) << " " <<
endl;

```

20.6: Template TypeList processing

This section serves two purposes. On the one hand it illustrates capabilities of the various meta-programming capabilities of templates, which can be used as a source for inspiration when developing your own templates. On the other hand, it culminates in a concrete example, showing some of the power template meta-programming has.

This section itself was inspired by Andrei Alexandrescu's (2001) book

Modern C++ design, and much of this section's structure borrows from Andrei's coverage of typelists.

A typelist is a very simple struct: like a `std::pair` it consists of two elements, although in this case the typelist does not contain data members, but type definitions. It is defined as follows:

```

template <typename First,  typename Second>
struct TypeList
{
    typedef First Head;
    typedef Second Tail;
};

```

The typelist allows us to store any number of types, using a recursive definition. E.g., the three types `char`, `short`, `int` may be stored as follows:

```
TypeList<char, Typelist<short, int> >
```

Although this is a possible representation, usually `NullType` (cf. section 20.5.2) is used as the final type, acting comparably to a 0-pointer. Using `NullType` the above three types are represented as follows:

```

TypeList<char,
    TypeList<short,
        TypeList<int, NullType> > >

```

This way to represent lists of types may be accepted by the compiler, but usually not as easily by programmers, who frequently have a hard time putting in the right number of parentheses. Alexandrescu suggest to ease the burden by defining a series of macros, even though macros are generally deprecated

in C++. The TYPELIST macros suggested by Alexandrescu allow us to

define typelists for varying numbers of types, and they are easily expanded if accommodating larger numbers of types is necessary. Here are the definitions of the first five TYPELIST macros:

```
#define TYPELIST_1(T1)                TypeList<T1, NullType >
#define TYPELIST_2(T1, T2)           TypeList<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3)       TypeList<T1, TYPELIST_2(T2, T3) >
#define TYPELIST_4(T1, T2, T3, T4)   TypeList<T1, TYPELIST_3(T2, T3, T4) >
#define TYPELIST_5(T1, T2, T3, T4, T5) TypeList<T1, TYPELIST_4(T2, T3, T4, T5) >
```

Note the recursive nature of these macro definitions: recursion is how template meta programs perform iteration and in the upcoming sections implementations heavily depend on recursion. With all solutions to the problems covered by the coming sections a verbal discussion is provided explaining the philosophies that underlie the recursive implementations.

Of course, even here macros are ugly. The
macro

processor will be confused if a type is somewhat complex, like `Wrap<HEAD, idx>`. Fortunately situations like these can be prevented using a simple typedef. E.g., `typedef Wrap<HEAD, idx> HEADWRAP` and then using `HEADWRAP` instead of the full type definition.

20.6.1: The length of a TypeList

To obtain the length of a typelist the following algorithm can be used:

- o If the typelist is empty, its size is zero
- o If the typelist is non-empty, its size equals 1 plus the size of its tail.

Note how recursion is used to define the length of a typelist. In 'normal' C++ code this recursion could be implemented as well, e.g., to determine the length of a plain C (ascii-Z) string, resulting in something like:

```
size_t c_length(char const *cp)
{
    return *cp == 0 ? 0 : 1 + c_length(cp + 1);
}
```

In the context of template meta programming the alternatives that are used to execute or terminate recursion are never written in one implementation, but instead specializations

are used: each specialization implements an alternative.

The length of a typelist will be determined by a struct ListSize ordinarily expecting a typelist as its template type parameter. It's merely declared, since it turns out that only its specializations are required:

```
template <typename TypeList>
struct ListSize;
```

Following the above algorithm specializations are now constructed:

- o If the ListSize's type is empty (i.e., a NullType), its size is 0:

```
template <>
struct ListSize<NullType>
{
    enum { size = 0 };
};
```

- o Otherwise, its size will be 1 plus the size of the tail of a TypeList:

```
template <typename Head, typename Tail>
struct ListSize<TypeList<Head, Tail> >
{
    enum { size = 1 + ListSize<Tail>::size };
};
```

That's all. The size of any typelist can now easily be determined. E.g., assuming all required headers (templates, iostream) have been included then the following statement will (of course) display the value 3:

```
std::cout << ListSize<TYPELIST_3(int, char, bool)>::size << "\n";
```

20.6.2: Searching a TypeList

To determine whether a type (called the searchtype below) is present in a given typelist, an algorithm is used that will either define `index' -1 (if the searchtype is not an element of the typelist) or it will define `index' as the index of the first occurrence of the searchtype in the typelist. The following algorithm is used:

- o If the typelist is equal to NullType, define `index' as -1;
- o If the typelist's head element equals the searchtype, define `index' as 0;

- o Otherwise define `index' as follows:
 - o If searching the searchtype in the typelist's tail results in a index -1, then searchtype is not an element of the typelist, and the (current) index will be set to -1 as well;
 - o Otherwise, searchtype was found in the typelist's tail, and the current index will be set to 1 + the index obtained for searchtype on the typelist's tail.

The implementation again sets out with the declaration of a struct: ListSearch expects two template type parameters: searchtype's type and a typelist:

```
template <typename SearchType, typename TypeList>
struct ListSearch;
```

Next, specializations are defined implementing the above alternatives:

- o If the typelist is empty, define `index' as -1:

```
template <typename SearchType>
struct ListSearch<SearchType, NullType>
{
    enum { index = -1 };
};
```

- o If the typelist's head element equals the searchtype, define `index'

as 0. Note how the test is performed by specifying SearchType twice:

```
template <typename SearchType, typename Tail>
struct ListSearch<SearchType, TypeList<SearchType, Tail> >
{
    enum { index = 0 };
};
```

- o Otherwise define `index' as either -1 (searchtype wasn't found in the typelist's tail) or as 1 + the index obtained from searching the typelist's tail. Note that the implementation uses a

private enum value tmp to store the index value obtained from searching the typelist's tail for searchtype:

```
template <typename SearchType, typename Head, typename Tail>
class ListSearch<SearchType, TypeList<Head, Tail> >
```

```

{
    enum { tmp = ListSearch<SearchType, Tail>::index } ;
    public:
        enum { index = tmp == -1 ? -1 : 1 + tmp };
};

```

Assuming all required headers have been included, the following example shows how ListSearch can be used:

```

int main()
{
    std::cout << ListSearch<char, TYPELIST_2(int, char)>::index << "\n";
}

```

20.6.3: Selecting from a TypeList

Next the selection of a type from a typelist given its index will be discussed. This is the inverse operation from obtaining the index of a 'searchtype', as covered by section 20.6.2.

Rather than defining an enum value, the current algorithm should define a type equal to the type at a given index position. If the type does not exist, the typedef can be made a synonym of NullType since NullType cannot appear in a typelist.

The following algorithm is used (the implementation of the parts is provided immediately following the descriptions of the algorithm's steps):

- o The foundation of the algorithm is provided by a declaration of a struct TypeAt, expecting an index and a typelist:

```

template <int index, typename Typelist>
struct TypeAt;

```

- o If the typelist equals NullType define the return type as NullType as well:

```

template <int index>
struct TypeAt<index, NullType>
{
    typedef NullType Type;
};

```

- o If the search index equals 0, define the return type as the typelist's head:

```

template <typename Head, typename Tail>
struct TypeAt<0, TypeList<Head, Tail> >
{
    typedef Head Type;
};

```

o Otherwise, define the return type as the return type of the type at offset index - 1 in the typelist's tail. Note the typename following typedef: it is required as the defining type's result type is a nested type:

```

template <int index, typename Head, typename Tail>
struct TypeAt<index, TypeList<Head, Tail> >
{
    typedef typename TypeAt<index - 1, Tail>::Type Type;
};

```

Assuming all required headers have been included, the following example shows how ListSearch can be used:

```

int main()
{
    typedef TYPELIST_3(int, char, bool) list3;
    enum { test = 2 };

    std::cout <<
        (ListSearch<TypeAt<test, list3>::result, list3>::index == -1 ?
         "Illegal Index\n"
         :
         "Index represents a valid type\n");
}

```

20.6.4: Appending to a TypeList

The question of how to add an element to a typelist is handled using the same rule of thumb as used for answering the previous questions: design a recursive algorithm and implement the recursion through specializations.

To append a new type to a typelist, the following algorithm can be used:

- o The basic template is a struct expecting a typelist and a type to add to the typelist:

```

template <typename TypeList, typename NewType>
struct Append;

```

- o Adding NullType:

- o If the type to add is NullType, and the original type is NullType, the result itself is the NullType:

```
template <>
struct Append<NullType, NullType>
{
    typedef NullType TList;
};
```

Note that the simple alternative:

```
template <typename TypeList>
struct Append<TypeList, NullType>
{
    typedef TypeList Result;
};
```

is not a good idea, as it will match all types that are offered as the template's first template type parameter. E.g., `Append<int, NullType>` would be accepted, but would certainly not result in a `TypeList`.

- o When attempting to append NullType to an existing TypeList, leave the TypeList as-is:

```
template <typename Head, typename Tail>
struct Append<TypeList<Head, Tail>, NullType>
{
    typedef TypeList<Head, Tail> TList;
};
```

- o Appending other types than NullType:

- o If the typelist itself is NullType, the final typelist consists of the typelist containing the new type:

```
template <typename NewType>
struct Append<NullType, NewType>
{
    typedef TYPELIST_1(NewType) TList;
};
```

- o Otherwise, the final typelist consists of the head of the initial typelist and the typelist resulting from appending the new type to the initial typelist's tail:

```
template <typename Head, typename Tail, typename NewType>
```

```

struct Append<TypeList<Head, Tail>, NewType>
{
    typedef TypeList<Head, typename Append<Tail, NewType>::TList>
        TList;
};

```

Once again: note that `typename` is required in front of `Append` (cf. section 20.1.1)

20.6.5: Erasing from a `TypeList`

The opposite from adding, erasing can simply be accomplished as well. Here is the algorithm, erasing the first occurrence of a type to erase from a `typelist`:

- o The basic template is a struct expecting a `typelist` and a type to erase from the `typelist`:

```

template <typename TypeList, typename EraseType>
struct Erase;

```

- o If the `typelist` itself is `NullType`, there's nothing to erase, and `NullType` is the result:

```

template <typename EraseType>
struct Erase<NullType, EraseType>
{
    typedef NullType Result;
};

```

- o If the `typelist`'s head equals the type to erase, then the result is the `typelist`'s tail:

```

template <typename EraseType, typename Tail>
struct Erase<TypeList<EraseType, Tail>, EraseType>
{
    typedef Tail Result;
};

```

- o Otherwise, the result is the `typelist`'s head and the result obtained after erasing the type to be erased from the `typelist`'s tail:

```

template <typename Head, typename Tail, typename EraseType>
struct Erase<TypeList<Head, Tail>, EraseType>

```

```

    {
        typedef TypeList<Head,
                        typename Erase<Tail, EraseType>::Result> Result;
    };
//
//ERASEALL
template <typename TypeList, typename EraseType>
struct EraseAll: public Erase<TypeList, EraseType>
{};

```

But there's more: what if the intention is to erase all elements from the typelist? In that case also apply Erase to the tail when the type to erase matches the typelist's head. E.g., a struct EraseAll can be defined similarly to Erase, except for the case where the typelist's head matches the type to be removed: in that case EraseAll must also be applied to the typelist's tail, as there may be additional types to be removed in the tail as well.

Since EraseAll closely resembles Erase, let's see how we can use

class derivation in combination with specializations to our benefit.

- o First step: note that all alternatives of Erase but one can be used unaltered by EraseAll. So, EraseAll inherits from Erase, using the generic struct's template parameters:

```

template <typename TypeList, typename EraseType>
struct EraseAll: public Erase<TypeList, EraseType>
{};

```

Thus, EraseAll is a mere copy of Erase, and it could be used as a synonym of Erase (of course, erasing only the first element).

- o Second (and last) step: define a specialization of EraseAll for the case where the type to be removed equals the typelist's head:

```

template <typename EraseType, typename Tail>
struct EraseAll<TypeList<EraseType, Tail>, EraseType>
{
    typedef typename EraseAll<Tail, EraseType>::Result Result;
};

```

Note the EraseAll action on the template's tail.

The above two EraseAll definitions are all it takes to create a template that will do the job of erasing all occurrences of a type from a

typelist, borrowing most of its code from the already existing Erase template. The effect of EraseAll vs. Erase can be seen when defining either Erase or EraseAll in the following example:

```
#include <iostream>
#include "erase.h"
#include "listsize.h"

// change Erase to EraseAll to erase all `int' types below
#define ERASE Erase

int main()
{
    std::cout <<
        ListSize<
            ERASE<TYPELIST_3(int, double, int), int>::Result
        >::size << "\n";
}
```

20.6.5.1: Erasing duplicates

So, erasing a type from a typelist can be accomplished. To remove duplicates

all `head' elements must be erased from a typelist's tail. To accomplish this, the following algorithm is used, defining the EraseDuplicates template:

- o First, the general EraseDuplicates struct is declared. It expects as its single template type a TypeList:

```
template <typename TypeList>
struct EraseDuplicates;
```

- o If the typelist is actually a NullType, we're done. The result will remain to be a NullType:

```
template <>
struct EraseDuplicates<NullType>
{
    typedef NullType Result;
};
```

- o Next, an EraseDuplicates specialization is defined for a true Typelist:

```
template <typename Head, typename Tail>
struct EraseDuplicates<TypeList<Head, Tail> >
...

```

This specialization implements the following reasoning:

- o If `EraseDuplicates` erases duplicates, then no duplicates will be encountered in the `typelist`'s tail if `EraseDuplicates` is recursively processes the template's tail. When this recursive call is completed, a `typelist` is obtained (called, e.g., `UniqueTail`) not containing any type duplicates in the `typelist`'s tail:

```
typedef typename EraseDuplicates<Tail>::Result UniqueTail;
```

- o Of course, the above operation only processed the original `typelist`'s tail. It may still contain duplicate's of the original template's head type. Since that latter type is already there (viz. at the original `typelist`'s head) it can simply be removed from `UniqueTail`, producing the `typelist` `NewTail`:

```
typedef typename Erase<UniqueTail, Head>::Result NewTail;
```

- o Finally, the resulting `typelist` is the original `typelist`'s head and `NewTail`:

```
typedef TypeList<Head, NewTail> Result;
```

Here's the full definition of `EraseDuplicates`, not exposing `UniqueTail` and `NewTail` for cosmetic reasons:

```
template <typename TypeList>
struct EraseDuplicates;

template <>
struct EraseDuplicates<NullType>
{
    typedef NullType Result;
};

template <typename Head, typename Tail>
class EraseDuplicates<TypeList<Head, Tail> >
{
    typedef typename EraseDuplicates<Tail>::Result UniqueTail;
    typedef typename Erase<UniqueTail, Head>::Result NewTail;

public:
    typedef TypeList<Head, NewTail> Result;
};
```

20.7: Using a `TypeList`

In the previous sections the definition and some of the features of typelists have been discussed. Most C++ programmers consider typelists both exciting and an intellectual challenge, honing their skills in the area of recursive programming.

Fortunately, there's more to typelist than a mere intellectual challenge. In this section of the chapter on Advanced Template Applications the following topics will be covered:

- o Creating classes from a typelist
- Here the aim is to construct a
- new class consisting of instantiations of an existing basic template for each of the types mentined in a provided typelist;
- o Accessing data members from the thus constructed conglomerate class by index, rather than name;
 - o Tuples, defining structs from typelists, having index-accessible data members for each of the types specified in a typelist.

Again, much (and more) of the materials covered below is found in

Alexandrescu's (2001) book. Hopefully the current section is an tasty appetizer for the main courses offered by Andrei Alexandrescu.

20.7.1: The Wrap and GenScat templates

In this section the template class GenScat will be developed. The purpose of GenScat is to create a new class using on the one hand a basic building block of the class that's finally constructed and on the other hand a series of types that will be fed to the building block.

The building block itself is provided as a

- template template parameter,

and the final class will inherit from all building blocks instantiated for each of the types specified in a provided typelist. However, there is a flaw in this plan.

If the typelist contains two types, say int and double and the building block class is `std::vector`, then the final GenScat class will inherit from `vector<int>` and `vector<double>`. There's nothing wrong with that. But what if the typelist contains two int type specifications? In that case the GenScat class will inherit from two `vector<int>` classes, and, e.g., `vector<int>::size()` will cause an ambiguity which is hard to solve. Alexandrescu (2001)

- in this regard writes (p.67):

There is one major source of annoyance....: you cannot use it when you have duplicate types in your typelist.
.... There is no easy way to

solve the ambiguity, [as the eventually derived class/FBB] ends up inheriting [the same base class/FBB] twice.

It is true that the same base class is inherited multiple times when the typelist contains duplicate types, but there is a way around this problem. If instead of inheriting from the plain base classes these base classes would themselves be wrapped in unique classes, then these unique classes can be used to access the base classes by implication: since they are mere wrappers they inherit the functionality of the 'true' base classes.

Thus, the problem is shifted from type duplication to finding unique

wrappers. Of course, that problem has been solved in principle in section 20.2.1.1, where wrappers around plain int values were introduced. A comparable wrapper can be designed in the context of class derivation. E.g.,

```
template <typename Base, int idx>
struct Wrap: public Base
{
    Wrap(Base const &base)
    :
        Base(base)
    {}
    Wrap()
    {}
};
```

Using Wrap two vector<int> classes can be distinguished easily: Wrap<1, vector<int> > could be used to refer to one of the vectors, Wrap<2, vector<int> > could refer to the other vector. By ensuring that the index values never collide all wrapper types will be unique.

Uniqueness of the Wrap values is realized by the GenScat class: it is itself a wrapper around the class GenScatter, that will do all the work. GenScat merely seeds GenScatter with an initial value:

```
template <typename Type, template <typename> class TemplateClass>
class GenScat: public GenScatter<Type, TemplateClass, 0>
{};
```

20.7.2: The GenScatter template

The interesting part of the exercise is of course the class template GenScatter. In its intended form (which actually turns out to be a specialization) GenScatter takes a typelist, a template class and a index value that is used to ensure uniqueness of the Wrap types.

With these ingredients, GenScatter creates a (fairly complex) class, that itself is normally derived from several GenScatter base classes. Each GenScatter class is eventually derived from a base class which is a Wrap class around the template class instantiated for each of the types in the provided typelist.

This complex arrangement itself causes yet another problem: it would be nice if the top-level derived class could be initialized by base class

initializers. However, with normal class derivation indirect base classes cannot be initialized using base class initializers. This is a severe problem: if indirect base classes cannot be initialized by a GenScat class or by a class derived from GenScat, the types in the provided typelist cannot be

reference types, as references must be initialized at construction time.

However, an indirect base class can be initialized by a top level derived class

if it is a virtual base class (cf. section 14.4.2). The consequence of a virtual base class is that any duplicates of a virtual base class, following different paths in a class hierarchy will be merged into one class.

In the GenScat hierarchy the Wrap template class wrappers are the final (usable) base classes of the hierarchy. By defining these Wrap base classes as virtual base classes they can be initialized by GenScat (or by a class that itself is derived from GenScat), while merging of the Wrap base classes is prevented due to the fact that all Wrap base classes are unique.

It's time for some code. The GenScatter class performs the following tasks:

- o It creates a class hierarchy, having unique Wrap template classes at its final nodes;
- o It creates a typelist containing all Wrap base class types in the order in which they were constructed, to allow clients to obtain the Wrap template class wrapper matching a specific type in the provided typelist.

An illustration showing the layout of the final GenScatter class hierarchy and its subclasses is provided in figure 19.

-----Insert Figure
19(Layout of a GenScatter class hierarchy)about here (file:
templateapp/genscatter)-----

The core definition of GenScatter expects a typelist, a template class and an index:

```

template <
    typename Head, typename Tail,
    template <typename> class TemplateClass, int idx
>
class GenScatter<TypeList<Head, Tail>, TemplateClass, idx>
:
    virtual public Wrap<TemplateClass<Head>, idx>,
    public GenScatter<Tail, TemplateClass, idx + 1>
{
    typedef typename GenScatter<Tail, TemplateClass, idx + 1>::WrapList
        BaseWrapList;
    public:
        typedef TypeList<Wrap<TemplateClass<Head>, idx>, BaseWrapList>
            WrapList;
};

```

o Since the typelist's head is a plain type, it can immediately be used to define a TemplateClass type, which itself is wrapped in a Wrap template class wrapper, using the unique index value that was passed to GenScatter.

o The Wrap template class wrapper is then defined as a virtual base class.

o A second hierarchical line starts at the typelist's tail, at the same time incrementing the index, thus ensuring that the next Wrap class will receive the next index value.

o At the end of the class definition the WrapList type is defined as the typelist consisting of the current Wrap wrapper as its head, and the base GenScatter class's WrapList as its tail.

GenScatter's main template definition expects a simple type as its first template parameter. Since this is a plain type, the class can immediately define a virtual Wrap template class wrapper as its base class, and it can immediately define the WrapList type as a typelist containing the class's base class:

```

template <typename Type, template <typename> class TemplateClass, int idx>
class GenScatter
:
    virtual public Wrap<TemplateClass<Type>, idx>
{
    typedef Wrap<TemplateClass<Type>, idx> Base;

    public:
        typedef TYPELIST_1(Base)    WrapList;
};

```

Finally, a specialization to handle the ending NullType is required:

it merely defines an empty WrapList:

```
template <template <typename> class TemplateClass, int idx>
class GenScatter<NullType, TemplateClass, idx>
{
    public:
        typedef NullType WrapList;
};
```

Both the Wrap template class wrapper and the GenScatter class can normally be defined in the anonymous namespace, as they are only used at file-scope, by themselves, by GenScatter and by the occasional additional support functions and classes.

20.7.3: Support struct and function

Since the GenScatter class returns a typelist containing all Wrap base classes matching the types in the order in which they appeared in GenScat's typelist, it is attractive to be able to obtain these Wrap base class types by their index numbers. Being able to reach these types by their indices allows, e.g., base class initializations as well as quick access to their respective members.

The class template BaseClass was designed with these thoughts in mind. It uses AtIndex (cf. section 20.6.3) to obtain a particular Wrap base class and returns the latter type as its Type type definition:

```
template <int idx, typename Derived>
struct BaseClass
{
    typedef typename TypeAt<idx, typename Derived::WrapList>::Type Type;
};
```

Once a GenScatter object is available, the following function template can be used to obtain a cast-less reference to any of its Wrap policy classes, given its index. Since the index can be matched one-to-one with the GenScatter's typelist, clients should have no problem finding the appropriate index values for a particular problem at hand:

```
template <int idx, typename Derived>
struct BaseClass
{
    typedef typename TypeAt<idx, typename Derived::WrapList>::Type Type;
};
```

20.7.4: Using GenScatter

The class template GenScat can be used by itself, to define a simple struct containing various data members. The foundation of such a conglomerate struct could be the following struct Field:

```
template <typename Type>
struct Field
{
    Type field;

    Field(Type v = Type())
    :
        field(v)
    {}
};
```

Such an instant struct could be useful in various situations; due to the nature of the struct Field, all data types would by default be initialized to their natural defaults. E.g., GenScat can be used directly as follows:

```
GenScat<TYPELIST_2(int, int), Field> gs;

base<1>(gs).d_value = 12;
cout << base<0>(gs).d_value << " " << base<1>(gs).d_value << endl;
```

The above code, when it is run from main() will write the values 0 and 12, showing that default initialization and assignment to the individual fields is simply realized.

Useful as this may be, sometimes more refined initializations may be necessary. E.g, an application needs a struct having two int data fields and a reference to a std::string. Since the struct contains a reference field, an initialization is required at construction time. In this case a struct can be derived from GenScat, while providing a constructor for the derived class performing the necessary initializations. For situations like these, the BaseClass support struct (section 20.7.3) comes in quite handy. Here is the struct MyStruct, derived from the appropriate GenScat template, including its field-initializations:

```
struct MyStruct: public
    GenScat<TYPELIST_3(int, std::string &, int), Field>
{
    MyStruct(int i, std::string &text, int i2)
    :
        BaseClass<0, MyStruct>::Type(i),
        BaseClass<1, MyStruct>::Type(text),
        BaseClass<2, MyStruct>::Type(i2)
    {}
};
```



```
};
```

Note how each of the types in the provided typelist has its order-number mapped to the index used with the `BaseClass` invocations. Also, since `MyStruct` is also an object of its base class (`GenScat`), it can be specified as the `Derived` argument of `BaseClass`. Furthermore, from the types specified in the typelist the types of acceptable arguments of the `Types` to be initialized can be derived. E.g., the string `text` is passed as argument to `Type` when initializing the second field.

The following example shows how `MyStruct` can be used:

```
string text("hello");
MyStruct myStruct(12345, text, 12);

cout << base<0>(myStruct).field << " " <<
      base<1>(myStruct).field << " " <<
      base<2>(myStruct).field << endl;

base<0>(myStruct).field = 123;
base<1>(myStruct).field = "new text";

cout << base<0>(myStruct).field << "\n" <<
      "`text' now contains: " << text << endl;
```

When these lines of code are placed in a `main()` function, and the program is run the following output is produced showing proper initialization, reassignment and reassignment of the referred to string `text` via the appropriate `MyStruct` field:

```
12345 hello 12
123
`text' now contains: new text
```

As a final example consider the struct `Vectors`:

```
struct Vectors: public GenScat<TYPELIST_3(int, std::string, int), std::vector>
{
    Vectors()
    :
        BaseClass<0, Vectors>::Type(std::vector<int>(1)),
        BaseClass<1, Vectors>::Type(std::vector<std::string>(2)),
        BaseClass<2, Vectors>::Type(std::vector<int>(3))
    {}
};
```

The struct `Vectors` uses `std::vector` as its template template parameter, and `Vectors` objects will thus offer three `std::vectors`: the

first containing ints, the second strings, and the third again ints. Due to the nature of the Wrap template class wrapper, the three `std::vector` base classes of `Vectors` must be initialized by `std::vector` objects, and the constructor simply provides three vectors of varying sizes. Alternatively, the constructor could be furnished with three vector references or three `size_t` values to allow a more flexible initialization.

A `Vectors` object could be used as follows, showing that the base support function (cf. section 20.7.3) provides easy access to the vector base class of choice:

```
Vectors vects;

cout << base<0>(vects).size() << " " << base<1>(vects).size() << " " <<
      base<2>(vects).size() << endl;
```

Running this code fragment produces the output ``1 2 3'`, as expected.

Capítulo 21: Exemplos Concretos em C++

Neste capítulo muitos exemplos concretos de programas em linguagem C++, classes e modelos serão apresentados. Tópicos cobertos por este documento tais como funções virtuais, membros estáticos, etc., são ilustrados neste capítulo. Os exemplos seguem estritamente a organização dos capítulos anteriores.

Primeiro, exemplos usando classes 'streams' são apresentados, incluindo alguns exemplos detalhados ilustrando o polimorfismo. Com o advento do estandarte ANSI/ISO, classes que suportam 'streams' baseadas em arquivos descritores já não estão disponíveis, incluindo a extensão 'procbuf' Gnu. Estas classes foram usadas com frequência em programas C++ mais antigos. Esta seção das Anotações C++ desenvolve uma alternativa: Classes que estendem 'streambuf', permitindo o uso de arquivos descritores e classes em volta do sistema de chamada 'fork()'.

Em seguida diversos modelos serão desenvolvidos, ambos, modelos de funções e modelos de classes.

Finalmente tocaremos temas sobre geradores de 'scanners' e 'parser', e mostraremos como essas ferramentas podem ser usadas em programas C++. Estes exemplos finais assumem certa familiaridade com os conceitos subjacentes a estas ferramentas, como gramáticas, árvores de 'parser' e decoração de árvore de 'parser'. Uma vez que um programa exceda certo nível de complexidade, é vantajoso usar geradores de 'scanner' e 'parser', para produzir código que realizam o reconhecimento de entrada. Um dos exemplos no capítulo descreve o uso dessas ferramentas num ambiente C++.

21.1: Usando descritores de arquivos com as classes 'streambuf'

21.1.1: Classes para operações de saída

Existem extensões da norma ANSI/ISO que permitem a leitura e/ou escritura a arquivos descritores. Contudo tais extensões não são estandartes e, por isso, podem variar ou não estarem disponíveis entre os compiladores e/ou versões dos compiladores. Por outro lado um arquivo descritor pode ser considerado um dispositivo. Assim parece natural usar a classe 'streambuf' como ponto de partida para construir classes de interface com os arquivos descritores.

Nesta seção construiremos classes para escrever num dispositivo identificado com um arquivo descritor: Pode ser um arquivo, mas também pode ser um 'pipe' (linha) ou 'socket' (adaptador). A seção 21.1.2 reconsidera o re-direcionamento, discutido antes na seção 5.8.3.

Basicamente, derivar uma classe para operações de saída é simples. A única função membro que precisa ser suprimida é o membro virtual 'int overflow(int c)'. Este membro é responsável por escrever caracteres no dispositivo uma vez que o bufer da classe esteja cheio. Se 'fd' é um descritor de arquivo onde se pode escrever informações e se decidimos re-usar o bufer, então o membro 'overflow()' simplesmente será:

```
class UnbufferedFD: public std::streambuf
{
    public:
        int overflow(int c)
        {
            if (c != EOF)
            {
                if (write(fd, &c, 1) != 1)
                    return EOF;
            }
            return c;
        }
        ...
int UnbufferedFD::overflow(int c)
{
    if (c != EOF)
    {
        if (write(d_fd, &c, 1) != 1)
            return EOF;
    }
    return c;
}
```

O argumento recebido por 'overflow()' ou é escrito como um valor do tipo 'char' num arquivo descritor ou retorna 'EOF'.

Esta função simples não usa um bufer de saída. Como o uso de um bufer de saída é fortemente recomendável (veja também a próxima seção), a construção de uma classe que use um bufer de saída será discutida em seguida com detalhe algo maior.

Quando um bufer de saída é usado, o membro 'overflow()' será um pouco mais complexo, já que então será chamada somente quando o bufer estiver cheio. Uma vez o bufer cheio, temos que primeiro esvaziá-lo, para o que a função (virtual) 'streambuf::sync()' está disponível. Como 'sync()' é uma função virtual, as classes derivadas de 'std::streambuf' podem redefinir 'sync()' para purgar um bufer 'std::streambuf' o qual não conhece.

Suprimir 'sync()' e usá-la em 'overflow()' não é tudo a ser feito: Eventualmente teremos menos informação que a de um bufer completo. Assim, no final da vida de nosso objeto 'streambuf', seu bufer pode estar parcialmente cheio. Por isso devemos nos assegurar que o bufer seja purgado uma vez que nosso objeto saia do escopo. Isto é simples: 'sync()' deve ser chamada pelo destrutor também.

Agora que consideramos as conseqüências de usar um bufer de saída, quase estamos prontos a construir nossa classe derivada. Adicionaremos um par de características também.

- Primeiro, permitiremos ao usuário da classe especificar o tamanho do bufer de saída.
- Segundo, pode ser possível construir um objeto de nossa classe antes do arquivo descritor ser conhecido. Mais adiante, na seção 20.3, encontraremos uma situação onde esta característica será usada.

Para salvar algum espaço, o bom sucesso das várias funções não foram examinados. Na `vida real` a implantação destes exames não deve ser omitida. Nossa classe 'ofdnstreambuf' tem as seguintes características:

- A classe é derivada de 'std::streambuf'. Usa três membros de dados, cuidando do tamanho do bufer, do descritor e do próprio bufer. Seu construtor só inicia o bufer em 0. Eis a interface da classe completa:

```
class ofdnstreambuf: public std::streambuf
{
    size_t d_bufsize;
    int     d_fd;
    char    *d_buffer;

public:
    ofdnstreambuf();
    ofdnstreambuf(int fd, size_t bufsize = 1);
    ~ofdnstreambuf();
    void open(int fd, size_t bufsize = 1);
    int sync();
    int overflow(int c);
};
```

- Seu construtor padrão simplesmente inicia o bufer em 0. Mais interessante é seu construtor que espera um descritor de arquivo e o tamanho do bufer: simplesmente passa seus argumentos para o membro 'open()' (veja abaixo). Eis os construtores:

```
inline ofdnstreambuf::ofdnstreambuf()
```

```

:
    d_bufsize(0),
    d_buffer(0)
{}

inline ofdnstreambuf::ofdnstreambuf(int fd, size_t bufsize)
{
    open(fd, bufsize);
}

```

- Seu destrutor chama a função suprimida 'sync()', escrevendo todos os caracteres do bufer de saída no dispositivo. Se não há bufer, o destrutor não realiza qualquer ação:

```

inline ofdnstreambuf::~~ofdnstreambuf()
{
    if (d_buffer)
    {
        sync();
        delete[] d_buffer;
    }
}

```

Mas o dispositivo não está fechado na implementação acima isto pode não ser o que queremos. Deixamos como exercício ao leitor mudar esta classe de modos que o dispositivo possa ser deixado aberto opcionalmente. Esta solução foi seguida, p.ex., na biblioteca Bobcat. Veja também a seção 21.1.2.2.

- O membro 'open()' inicia o bufer. Usando 'setp()' os pontos de começo e fim do bufer são adjudicados. Isto é usado pela classe de base 'streambuf' para iniciar 'pbase()', 'pptr()' e 'epptr()':

```

inline void ofdnstreambuf::open(int fd, size_t bufsize)
{
    d_fd = fd;
    d_bufsize = bufsize == 0 ? 1 : bufsize;

    d_buffer = new char[d_bufsize];
    setp(d_buffer, d_buffer + d_bufsize);
}

```

- O membro 'sync()' escreverá todo caracter ainda presente no bufer no dispositivo. Em seguida o bufer é reiniciado usando 'setp()'. Note que 'sync()' retorna 0 depois de uma operação de purga bem sucedida:

```

inline int ofdnstreambuf::sync()
{
    if (pptr() > pbase())
    {
        write(d_fd, d_buffer, pptr() - pbase());
        setp(d_buffer, d_buffer + d_bufsize);
    }
}

```

```

    }
    return 0;
}

```

- Finalmente o membro 'overflow()' é suprimido. Como este membro é chamado pela classe de base 'streambuf' quando o bufer está cheio, 'sync()' é chamada antes para purgar o bufer cheio para o dispositivo. Como isto recria um bufer vazio, o caracter 'c' que pode não ter sido escrito no bufer pela classe de base 'streambuf' é agora entrado no bufer usando as funções membro 'pptr()' e 'pbump()'. Atenção que a entrada de um caracter no bufer é feita usando-se as funções disponíveis de 'streambuf', antes que `a mão`, que invalidaria o controle interno de 'streambuf':

```

inline int ofdnstreambuf::overflow(int c)
{
    sync();
    if (c != EOF)
    {
        *pptr() = c;
        pbump(1);
    }
    return c;
}

```

- As implantações das funções membro usa funções de baixo nível para operar sobre o arquivo descritor, assim, além do arquivo cabeçalho de 'streambuf' é preciso que 'unistd.h' seja lido pelo compilador antes que a implantação das funções possa ser compilada.

Dependendo do número de argumentos, o seguinte programa usa a classe 'ofdnstreambuf' para copiar sua entrada padrão para um arquivo descritor 'STDOUT_FILENO', que é o nome simbólico do arquivo descritor usado pela saída padrão. Aqui está o programa:

```

#include <string>
#include <iostream>
#include <istream>
#include "fdout.h"
using namespace std;

int main(int argc)
{
    ofdnstreambuf    fds(STDOUT_FILENO, 500);
    ostream          os(&fds);

    switch (argc)
    {
        case 1:
            os << "COPYING cin LINE BY LINE\n";
            for (string s; getline(cin, s); )
                os << s << endl;
            break;
    }
}

```

```

        case 2:
            os << "COPYING cin BY EXTRACTING TO os.rdbuf()\n";

            cin >> os.rdbuf();          // Alternatively, use: cin >> &fds;
            break;

        case 3:
            os << "COPYING cin BY INSERTING cin.rdbuf() into os\n";
            os << cin.rdbuf();
            break;
    }
}

```

21.1.2: Classes para operações de entrada

Para derivar uma classe que faça operações de entrada de uma 'std::streambuf' a classe deve usar um bufer de entrada de pelo menos um caracter, para permitir o uso da funções membro 'istream::putback()' ou 'istream::ungetc()'. As classes 'stream' (como 'istream') normalmente permitem voltar pelo menos um caracter usando suas funções 'putback()' ou 'ungetc()'. Isto é importante, já que estas classes 'stream' usualmente servem de interface a objetos 'streambuf'. Apesar de que falando estritamente não é necessário implantar um bufer na classe derivada de 'streambuf', usar buffers nestas classes é fortemente aconselhável: A implantação é muito simples e segura e a aplicabilidade dessas classes será bem melhor. Por isso em todas nossas classes derivadas da classe 'streambuf' pelo menos um bufer de um caracter será definido.

21.1.2.1: Usando um bufer de um caracter

Quando derivamos uma classe (p.ex., ifdstreambuf) de 'streambuf' usando um bufer de um caracter, pelo menos seu membro 'streambuf::underflow()' pode ser mantida, já que todas as requisições de entrada são a ela dirigidas. Como é necessário um bufer, o membro 'streambuf::setg()' é usado para informar à classe de base 'streambuf' do tamanho do bufer de entrada, assim estará habilitada a iniciar corretamente o ponteiro do bufer. Isto assegurará que: eback(), gptr(), e egptr() retornem valores corretos.

A classe possui as seguintes características:

- Como a classe projetada para operações de saída, esta classe é derivada de 'std::streambuf' também:

```
class ifdstreambuf: public std::streambuf
```


- A classe recebe dois membros de dados, um deles de tamanho fixo do bufer de um caracter. Os membros de dados são definidos como protegidos, assim as classes derivadas (p.ex., veja seção 21.1.2.3) podem acessá-los:

```
protected:
    int      d_fd;
    char     d_buffer[1];
```

- O construtor inicia o bufer. Contudo, este início é feito tal que 'gptr()' será igual a 'egptr()'. Já que isto implica que o bufer está vazio, 'underflow()' será imediatamente chamada para preencher o bufer:

```
ifdstreambuf(int fd)
:
    d_fd(fd)
{
    setg(d_buffer, d_buffer + 1, d_buffer + 1);
}
```

- Finalmente 'underflow()' é mantida. Primeiro assegurará que o bufer esteja realmente vazio. Se não, o próximo caracter no bufer é retornado. Se o bufer está realmente vazio, é preenchido lendo do arquivo descritor. Se isto falha (por qualquer razão), 'EOF' é retornado. Uma implantação mais sofisticada reagiria mais inteligentemente aqui, claro. Se o bufer pode ser preenchido, 'setg()' é chamada para por os apontadores de 'streambuf' em valores corretos;
- A implantação das funções membro usa funções de baixo nível para operar sobre os arquivos descritores, assim que além do arquivo cabeçalho de 'streambuf', 'unistd.h' deve ser lido pelo compilador antes de poder compilar a implantação das funções membro.

Isto completa a construção da classe 'ifdstreambuf'. É usada no seguinte programa:

```
#include <iostream>
#include <istream>
#include "ifdbuf.h"
using namespace std;

int main(int argc)
{
    ifdstreambuf fds(0);
    istream      is(&fds);

    cout << is.rdbuf();
}
```

21.1.2.2: Usando um bufer de n caracteres

Quão complexas se poriam as coisas se decidíssemos usar um bufer de tamanho substancial? Não tão complexas. A seguinte classe nos permite especificar o tamanho de um bufer, mas além disso é basicamente a mesma classe que a desenvolvida na seção 'ifdstreambuf' anterior. Para fazer as coisas um pouco mais interessantes, na classe 'ifdnstreambuf' desenvolvida aqui o membro 'streambuf::xsgetn()' é também incluído, para leitura de uma série de caracteres. Ainda mais, um construtor padrão é fornecida que pode ser usada em combinação com o membro 'open()' para construir um objeto 'istream' antes que o arquivo descritor se torne disponível. Então uma vez que o descritor fique disponível o membro 'open()' pode ser usado para iniciar o bufer do objeto. Depois, na seção 20.3, encontraremos tal situação.

Para economizar algum espaço, o sucesso de várias chamadas não são examinados. Na 'vida real' não se deve omitir tais exames. A classe 'ifdnstreambuf' possui as seguintes características:

- Mais uma vez é derivada de 'std::streambuf':

Como a classe 'ifdstreambuf' da seção 20.1.2.1, seus membros de dados são protegidos. Como o tamanho do bufer é configurável, esse tamanho é guardado no membro de dados dedicado 'd_bufsize':

```
class ifdnstreambuf: public std::streambuf
{
    protected:
        int          d_fd;
        size_t       d_bufsize;
        char*        d_buffer;
    public:
        ifdnstreambuf();
        ifdnstreambuf(int fd, size_t bufsize = 1);
        ~ifdnstreambuf();
        void open(int fd, size_t bufsize = 1);
        int underflow();
        std::streamsize xsgetn(char *dest, std::streamsize n);
};
```

- O construtor padrão não aloca o bufer e pode ser usado antes do arquivo descritor seja conhecido, para ser construído um objeto.
- Um segundo construtor simplesmente passa seus argumentos a 'open()' que então inicia o objeto, tornando-o disponível ao uso:

```
inline ifdnstreambuf::ifdnstreambuf()
:
    d_bufsize(0),
    d_buffer(0)
```

```

{}
inline ifdnstreambuf::ifdnstreambuf(int fd, size_t bufsize)
{
    open(fd, bufsize);
}

```

- Se o objeto foi iniciado por 'open()', seu destrutor eliminará o bufer do objeto e usa o arquivo descritor para fechar o dispositivo:

```

ifdnstreambuf::~ifdnstreambuf()
{
    if (d_bufsize)
    {
        close(d_fd);
        delete[] d_buffer;
    }
}

```

Verdadeiramente o dispositivo foi fechado na implementação acima, isto nem sempre pode coincidir com o que queremos. Nos casos em que o arquivo descritor já está disponível a intenção pode ser usar esse descritor repetidamente, cada vez usando um objeto `ifdnstreambuf` recém construído. Deixamos como exercício ao leitor alterar esta classe de tal forma que o dispositivo possa ser fechado opcionalmente. Este modelo foi seguido em, p.ex., na biblioteca `Bobcat`.

- O membro 'open()' simplesmente aloca o bufer do objeto. É assumido que o programa que a chama já abriu o dispositivo. Uma vez que o bufer foi alocado, o membro da classe de base 'setg()' é usado para assegurar que 'eback()', 'gptr()' e 'egptr()' retornem valores corretos:

```

void open(int fd, size_t bufsize = 1)
{
    d_fd = fd;
    d_bufsize = bufsize;
    d_buffer = new char[d_bufsize];
    setg(d_buffer, d_buffer + d_bufsize, d_buffer + d_bufsize);
}

```

- O membro ignorado 'underflow()' é implantado quase identicamente que em 'ifdstreambuf' (seção 20.1.2.1). A única diferença é que a classe atual suporta um bufer de maior tamanho. Por isso mais caracteres (até 'd_bufsize') pode ser lido do dispositivo de uma vez:

```

int ifdnstreambuf::underflow()
{
    if (gptr() < egptr())
        return *gptr();

    int nread = read(d_fd, d_buffer, d_bufsize);

    if (nread <= 0)

```

```

        return EOF;

        setg(d_buffer, d_buffer, d_buffer + nread);
        return *gptr();
    }

```

- Finalmente 'xsgetn()' é ignorada. Num laço 'n' é reduzido até 0, onde a função termina. Alternativamente, o membro retorna , se 'underflow()' falha em obter mais caracteres. Este membro otimiza a leitura de séries de caracteres: Em lugar de chamar 'streambuf::sbumpc()' 'n' vezes, um bloco de caracteres de ajuda é copiado ao destino, usando 'streambuf::gpumb()' para consumir os caracteres de ajuda do bufer usando uma chamada à função:

```

std::streamsize xsgetn(char *dest, std::streamsize n)
{
    int nread = 0;

    while (n)
    {
        if (!in_avail())
        {
            if (underflow() == EOF)
                break;
        }

        int avail = in_avail();

        if (avail > n)
            avail = n;

        memcpy(dest + nread, gptr(), avail);
        gbump(avail);

        nread += avail;
        n -= avail;
    }

    return nread;
}

```

- A implantação das funções membro usa funções de baixo nível para operar sobre os arquivos descritores, assim, além do arquivo cabeçalho de 'streambuf' o arquivo cabeçalho 'unistd.h' deve ser lido pelo compilador antes que as funções membro possam ser compiladas.
- A função membro 'xsgetn()' é chamada por 'streambuf::sgetn()' que é um membro de 'streambuf'. O seguinte exemplo ilustra o uso desta função membro com um objeto 'ifdnstreambuf':

```

#include "ifdnbuf.h"
#include <iostream>

```

```

#include <istream>
using namespace std;

int main(int argc)
{
    ifdnstreambuf fds(0, 30);    // internamente: bufer de 30 char
    char buf[80];               // main() lê blocos de 80
                                // caracteres

    while (true)
    {
        size_t n = fds.sgetn(buf, 80);
        if (n == 0)
            break;
        cout.write(buf, n);
    }
}

```

21.1.2.3: Deslocando posições em objetos 'streambuf'

Quando dispositivos que suportam operações de deslocamento, classes derivadas de 'streambuf' devem ignorar 'streambuf::seekoff()' e 'streambuf::seekpos()'. A classe 'ifdseek', desenvolvida nesta seção pode ser usada para ler informações de dispositivos que tais suportem operações de deslocamento. A classe 'ifdseek' foi derivada de 'ifdstreambuf', portanto usa um bufer de caracteres de um só caracter. As facilidades para realizar operações de deslocamento, que foram agregadas à nossa nova classe 'ifdseek' se assegurarão de que o bufer de entrada esteja zerado quando uma operação de deslocamento seja requerida. A classe também pode ser derivada da classe 'ifdnstreambuf', em tal caso o argumento para zerar o bufer de entrada deve ser adaptado de tal forma que seus segundo e terceiro parâmetros aponte além do bufer de entrada disponível. Demos uma olhada nas características de 'ifdseek':

- Como já mencionado, 'ifdseek' é derivada de 'ifdstreambuf'. Como a classe anterior, as funções membro de 'ifdseek' usam facilidades declaradas em 'unistd.h'. Portanto o compilador deve ler 'unistd.h' antes de poder compilar as funções membro da classe. A interface de classe começa com:

```
class ifdseek: public ifdstreambuf
```

- Para reduzir a quantidade de digitação ao especificar tipos e constantes de 'std::streambuf' e 'std::ios' diversos 'typedefs' são definidos no topo da interface de classe.

```

typedef std::streambuf::pos_type      pos_type;
typedef std::streambuf::off_type     off_type;
typedef std::ios::seekdir            seekdir;
typedef std::ios::openmode           openmode;

```

Estes 'typedefs' se referem a tipos definidos no arquivo cabeçalho de 'ios', que portanto precisa ser incluído também antes de que o compilador leia a definição da classe 'ifdseek'.

- A classe é implantada de forma básica. O único construtor requerido espera o dispositivo do arquivo descritor. Não tem tarefas especiais a realizar e só necessita chamar o construtor de sua classe de base:

```
ifdseek(int fd)
:
    ifdstreambuf(fd)
{ }
```

- O membro 'seek_off()' é responsável de realizar as operações de deslocamento. Chama 'lseek()' para deslocar em outra posição no dispositivo do qual um arquivo descritor é conhecido. Se o deslocamento é bem sucedido, 'setg()' é chamada para definir um bufer já vazio, assim, o membro da classe de base 'underflow()' encherá o bufer na próxima requisição de leitura:

```
pos_type seekoff(off_type offset, seekdir dir, openmode)
{
    pos_type pos =
        lseek
        (
            d_fd, offset,
            (dir == std::ios::beg) ? SEEK_SET :
            (dir == std::ios::cur) ? SEEK_CUR :
                                     SEEK_END
        );

    if (pos < 0)
        return -1;

    setg(d_buffer, d_buffer + 1, d_buffer + 1);
    return pos;
}
```

- Finalmente, a função companheira 'seekpos()' é ignorada também: É definida como uma chamada a 'seekoff()':

```
pos_type seekpos(pos_type offset, openmode mode)
{
    return seekoff(offset, std::ios::beg, mode);
}
```

Um exemplo de um programa que use 'ifdseek' é o seguinte. Se a este programa é dado seu próprio arquivo fonte usando re-direcionamento de entrada, então o deslocamento é suportado e com exceção da primeira linha, todas as outras linhas são mostradas duas vezes:

```
#include "fdinseek.h"
```

```

#include <string>
#include <iostream>
#include <istream>
#include <iomanip>
using namespace std;

int main(int argc)
{
    ifdseek fds(0);
    istream is(&fds);
    string s;

    while (true)
    {
        if (!getline(is, s))
            break;

        streampos pos = is.tellg();

        cout << setw(5) << pos << ": " << s << "'\n";

        if (!getline(is, s))
            break;

        streampos pos2 = is.tellg();

        cout << setw(5) << pos2 << ": " << s << "'\n";

        if (!is.seekg(pos))
        {
            cout << "Seek failed\n";
            break;
        }
    }
}

```

21.1.2.4: Chamadas múltiplas a 'unget()' em objetos 'streambuf'

Como dito antes, as classes 'streambuf' e suas classes derivadas podem suportar pelo menos a devolução do último caracter lido. Especial cuidado deve ser tomado quando uma série de chamadas a 'unget()' deve ser suportado. Nesta seção fazemos a construção de uma classe que suporta um número configurável de chamadas a 'istream::unget()' ou 'istream::putback()'

O suporte de chamadas múltiplas (digamos 'n') a 'unget()' é realizado reservando-se uma seção inicial no bufer de entrada, que é preenchido gradualmente para conter os últimos 'n' caracteres lidos. A classe foi implantada como segue:

- Novamente a classe é derivada de 'std::streambuf'. Define diversos membros de dados, permitindo à classe realizar o controle requerido para manter um bufer de devolução de tamanho configurável:

```
class fdunget: public std::streambuf
{
    int          d_fd;
    size_t       d_bufsize;
    size_t       d_reserved;
    char*        d_buffer;
    char*        d_base;

    public:
        fdunget(int fd, size_t bufsz, size_t unget);
        ~fdunget();
        int underflow();
};
```

- O construtor da classe espera um arquivo descritor, um tamanho de bufer e o número de caracteres que podem ser devolvidos como argumentos. Este membro determina o tamanho da área reservada, definido como os 'd_reserved' primeiros bytes do bufer de entrada da classe;
- O bufer de entrada sempre será pelo menos um byte maior que 'd_reserved'. Assim, um certo número de bytes pode ser lido. Então, uma vez que os bytes reservados tenham sido lidos pelo menos os bytes reservados podem ser devolvidos;
- Em seguida o ponto de leitura é configurado: É chamado 'd_base', localizados no 'd_buffer' e reservados. Este é o ponto onde o preenchimento do bufer sempre começa.
- Agora que o bufer foi construído, estamos prontos a definir os apontadores ao bufer usando 'setg()'. Como ainda não foi lido nenhum caracter, todos os ponteiros apontam para 'd_base'. Se neste ponto 'unget()' for chamada, sem caracteres disponíveis, corretamente falhará.
- Eventualmente o tamanho de preenchimento do bufer é determinado pelo número de bytes alocados menos o tamanho da área reservada.

Eis o construtor da classe:

```
fdunget (int fd, size_t bufsz, size_t unget)
:
    d_fd(fd),
    d_reserved(unget)
{
    size_t allocate =
        bufsz > d_reserved ?
        bufsz
```



```

        :
        d_reserved + 1;

    d_buffer = new char [allocate];

    d_base = d_buffer + d_reserved;
    setg(d_base, d_base, d_base);

    d_bufsize = allocate - d_reserved;
}

```

- O destrutor da classe simplesmente desaloja a memória do bufer:

```

~fdunget ()
{
    delete [] d_buffer;
}

```

- Finalmente, 'underflow()' é ignorada;
- Primeiramente um exame estandarte para determinar se o bufer está realmente vazio é aplicado;
- Se está vazio, determina o número de caracteres que potencialmente podem ser devolvidos. Neste ponto o bufer de entrada é esvaziado, este valor pode ser entre 0 (estado inicial) e o tamanho do bufer (quando a área reservada foi preenchida completamente e todos os caracteres na seção seguinte do bufer também foram lidos);
- Em seguida o número de bytes a serem movidos para a área reservada é calculado. Este número é no máximo 'd_reserved', mas é igual ao número atual de caracteres que podem ser devolvidos se este valor for menor;
- Agora o número de caracteres a ser movido para a área reservada é conhecido, este número de caracteres é movido do fim do bufer de entrada para a área imediatamente anterior a 'd_base';
- Então o bufer é preenchido. Tudo isto é padrão, mas atenção, a leitura começa de 'd_base' e não de 'd_buf';
- Finalmente, os apontadores de leitura de 'streambuf' são atualizados:

'Eback()' é posto a mover locais antes de 'd_base', garantindo assim a área de devolução,
 'gptr()' é posto em 'd_base', já que esta é a localização do primeiro caracter lido depois do preenchimento, e
 'egptr()' é posto justo depois do local do último caracter lido no bufer.

Eis a implantação de 'underflow()':

```
int underflow()
{
    if (gptr() < egptr())
        return *gptr();

    size_t ungetsize = gptr() - eback();
    size_t move = std::min(ungetsize, d_reserved);

    memcpy(d_base - move, egptr() - move, move);

    int nread = read(d_fd, d_base, d_bufsize);
    if (nread <= 0)          // none read -> return EOF
        return EOF;

    setg(d_base - move, d_base, d_base + nread);

    return *gptr();
}
};
```

O seguinte programa ilustra a classe 'fdunget'. Lê no máximo 10 caracteres da entrada padrão, parando em 'EOF'. Um bufer para devolução garantido de 2 caracteres é definido, dentro de um bufer de 3 caracteres. Justo antes de ler um caracter intenta um 'unget()' de até 6 caracteres. Isto é, claro está, impossível, mas o programa devolverá corretamente tantos caracteres quantos possível, considerando o número atual de caracteres lidos:

```
#include "fdunget.h"
#include <string>
#include <iostream>
#include <istream>
using namespace std;

int main(int argc)
{
    fdunget fds(0, 3, 2);
    istream is(&fds);
    char    c;

    for (int idx = 0; idx < 10; ++idx)
    {
        cout << "after reading " << idx << " characters:\n";
        for (int ug = 0; ug <= 6; ++ug)
        {
            if (!is.unget())
            {
                cout
                << "\tunget failed at attempt " << (ug + 1) << "\n"
                << "\trereading: ";
            }
        }
    }
}
```

```

        is.clear();
        while (ug--)
        {
            is.get(c);
            cout << c;
        }
        cout << "'\n";
        break;
    }
}

if (!is.get(c))
{
    cout << " reached\n";
    break;
}
cout << "Next character: " << c << endl;
}
}
/*
Saída Gerada depois de 'echo abcde | programa':

after reading 0 characters:
    unget failed at attempt 1
    rereading: ''
Next character: a
after reading 1 characters:
    unget failed at attempt 2
    rereading: 'a'
Next character: b
after reading 2 characters:
    unget failed at attempt 3
    rereading: 'ab'
Next character: c
after reading 3 characters:
    unget failed at attempt 4
    rereading: 'abc'
Next character: d
after reading 4 characters:
    unget failed at attempt 4
    rereading: 'bcd'
Next character: e
after reading 5 characters:
    unget failed at attempt 4
    rereading: 'cde'
Next character:

after reading 6 characters:
    unget failed at attempt 4
    rereading: 'de
,

```

reached
*/

21.2: Extração de campos de tamanho fixo de objetos 'istream'

Usualmente ao extrair informações de objetos 'istream' usamos o operador 'operator>>()' que é perfeitamente desejável para a tarefa que na maioria dos casos os campos extraídos são de espaços ou claramente separados uns dos outros. Mas isto não é assim em todas as situações. Por exemplo, quando um formulário é postado por internet para algum processamento, o programa receptor pode receber os caracteres codificados 'url' : letras e espaços são enviados inalterados, os espaços são enviados como o caracter '+' e todos os outros caracteres começam por '%' seguido pelo valor ASCII representado pelos seus dois dígitos hexadecimais.

Ao decodificar informação codificada 'url', uma simples extração hexadecimal não funciona, já que extrairá tantos caracteres hexadecimais quantos disponíveis, no lugar de somente dois. Como as letras a-f e 0-9 são hexadecimais legais, um texto como 'My name is Ed', será codificado 'url' como:

My+name+is+%60Ed%27

O que resultará na extração de hexadecimais 60ed e 27, no lugar de 60 e 27. O nome 'Ed' desaparecerá de vista, o que claramente não é o que queremos.

Neste caso ao encontrarmos '%' deveríamos extrair 2 caracteres, pô-los num objeto 'istringstream'. Algo estranho, mas realizável. Outras soluções também são possíveis.

A seguinte classe 'fistream' para campos 'istream' de tamanho fixo define uma classe 'istream' que suporta extrações de campos de tamanho fixo e extrações delimitadas por espaços (como também chamadas a 'read()' não formatado). A classe pode ser iniciada como envolvente a uma 'istream' existente ou pode ser iniciada usando de um arquivo existente. A classe é derivada de 'istream' , permitindo qualquer extração e operações, em geral, suportadas por 'istream'. A classe necessitará os seguintes membros de dados:

- - 'd_filebuf': Um bufer para arquivo usado quando 'fistream' lê informações de um arquivo (existente). Como o bufer para arquivo só é necessário nesses casos e precisa estar alocado dinamicamente, é definido como objeto 'auto_ptr<filebuf>;
- - 'd_streambuf': Um apontador ao 'streambuf' de 'fistream'. Apontará a 'filebuf' quando 'fistream' abrir um arquivo pelo nome. Quando é usada uma 'istream' existente para construir uma 'fistream' apontará ao 'streambuf' da 'istream' existente.

- - 'd_iss': É um objeto 'istream' usado para extrações de campos fixos.
- - 'd_with': um 'unsigned' que indica a extensão do campo a extrair. Se for 0 não será usada extração de campo fixo, mas a informação será extraída do objeto 'istream' da classe de base usando extração padrão.

Eis aqui a seção inicial da interface de classe de 'fistream':

```
class fistream: public std::istream
{
    std::auto_ptr<std::filebuf> d_filebuf;
    std::streambuf *d_streambuf;
    std::istreamstream d_iss;
    unsigned d_width;
```

Como já mencionado os objetos podem ser construídos a partir de um nome de arquivo ou de um objeto 'istream' existente. Portanto a interface de classe tem dois construtores:

```
fistream(std::istream &stream);
fistream(char const *name,
        std::ios::openmode mode = std::ios::in);
```

Quando um objeto 'fistream' é construído usando um objeto 'istream' existente, a 'istream' da 'fistream' usa simplesmente o 'streambuf' do objeto 'stream':

```
fistream::fistream(istream &stream)
:
    istream(stream.rdbuf()),
    d_streambuf(rdbuf()),
    d_width(0)
{ }
```

Quando uma 'fistream' é construída usando o nome de um arquivo, o iniciador da 'istream' de base lhe dá um novo objeto 'filebuf' para ser usado como seu 'streambuf'. Como os membros de dados da classe não foram iniciados antes da classe de base ter sido construída, 'd_filebuf' só pode ser iniciado depois disso. Nesse momento o 'filebuf' está disponível só como 'rdbuf()', que retorna um 'streambuf'. Contudo, como é um 'filebuf' agora, um 'reinterpret_cast' é usado para fazê-lo de um ponteiro retornado por 'rdbuf()' um 'filebuf*', Assim, 'd_filebuf' pode ser iniciado:

```
fistream::fistream(char const *name, ios::openmode mode)
:
    istream(new filebuf()),
    d_filebuf(reinterpret_cast<filebuf *>(rdbuf())),
    d_streambuf(d_filebuf.get()),
    d_width(0)
{
    d_filebuf->open(name, mode);
}
```

Há um só membro público adicional: 'setField(field const &)'. Este membro é usado para definir o tamanho do próximo campo a ser extraído. Seu parâmetro é uma referência à classe 'field', uma classe de manipulação que define o tamanho do próximo campo.

Como um 'field &' é mencionado na interface de 'fistream', 'field' precisa ser declarado antes da interface de 'fistream' começar. A classe 'field' é simples: Declara 'fistream' como amiga e tem dois membros de dados: 'd_width', que especifica o tamanho do próximo campo e 'd_newWidth' que é posta em verdadeiro se o valor de 'd_width' deve ser usado. Se 'd_newWidth' é falso, 'fistream' retorna ao seu modo padrão de extração. A classe 'field' tem dois construtores: Um padrão que põe 'd_newWidth' em falso e um segundo que espera o tamanho do próximo campo a extrair como seu valor. Eis a classe 'field':

```
class field
{
    friend class fistream;
    size_t d_width;
    bool    d_newWidth;

    public:
        field(size_t width)
        field();
};

inline field::field(size_t width)
:
    d_width(width),
    d_newWidth(true)
{}

inline field::field()
:
    d_newWidth(false)
{}

```

Como 'field' declara 'fistream' como amiga, 'setField' pode inspecionar os membros de 'field' diretamente.

Voltando a 'setField()'. Esta função espera uma referência a um objeto 'field', iniciado num dos três modos diferentes:

- - 'field()': Quando o argumento de 'setField()' for um objeto de 'field' construído pelo construtor padrão a próxima extração usará o mesmo 'fieldwidth' que a extração anterior;
- - 'field(0)': Quando este objeto 'field' for usado como argumento de 'setField()', para a extração de campos com tamanho fixo e 'fistream' se comportará como um objeto 'istream' estandarte.
- - 'field(x)': Quando o objeto 'field' é iniciado com um valor size_t diferente de zero, então o próximo campo terá o tamanho de x caracteres. A preparação de tal campo é deixada a 'setBuffer()', membro privado de

'fistream' só.

Aqui está a implantação de 'setField()':

```
std::istream &fstream::setField(field const &params)
{
    if (params.d_newWidth)                // requisitado novo tamanho de campo
        d_width = params.d_width;        // põe novo tamanho

    if (!d_width)                         // sem tamanho?
        rdbuf(d_streambuf);              // retorna ao bufer antigo
    else
        setBuffer();                     // define o bufer de extração

    return *this;
}
```

O membro privado 'setBuffer()' define um bufer de tamanho 'd_width + 1' caracteres e usa 'read()' para preencher o bufer com 'd_width' caracteres. O bufer é terminado com o caracter ASCII-Z. Este bufer é então usado para iniciar o membro 'd_str'. Finalmente a 'rdbuf()', membro de 'fstream', é usada para extrair os dados 'd_str' através do objeto de 'fstream':

```
void fstream::setBuffer()
{
    char *buffer = new char[d_width + 1];

    rdbuf(d_streambuf);                  // usa o bufer de istream para
    buffer[read(buffer, d_width).gcount()] = 0; // ler d_width caracteres,
                                              // terminados por ascii-Z

    d_iss.str(buffer);
    delete buffer;

    rdbuf(d_iss.rdbuf());                // chaveia os buffers
}
```

Apesar de que 'setField()' poderia ser usada para configurar 'fstream' para usar ou não extração com tamanho fixo, usar manipuladores, provavelmente é preferível. Para permitir que objetos 'field' sejam usados como manipuladores um operador sobrecarregado de extração foi definido, que aceita uma 'istream &' e um objeto 'field const &'. Usando este operador de extração, os comandos como:

```
fis >> field(2) >> x >> field(0);
```

São possíveis (assumindo que 'fis' é um objeto 'fstream'). Aqui está o 'operator>>()' sobrecarregado, bem como sua declaração:

```
istream &std::operator>>(istream &str, field const &params)
{
    return reinterpret_cast<fstream *>(&str)->setField(params);
}
```

Declaration:

```
namespace std
{
    istream &operator>>(istream &str, FBB::field const &params);
}
```

Finalmente um exemplo. O seguinte programa usa um objeto 'fistream' para decodificar informação codificada 'url' que aparece em sua entrada padrão:

```
int main()
{
    ifstream fis(cin);

    fis >> hex;
    while (true)
    {
        size_t x;
        switch (x = fis.get())
        {
            case '\n':
                cout << endl;
                break;
            case '+':
                cout << ' ';
                break;
            case '%':
                fis >> field(2) >> x >> field(0);
                // FALLING THROUGH
            default:
                cout << static_cast<char>(x);
                break;
            case EOF:
                return 0;
        }
    }
}
/*
Entrada:

echo My+name+is+%60Ed%27 | a.out

Saída Gerada:

My name is `Ed`
*/
```


21.3: O sistema de chamadas 'fork()'

O sistema de chamadas 'fork()' é bem conhecido da linguagem de programação C. Quando um programa necessita partir um novo processo pode-se usar 'system()', mas esta requer esperar que um processo filho termine. O modo mais geral de criar sub-processos é chamar 'fork()'.

Nesta seção veremos como a linguagem C++ pode ser usada para envolver classes num sistema complexo como 'fork()'. Muito do que segue nesta seção se aplica ao sistema operacional Unix e a discussão será focalizada, portanto, naquele sistema. Contudo, Outros sistemas fornecem, usualmente, facilidades comparáveis. A discussão a seguir está fortemente baseada na noção de padrões de projeto, conforme publicado em Gamma et al. (1995).

Quando 'fork()' é chamada, o programa atual é duplicado em memória, isto cria um novo processo e ambos processos continuam sua execução justo abaixo da chamada a 'fork()': O valor de retorno do processo original (chamado processo pai) difere do valor de retorno do recém criado processo (chamado processo filho):

- No processo pai, 'fork()' retorna o ID do processo filho criado pela chamada ao sistema 'fork()'. Este é um valor inteiro positivo.

Uma classe básica 'Fork' deve esconder todos detalhes de controle de uma chamada ao sistema 'fork()' dos usuários. A classe 'Fork' desenvolvida aqui o fará assim. A própria classe necessita só cuidar da execução apropriada da chamada ao sistema 'fork()'. Normalmente 'fork()' é chamada para iniciar um novo processo, usualmente retardando a execução de um processo separado. Este processo filho pode esperar entradas em sua 'stream' de entrada padrão e/ou pode gerar saídas em sua saída padrão e/ou 'streams' padrão de erro. A classe 'Fork' não conhece nada disto e não tem que saber o que o processo filho fará. Contudo, os objetos 'Fork' devem estar habilitados a ativar seus processos filhos.

Desafortunadamente o construtor de 'Fork' não pode saber que ações seu processo filho realizará. Igualmente não sabe que ações o processo pai pode realizar. Devido a esta situação particular, o modelo do método de padrão de projeto foi desenvolvido. De acordo com Gamma c.s., o modelo do método de padrão de projeto é:

*“Define o esqueleto de um algoritmo numa operação, deixando alguns passos a sub-classes.
(O) Método de Modelagem (de Padrão de Projeto) permite a sub-classes redefinir certos passos de um algoritmo, sem mudar a estrutura do algoritmo”*

Este padrão de projeto nos permite definir uma classe de base abstrata implantando já os passos essenciais relativos à chamada ao sistema 'fork()' e deixa a implantação de algumas partes

normalmente usadas na chamada do sistema 'fork()' a sub-classes.

A classe de base abstrata 'Fork' tem as seguintes características:

- Define um membro de dados 'd_pid'. Este membro de dados conterá o ID do processo filho (no processo pai) e o valor 0 no processo filho:

```
class Fork
{
    int d_pid;
```

- Sua interface pública declara dois membros:
 - Uma função membro 'fork()', realiza a bifurcação (i.e., criará um (novo) processo filho);
 - Um destrutor virtual vazio '~Fork()', que pode ser ignorado pelas classes derivadas.

Eis a interface pública completa:

```
virtual ~Fork()
{
}
void fork();
```

Todas as funções membros restantes são declaradas na seção protegida da classe e só podem, portanto, serem usadas pelas classes derivadas. São elas:

- A função membro 'pid()', que permite às classes derivadas acessarem o valor de retorno do sistema 'fork()'

```
int pid()
{
    return d_pid;
}
```

- Um membro 'int waitForChild()', que pode ser chamada pelo processo pai para esperar a conclusão do processo filho (como discutido abaixo). Este membro está declarado na interface de classe. Sua implantação é:

```
#include "fork.ih"

int Fork::waitForChild()
{
    int status;
```

```

waitpid(d_pid, &status, 0);

return WEXITSTATUS(status);
}

```

Esta implantação simples retorna o estado de saída do filho ao pai. A função do sistema chamada 'waitpid()' bloqueia até que o filho termine;

- Quando a chamada ao sistema 'fork()' usa processo pai e processo filho sempre podem ser distinguidos. A distinção principal entre esses processos é que o 'd_pid' será igual ao ID do processo filho enquanto 'd_pid' será igual a 0 no processo pai. Como esses dois processos sempre podem ser distinguidos, é necessário que sejam implantados por classes derivadas de 'Fork'. Para reforçar este requerimento, o membro 'childProcess()', que define as ações do processo filho e 'parentProcess()', que define as ações do processo pai definimos como funções virtuais puras:

```

virtual void childProcess() = 0;    // both must be implemented
virtual void parentProcess() = 0;

```

- Adicionalmente, as comunicações entre os processos pai e filho podem usar 'streams' padrão ou outras facilidades, como 'pipes' (cf. seção 20.3.3). Para facilitar esta comunicação entre processos, as classes derivadas podem implantar:

- : Este membro pode ser implantado se qualquer 'stream' padrão ('cin', 'cout' ou 'cerr') no processo pai for redirecionada no processo filho (cf. seção 20.3.1);
- : Este membro pode ser implantado se qualquer 'stream' padrão ('cin', 'cout' ou 'cerr') no processo filho for redirecionada no processo pai.

O redirecionamento das 'streams' padrão será necessário se pai e filho podem se comunicar entre si via 'streams' padrão. Eis aqui as definições padrão dadas na interface de classe:

```

virtual void childRedirections()
{}
virtual void parentRedirections()
{}

```

A função membro 'fork()' chama a função do sistema 'fork()' (Atenção: Como a função do sistema 'fork()' é chamada pela função membro com o mesmo nome, o operador de resolução de escopo (::) deve ser usado para prevenir chamada recursiva à função membro). Depois da chamada a '::fork()', dependendo de seu valor de retorno, ou 'parentProcess()' ou 'childProcess()' é chamada. Talvez seja necessária redireção. A implantação de 'Fork::fork()' chama 'childRedirections()' justo antes da chamada a 'childProcess()' e 'parentRedirections()' justo antes de chamar 'parentProcess()':

```

#include "fork.ih"

void Fork::fork()
{

```

```

if ((d_pid = ::fork()) < 0)
    throw "Fork::fork() failed";

if (d_pid == 0)                // processo filho tem pid == 0
{
    childRedirections();
    childProcess();

    exit(1);                    // não deveria chegar aqui:
                                // childProcess() sairia
}

parentRedirections();
parentProcess();
}

```

Na classe 'fork.cc' o arquivo cabeçalho interno 'fork.ih' é incluído. Este arquivo cabeçalho cuida de inclusão dos arquivos cabeçalhos do sistema necessários, bem como da inclusão de 'fork.h'. Sua implantação é:

```

#include "fork.h"
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

```

O processo filho pode não retornar: Uma vez completadas suas tarefas pode terminar. Isto acontece automaticamente quando um processo filho realiza uma chamada a um membro da família 'exec...()', mas se o filho permanece ativo, então precisa assegurar-se de terminar com propriedade. Um processo filho normalmente usa 'exit()' para terminar, mas pode executar 'exit()' sem ativar destrutores de objetos definidos no mesmo nível ou noutro mais superficial que o da chamada a 'exit()'. Os destrutores de objetos definidos globalmente são ativados quando 'exit()' é usado. Quando um processo filho termina usando 'exit()', deve ou chamar uma função membro de suporte que defina todos os objetos aninhados que necessita ou pode definir todos os seus objetos num comando composto (p.ex., usando um bloco de lançamento) chamando 'exit()' abaixo do comando composto.

O processo pai deve normalmente esperar seu filho completar. Ao terminar o processo filho, este informa seu pai que está em vias de terminar, enviando um sinal que será apanhado pelo pai. Se o processo filho termina e o processo pai não recebe esse sinal então tal processo filho permanece visível como os ditos processos zumbi.

Se um processo pai precisa esperar o fim de seu filho, deve chamar o membro 'waitForChild()'. Este membro retorna o estado de saída do processo filho ao seu pai.

Existe uma situação em que o processo filho continua a viver, mas o pai morre. Na natureza

isto acontece todo o tempo: Os pais tendem a morrer antes de seus filhos. Em nosso contexto (i.e., C++), isto é chamado programa `daemon` (demônio): o processo pai morre e o programa filho continua executando como filho do processo inicial básico. Novamente, quando o filho eventualmente morre um sinal é enviado ao seu `padrasto` inicial. Aqui não é criado um zumbi, pois 'init' apanha o sinal de terminação de todos os seus enteados e filhos. A construção de um processo demônio é bem simples, dada a disponibilidade da classe 'Fork' (seção 20.3.2).

21.3.1: Re-direção re-visitada

Anteriormente, na seção 5.8.3 foi notado que num programa C++ é possível re-direcionar usando-se a função membro 'ios::rdbuf()'. Adjudicando o 'streambuf' de uma 'stream' a outra 'stream', ambos objetos 'stream' acessam o mesmo 'streambuf', realizando um re-direcionamento no nível da própria linguagem de programação.

É factível realizar isto dentro do contexto do programa C++, mas se sairmos desse contexto, a re-direção termina, já que o sistema operacional não conhece objetos 'streambuf'. Isto ocorre, p.ex., quando o programa usa uma chamada a 'system()' para começar um sub-programa. O programa no fim desta seção usa a re-direção C++ para re-direcionar a informação dentro de 'cout' para um arquivo e então chama:

```
system("echo hello world")
```

Para ecoar uma linha de texto bem conhecida. Como 'echo' escreve sua informação na saída padrão esta seria o arquivo de re-direção do programa se a re-direção da C++ pudesse ser reconhecida pelo sistema operacional.

Atualmente isto não acontece e 'hello world' ainda aparece na saída padrão do programa no lugar de no arquivo para o qual foi re-dirigido. Uma solução deste problema envolve uma re-direção a nível do sistema operacional, para o que alguns sistemas operacionais (p.ex., Unix e amigos) dispõem de chamadas ao sistema como 'dup()' e dup2(). Exemplos destas chamadas ao sistema são dados na seção 21.3.3.

Aqui está o exemplo da falha ao re-direcionarmos para o nível de sistema segundo C++ re-direção usando re-direção para 'streambuf':

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace::std;

int main()
```

```

{
    ofstream of("outfile");

    cout.rdbuf(of.rdbuf());
    cout << "To the of stream" << endl;
    system("echo hello world");
    cout << "To the of stream" << endl;
}
/*
Saída Gerada:

on the file 'outfile':
To the of stream
To the of stream

On standard output:
hello world
*/

```

21.3.2: O programa 'Daemon'

Existem aplicações onde o único propósito de 'fork()' é iniciar um processo filho. O processo pai termina imediatamente depois de criar o processo filho. Se isto acontecer, o processo filho continua correndo como processo filho de 'init', o primeiro processo em sistemas Unix que está sempre em execução. Tal processo é chamado frequentemente de demônio, correndo como um processo em segundo plano.

Apesar de que o exemplo seguinte pode ser construído como um programa pleno em C, foi incluído nas Anotações C++ porque está estreitamente relacionado à discussão corrente da classe 'Fork'. Pensei em adicionar um membro 'daemon()' a esta classe, mas decidi o contrário porque a construção de um programa demônio é muito simples e não requer características que já não estejam atualmente disponíveis na classe 'Fork'. Eis um exemplo que ilustra a construção de um programa demônio:

```

#include <iostream>
#include <unistd.h>
#include "fork.h"

class Daemon: public Fork
{
public:
    virtual void parentProcess()           // o pai não faz nada.
    {}

    virtual void childProcess()
    {
        sleep(3);                          // ações realizadas pelo filho
                                           // só uma mensagem...
    }
}

```

```

        std::cout << "Alô do processo filho\n";
        exit (0);                                // O processo filho sai.
    }
};

int main()
{
    Daemon daemon;

    daemon.fork();                                // o programa imediatamente retorna
    return 0;
}

/*
    Saída Gerada:
    A próxima espera por comando depois de 3 segundos:
    Alô do processo filho
*/

```

21.3.3: A classe 'Pipe'

A re-direção a nível do sistema envolve o uso de arquivos descritores, criado por chamadas ao sistema 'pipe()'. Quando dois processos querem se comunicar usando tais arquivos descritores, ocorre o seguinte:

- O processo constroi dois arquivos descritores associados usando chamadas ao sistema 'pipe()'. Um dos arquivos descritores é usado para escritura, o outro para leitura;
- O sistema 'fork' tem lugar (i.e., a função 'fork()' do sistema é chamada), duplicando os arquivos descritores. Agora temos quatro arquivos descritores já que ambos, o processo filho e o processo pai possuem suas próprias cópias dos dois arquivos descritores criados por 'pipe()';
- Um processo (digamos o processo pai) usará os arquivos descritores para leitura. Fechará seu arquivo descritor para escritura;
- O outro processo (digamos o filho) usará o arquivo descritor para escritura. Pode fechar seu arquivo descritor para leitura;
- Toda informação escrita pelo processo filho no arquivo descritor, pode agora ser lida pelo processo pai do seu arquivo descritor correspondente à leitura, isto estabelece um canal de comunicação entre os processos filho e pai.

Apesar de basicamente simples, podem facilmente insinuarem-se erros: os propósitos

dos arquivos descritores disponíveis aos dois processos (pai ou filho) podem ser confundidos facilmente. Para prevenir erros de controle, o controle pode ser definido corretamente uma vez e depois deixado a uma classe como a classe 'Pipe' construída aqui. Vejamos as características (antes que a implantação possa ser compilada, o compilador deve ter lido o cabeçalho da classe bem como o arquivo 'unistd.h'):

- A chamada ao sistema 'pipe()' espera um ponteiro a dois valores inteiros, que representarão, respectivamente, o arquivo descritor usado para acessar o fim da leitura e o fim da escrita do 'pipe' construído, depois de uma finalização bem sucedida de 'pipe()'. Para evitar confusões, um 'enum' é definido associando estes fins com constantes simbólicas. Ainda mais, a classe armazena os dois arquivos descritores num membro de dados 'd_fd'. Eis o cabeçalho e seus dados privados:

```
class Pipe
{
    enum    RW { READ, WRITE };
    int     d_fd[2];
```

- A classe só necessita um construtor padrão. Este construtor chama 'pipe()' para criar um conjunto de arquivos descritores associados usados para acessar ambos extremos de um 'pipe':

```
Pipe::Pipe()
{
    if (pipe(d_fd))
        throw "Pipe::Pipe(): pipe() failed";
}
```

- Os membros 'readOnly()' e 'readFrom()' são usados para configurar o extremo de leitura do 'pipe'. A segunda função é usada para por a re-direção, fornecendo um arquivo descritor alternativo que pode ser usado para ler do 'pipe'. Usualmente este arquivo descritor alternativo é 'STDIN_FILENO', que permite 'cin' extrair informação do 'pipe'. A função formadora é usada meramente para configurar o extremo de leitura do 'pipe': Fecha o extremo de escrita correspondente e retorna um arquivo descritor que pode ser usado para ler do 'pipe':

```
int Pipe::readOnly()
{
    close(d_fd[WRITE]);
    return d_fd[READ];
}
void Pipe::readFrom(int fd)
{
    readOnly();

    redirect(d_fd[READ], fd);
    close(d_fd[READ]);
}
```

- Estão disponíveis os membros 'writeOnly()' e dois 'writtenBy()' para configurar o extremo de

escritura do 'pipe'. A função formadora é usada só para configurar o extremo de escritura do 'pipe': Fecha o extremo de leitura correspondente e retorna um arquivo descritor que pode ser usado para escrever no 'pipe':

```
int Pipe::writeOnly()
{
    close(d_fd[READ]);
    return d_fd[WRITE];
}
void Pipe::writtenBy(int fd)
{
    writtenBy(&fd, 1);
}
void Pipe::writtenBy(int const *fd, size_t n)
{
    writeOnly();

    for (size_t idx = 0; idx < n; idx++)
        redirect(d_fd[WRITE], fd[idx]);

    close(d_fd[WRITE]);
}
```

Para o último membro duas versões sobrecarregadas estão disponíveis:

- 'writtenBy(int fileDescriptor)': Usado para configurar re-direções unitárias, tal que um arquivo descritor específico (usualmente STDOUT_FILENO ou STDERR_FILENO) usado para escrever no 'pipe';
- 'writtenBy(int *fileDescriptor, size_t n = 2)': Usado para configurar re-direções múltiplas, fornecendo como argumento um conjunto contendo arquivos descritores. As informações escritas em qualquer destes arquivos descritores é escrita no 'pipe'.
- A classe possui um membro de dados privado, 'redirect()', que é usado para definir um re-direcionamento usando a chamada ao sistema 'dup2()'. Esta função espera dois arquivos descritores. O primeiro representa um arquivo descritor que usado para acessar a informação do dispositivo, o segundo é um arquivo descritor alternativo também usado para acessar a informação do dispositivo uma vez que 'dup2()' completou com sucesso. Eis a implantação de 'redirect()':

```
void Pipe::redirect(int d_fd, int alternateFd)
{
    if (dup2(d_fd, alternateFd) < 0)
        throw "Pipe: redirection failed";
}
```

Agora essa re-direção pode ser configurada facilmente usando um ou mais objetos 'pipe',

usaremos agora 'Fork' e 'Pipe' em diversos programas de demonstração.

21.3.4: A classe 'ParentSlurp'

A classe 'ParentSlurp', derivada de 'Fork', começa um processo filho que executa um programa (como '/bin/ls'). A saída (padão) do programa 'execed' é então lida pelo processo pai. O processo pai (para fins de demonstração) escreverá as linhas recebidas na sua 'stream' de saída padrão, enquanto agrega os números de linha nas linhas recebidas. O mais conveniente aqui é re-direcionar a 'stream' de entrada estandarte, assim o pai pode ler a saída do processo filho de sua 'stream' de entrada 'std::cin'. Para isso o único 'pipe' usado é usado como um 'pipe' de entrada do pai e um 'pipe' de saída do filho.

A classe 'ParentSlurp' tem as seguinte características:

- É derivada de 'Fork'. Antes de começar a interface de classe de 'ParentSlurp', o compilador precisa ter lido 'fork.h' e 'pipe.h'. Ainda mais, a classe só usa um membro de dados: Um objeto 'd_pipe':

```
class ParentSlurp: public Fork
{
    Pipe    d_pipe;
```

- Como o construtor de 'Pipe' automaticamente constroi um 'pipe' e como 'd_pipe' é automaticamente construído pelo construtor padrão de 'ParentSlurp', não há necessidade de definir explicitamente o construtor de 'ParentSlurp'. Como não se necessita implantar um construtor, todos os membros de 'ParentSlurp' podem ser declarados como protegidos;
- O membro 'childRedirections()' configura o 'pipe' como 'pipe' de leitura. Assim, toda informação escrita na 'stream' de saída do filho terminará no 'pipe'. A grande vantagem disto é que não se necessita de 'streams' nos arquivos descritores para escrever num arquivo descritor:

```
virtual void childRedirections()
{
    d_pipe.writtenBy(STDOUT_FILENO);
}
```

- O membro 'parentRedirections()', configura seu extremo do 'pipe' como 'pipe' de leitura. O faz re-direcionando o extremo de leitura do 'pipe' para sua entrada padrão, o arquivo descritor ('STDIN_FILENO'), permitindo, assim, extrair de 'cin' no lugar de usar uma 'stream' construída para o arquivo descritor:

```
virtual void parentRedirections()
{
```

```

        d_pipe.readFrom(STDIN_FILENO);
    }

```

- O membro 'childProcess()' tem que se concentrar em suas próprias ações. Como só tem que executar um programa (que escreve informações em sua saída estandarte), o membro consiste de um único comando:

```

virtual void childRedirections()
{
    d_pipe.writtenBy(STDOUT_FILENO);
}

```

- O membro 'parentProcess()' simplesmente apanha a informação que aparece em sua entrada estandarte. Fazendo isto, lê a saída do filho. Copia as linhas recebidas para sua 'stream' de saída estandarte, depois de numerar as linhas:

```

void ParentSlurp::parentProcess()
{
    std::string    line;
    size_t        nr = 1;

    while (getline(std::cin, line))
        std::cout << nr++ << ": " << line << std::endl;

    waitForChild();
}

```

O seguinte programa simplesmente constroi um objeto 'ParentSlurp' e chama seu membro 'fork()'. Sua saída consiste numa lista numerada de arquivos do diretório onde o programa partiu. Note que o programa também necessita dos arquivos objeto 'fork.o', 'pipe.o' e 'waitforchild.o' (veja as fontes anteriores):

```

int main()
{
    ParentSlurp ps;

    ps.fork();
    return 0;
}
/*
Saída Gerada (somente como exemplo, a saída obtida pode diferir):

1: a.out
2: bitand.h
3: bitfunctional
4: bitnot.h
5: daemon.cc
6: fdinseek.cc
7: fdinseek.h
...

```

21.3.5: Comunicando-se com múltiplos filhos

O próximo passo ladeira acima é construir um processo filho monitor. Aqui o processo pai é responsável por todos seus processos filhos, mas também tem que ler suas saídas estandartes. O usuário pode entrar informação pela entrada estandarte do processo pai, para o que é definida uma linguagem simples:

- Inicialmente lançará um novo processo filho. O pai retornará o ID (um número) ao usuário. O ID será usado para enviar mensagens a aquele processo filho particular;
- - '`<nr> text`' enviará "`text`" ao processo filho com ID '`<nr>`';
- - '`stop <nr>`' terminará o processo filho com ID '`<nr>`';
- - '`exit`' terminará o pai bem como todos seus filhos.

Além disso, o processo filho que não receba texto por algum tempo reclamará, enviando uma mensagem ao processo pai. O processo pai então simplesmente retransmitirá a mensagem recebida ao usuário, copiando-a à sua 'stream' de saída.

Um problema com programas como nosso monitor é que esses programas permitem entradas assíncronas de múltiplas fontes: As entradas aparecerão na entrada estandarte bem como nos extremos de entrada dos 'pipes'. Também são usados múltiplos canais de saída. Para manipular situações como estas, o sistema 'select()' de chamadas foi desenvolvido.

21.3.5.1: A classe 'Select'

O sistema de chamadas 'select()' foi desenvolvido para manipular entradas e saídas multiplexadas assíncronas.

Este sistema de chamadas pode ser usado para manipular, p.ex., entradas que apareçam simultaneamente num conjunto de arquivos descritores.

A função do sistema 'select()' é mais complexa e sua discussão completa está além dos objetivos das Anotações C++. Contudo seu uso pode ser simplificado fornecendo uma classe 'Selector', escondendo seus detalhes e oferecendo uma interface pública fácil de usar. Aqui suas características são discutidas:

- A maioria dos membros são muito pequenos, permitindo-nos definir a maioria de suas funções membro em linha. A classe requer bem poucos membros de dados. A maioria deles de tipos

construídos para uso em 'select()'. Por isso antes da interface da classe poder ser manipulada pelo compilador, vários arquivos cabeçalhos devem ser lidos por ele:

```
#include <limits.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
```

- A definição da classe e seus membros de dados aparecem em seguida. O tipo de dados 'fd_set' é um tipo projetado para ser usado em 'select()' e as variáveis desse tipo contêm o conjunto de arquivos descritores sobre os quais 'select()' detetou alguma atividade. Além disso, 'select()' nos permite lançar um alarme assíncrono. Para especificar os tempos desse alarme, a classe recebe o membro de dados 'timeval'. Os membros restantes são usados pela classe para controle interno, ilustrado abaixo. Eis o cabeçalho da classe e seus membros de dados:

```
class Selector
{
    fd_set      d_read;
    fd_set      d_write;
    fd_set      d_except;
    fd_set      d_ret_read;
    fd_set      d_ret_write;
    fd_set      d_ret_except;
    timeval     d_alarm;
    int         d_max;
    int         d_ret;
    int         d_readidx;
    int         d_writeidx;
    int         d_exceptidx;
```

As seguintes funções membro estão localizadas na interface pública da classe:

- - 'Selector()': O construtor padrão. Limpa a leitura, escritura e executa as variáveis 'fd_set', e desliga o alarme. Exceto por 'd_max' os restantes membros de dados não requerem iniciação. Eis a implantação do construtor de 'Selector':

```
Selector::Selector()
{
    FD_ZERO(&d_read);
    FD_ZERO(&d_write);
    FD_ZERO(&d_except);
    noAlarm();
    d_max = 0;
}
```

- - 'int wait()': Esta função membro bloqueará com 'block()' até que sensore atividade em qualquer dos arquivos descritores monitorados pelo objeto 'Selector' ou se expira o tempo de

alarme. Lançará uma exceção quando o sistema de chamadas 'select()' falhe. Aqui está a implantação de 'wait()':

```
int Selector::wait()
{
    timeval t = d_alarm;

    d_ret_read = d_read;
    d_ret_write = d_write;
    d_ret_except = d_except;

    d_readidx = 0;
    d_writeidx = 0;
    d_exceptidx = 0;

    d_ret = select(d_max, &d_ret_read, &d_ret_write, &d_ret_except, &t);

    if (d_ret < 0)
        throw "Selector::wait()/select() failed";

    return d_ret;
}
```

- - 'int nReady': O valor de retorno desta função membro é definido somente quando 'wait()' retornou. Nesse caso ela retorna 0 para tempo do alarme expirado, -1 se 'select()' falha e o número do arquivo descritor onde se sentiu atividade. Pode ser implantada em linha:

```
int nReady()
{
    return d_ret;
}
```

- - 'int readFd()': o valor de retorno desta função membro também é definido depois que 'wait()' retornou. Seu valor de retorno é -1 se não há (mais) arquivos descritores de entrada disponíveis. De outra forma o seguinte arquivo descritor disponível para leitura é retornado. Sua implantação em linha é:

```
int readFd()
{
    return checkSet(&d_readidx, d_ret_read);
}
```

- - 'int writeFd()': Opera analogamente a 'readFd()', retorna o próximo arquivo descritor onde haja saída escrita. Usando 'd_writeidx' e 'd_ret_read', é implementada analogamente a 'readFd()';

- - 'int exceptFd()': opera analogamente a 'readFd()', retorna o arquivo descritor de exceção seguinte onde houve atividade. Usa 'd_except_idx' e 'd_ret_except', é implementada analogamente a 'readFd()';
- - 'void setAlarm(int sec, int usec = 0)': Esta função membro ativa o alarme de 'Select'. Por fim o número de segundos a esperar para desativar o alarme deve ser especificado. Simplesmente coloca valores nos campos de 'd_alarm'. Então na próxima chamada a 'Select::wait()' o alarme será disparado (i.e., 'wait()' retorna 0) uma vez que o intervalo de alarme configurado expirou. Eis sua implantação em linha:

```
void setAlarm(int sec, int usec = 0)
{
    d_alarm.tv_sec  = sec;
    d_alarm.tv_usec = usec;
}
```

- - 'void noAlarm()': Este membros desliga o alarma, simplesmente definindo o intervalo de alarme a um valor muito longo. Implantada em linha fica:

```
void noAlarm()
{
    setAlarm(INT_MAX, INT_MAX);
}
```

- - 'void addReadFd(int fd)': Agrega um arquivo descritor ao conjunto de entrada de arquivos descritores monitorados pelo objeto 'Selector'. A função membro 'wait()' retornará uma vez esteja disponível uma entrada no arquivo descritor indicado. Eis sua implantação em linha:

```
void addReadFd(int fd)
{
    addFd(&d_read, fd);
}
```

- - 'void addWriteFd(int fd)': Agrega um arquivo descritor ao conjunto de arquivos descritores monitorados pelo objeto 'Selector'. A função membro 'wait()' retornará uma vez esteja disponível uma saída no arquivo descritor indicado. Usando 'd_write', é implantada analogamente a 'addReadFd()';
- - 'void addExceptFd(int fd)': Agrega um arquivo descritor ao conjunto de arquivos descritores de exceção para ser monitorado pelo objeto 'Selector'. A função membro 'wait()' retorna uma vez sensorada atividade no arquivo descritor indicado. Usando 'd_except' é implantada analogamente

a 'addReadFd()';

- - 'void rmReadFd(int fd)': Remove um arquivo descritor do conjunto de arquivos descritores de entrada monitorados pelo objeto 'Selector'. Eis sua implantação em linha:

```
void rmReadFd(int fd)
{
    FD_CLR(fd, &d_read);
}
```

- - 'void rmWriteFd(int fd)': Remove um arquivo descritor do conjunto de arquivos descritores de saída monitorados pelo objeto 'Selector'. Usando 'd_wrtite' é implantada analogamente a 'rmReadFd()';
- - 'void rmExceptFd(int fd)': Remove um arquivo descritor do conjunto de arquivos descritores monitorados pelo objeto 'Selector'. Usando 'd_except' sua implantação é análoga à de 'rmReadFd()';

Os dois membros restantes da classe são membros de suporte e não podem ser usados por funções que não sejam membros. Por isso estão declarados na seção privada:

- - O membro 'addFd()' adiciona um certo arquivo descritor a um certo 'fd_set'. Eis sua implantação:

```
void Selector::addFd(fd_set *set, int fd)
{
    FD_SET(fd, set);
    if (fd >= d_max)
        d_max = fd + 1;
}
```

- - O membro 'checkSet()' testa se um certo arquivo descritor (*index) está em certo 'fd_set'. Eis sua implantação:

```
int Selector::checkSet(int *index, fd_set &set)
{
    int &idx = *index;

    while (idx < d_max && !FD_ISSET(idx, &set))
        ++idx;

    return idx == d_max ? -1 : idx++;
}
```



```
}
```

21.3.5.2: A classe *'Monitor'*

O programa monitor usa um objeto 'Monitor' para realizar a maior parte do trabalho. A classe possui somente um construtor e um membro público, 'run()', para realizar suas tarefas. Por isso todas as outras funções membro descritas abaixo estão declaradas na seção privada da classe.

'Monitor' define uma enumeração 'Commands', que lista simbolicamente os vários comandos que sua linguagem de entrada suporta, bem como muitos membros de dados, entre os quais um objeto 'Selector', um mapa usando os números de ordem dos filhos como chave e um apontador a objetos 'Child' (veja seção 20.3.5.3) como seus valores. Além disso 'Monitor' possui um conjunto estático como membro 's_handler[]' que guarda apontadores a funções membro que manipulam comandos do usuário.

Um destrutor poderia ter sido implantado também, mas sua implantação foi deixada como exercício ao leitor. Antes da interface de classe poder ser processada pelo compilador, precisa ter visto 'select.h' e 'child.h'. Eis seu cabeçalho de classe, incluindo sua seção privada:

```
class Monitor
{
    enum Commands
    {
        UNKNOWN,
        START,
        EXIT,
        STOP,
        TEXT
    };

    static void (Monitor::*s_handler[])(int, std::string const &);

    Selector          d_selector;
    int               d_nr;
    std::map<int, Child *> d_child;
```

Como há só um tipo de dados sem classe, o construtor da classe é muito curto e pode ser implantado em linha:

```
Monitor()
:
    d_nr(0)
{ }
```

O núcleo das atividades de 'Monitor' é feito por 'run()'. Esta realiza as seguintes tarefas:

- Evita zumbis, precisa ser capaz de agarrar os sinais de terminação de seus filhos. Os sinais de terminação são tomados pelo membro 'waitForChild()'. Este membro é instalado pela 'run()' e espera a finalização do filho. Uma vez que o filho completou, se re-instala assim o próximo sinal de terminação também pode ser recebido. Eis 'waitForChild()':

```
void Monitor::waitForChild(int signum)
{
    int status;
    wait(&status);

    signal(SIGCHLD, waitForChild);
}
```

- Inicialmente, o objeto 'Monitor' aguarda só em sua entrada padrão: o conjunto de arquivos descritores de entrada onde 'd_selector' aguarda é iniciado por 'STDIN_FILENO'.
- Então, num laço a função 'wait()' de 'd_selector' é chamada. Se a entrada em 'cin' está disponível, é processada por 'processInput()'. De outra forma, a entrada que chegou do processo filho. As informações enviadas pelos filhos são processadas por 'processChild()';

Eis a implantação de 'run()':

```
#include "monitor.ih"

void Monitor::run()
{
    signal(SIGCHLD, waitForChild);
    d_selector.addReadFd(STDIN_FILENO);

    while (true)
    {
        cout << "? " << flush;
        try
        {
            d_selector.wait();

            int fd;
            while ((fd = d_selector.readFd()) != -1)
            {
                if (fd == STDIN_FILENO)
                    processInput();
                else
                    processChild(fd);
            }
        }
        catch (...)
        {
            cerr << "select failed, exiting\n";
            exiting();
        }
    }
}
```

```

    }
}
}

```

A função membro 'processInput()' lê os comandos entrados pelo usuário via a 'stream' de entrada estandarte do programa. O próprio membro é simples: Chama 'next()' para obter o próximo comando entrado pelo usuário e então chama a função correspondente via o elemento correspondente do conjunto 's_handler[]'. O conjunto e os membros 'processInput()' e 'next()' são definidos como segue:

```

void (Monitor::*Monitor::s_handler[])(int, string const &) =
{
    &Monitor::unknown,           // ordem seguida pelos elementos de
    &Monitor::createNewChild,     // enum Command's
    &Monitor::exiting,
    &Monitor::stopChild,
    &Monitor::sendChild,
};

```

```

void Monitor::processInput()
{
    string line;
    int value;
    Commands cmd = next(&value, &line);
    (this->*s_handler[cmd])(value, line);
}

```

```

Monitor::Commands Monitor::next(int *value, string *line)
{
    if (!getline(cin, *line))
        throw "Command::next(): reading cin failed";

    if (*line == "start")
        return START;

    if (*line == "exit")
        return EXIT;

    if (line->find("stop") == 0)
    {
        istringstream istr(line->substr(4));
        istr >> *value;
        return !istr ? UNKNOWN : STOP;
    }
    istringstream istr(line->c_str());
    istr >> *value;
    if (istr)
    {

```

```

        getline(istr, *line);
        return TEXT;
    }
    return UNKNOWN;
}

```

Todas outras entradas sensoradas por 'd_select' foram criadas por processos filhos. Como o membro readFd() de 'd_select' retorna o arquivo descritor de entrada correspondente, este descritor pode ser passado ao processo filho através de 'processChild()'. Então, usando um 'ifdstreambuf' (veja seção 21.1.2.1), sua informação é lida pela 'stream' de entrada. O protocolo de comunicação usado aqui é bem básico: Para cada linha de entrada enviada a um filho, o filho envia exatamente uma linha de texto em retorno. Conseqüentemente, 'processChild()' tem que ler só uma linha de texto:

```

void Monitor::processChild(int fd)
{
    ifdstreambuf ifdbuf(fd);
    istream istr(&ifdbuf);
    string line;

    getline(istr, line);
    cout << d_child[fd]->pid() << ": " << line << endl;
}

```

Por favor note a construção 'd_child[fd]->pid()' usada na fonte acima. 'Monitor' define o membro de dados 'map<int, Child *> d_child'. Este mapa contém o número de ordem de 'child' como sua chave e um apontador ao objeto 'Child' como seu valor. Um apontador é usado aqui, melhor que um objeto 'Child', pois queremos usar as facilidades oferecidas pelo mapa, mas não queremos copiar um objeto 'Child'.

A implicação do uso de ponteiros como valores do mapa é, claro está, que a responsabilidade da destruição do objeto 'Child', uma vez que se torne supérfluo, recai agora sobre o programador e não mais no sistema de suporte em tempo de execução.

Agora que a implantação de 'run()' foi vista, nos concentraremos nos vários comandos que o usuário pode entrar:

- Quando o comando 'start' sai, um novo processo filho começa. Um novo elemento é agregado a 'd_child' pelo membro 'createNewChild()'. Em seguida o objeto 'Child' pode começar suas atividades, mas o objeto 'Monitor' não pode esperar aqui esperando que o processo filho complete suas atividades, já que não há um ponto final bem definido no futuro próximo e o usuário talvez queira entrar novos comandos. Por isso o processo filho executará como demônio: Seu processo pai terminará imediatamente e seu próprio processo filho continuará em segundo plano. Conseqüentemente 'createNewChild()' chama o membro 'fork()' do filho. Apesar de que é a função 'fork()' do filho que é chamada, é a 'fork()' do programa monitor que é chamada. Assim o

programa monitor é duplicado por 'fork()'. A execução então continua:

- Em 'parentProcess()' do filho em seu processo pai;
- Em 'childProcess()' do filho em seu processo filho.

Como 'parentProcess()' do filho é uma função vazia, retornado imediatamente, o processo pai de 'Child' efetivamente continua imediatamente abaixo do comando 'cp->fork()' de 'createNewChild()'. Como o processo filho jamais retorna (veja seção 20.3.5.3), o código abaixo de 'cp->fork()' nunca é executado pelo processo filho de 'Child'. Isto é exatamente o que deve acontecer.

No processo pai, o código restante de 'createNewChild()' simplesmente agrega um arquivo descritor disponível para leitura de informações do filho ao conjunto de arquivos descritores de entrada monitorados por 'd_select' e usa 'd_child' para estabelecer a associação entre esse arquivo descritor e o endereço do objeto 'Child':

```
void Monitor::createNewChild(int, string const &)\n{\n    Child *cp = new Child(++d_nr);\n\n    cp->fork();\n\n    int fd = cp->readFd();\n\n    d_selector.addReadFd(fd);\n    d_child[fd] = cp;\n\n    cerr << "Child " << d_nr << " started\\n";\n}
```

- Os comandos 'stop <nr>' e '<nr> text' requerem comunicação direta com o filho. O comando precedente termina o processo filho '<nr>', chamando 'stopChild()'. Esta função encontra o processo filho com o número de ordem usando um objeto anônimo 'Find', aninhada dentro de 'Monitor'. A classe 'Find' simplesmente compara o 'nr' fornecido com o número de ordem do filho retornado pelo seu membro 'nr()':

```
class Find\n{\n    int      d_nr;\n    public:\n        Find(int nr)\n        :\n            d_nr(nr)\n        {}\n        bool operator()(std::map<int, Child *>::value_type &vt)\n                                     const\n        {\n
```

```

        return d_nr == vt.second->nr();
    }
};

```

Se o filho com número de ordem 'nr' for encontrado, seu arquivo descritor é removido do conjunto de arquivos descritores de entrada 'd_selector'. Então o processo filho é terminado pelo membro estático 'killChild()'. O membro 'killChild()' é declarado como função membro estática, pois usa como argumento a função do algoritmo genérico 'for_each()' por 'erase()' (veja abaixo). Eis a implantação de 'killChild()':

```

void Monitor::killChild(map<int, Child *>::value_type it)
{
    if (kill(it.second->pid(), SIGTERM))
        cerr << "Couldn't kill process " << it.second->pid() << endl;
}

```

- Tendo terminado o processo filho especificado, o objeto 'Child' correspondente é destruído e seu ponteiro é removido de 'd_child':

```

void Monitor::stopChild(int nr, string const &)
{
    map<int, Child *>::iterator it =
        find_if(d_child.begin(), d_child.end(), Find(nr));

    if (it == d_child.end())
        cerr << "No child number " << nr << endl;
    else
    {
        d_selector.rmReadFd(it->second->readFd());

        killChild(*it);

        delete it->second;
        d_child.erase(it);
    }
}

```

- O comando '<nr> text' enviará um texto ao processo filho 'nr', usando a função membro 'sendChild()'. Esta função também usará um objeto 'Find' para localizar o processo com número de ordem 'nr' e então simplesmente inserirá o texto no extremo de escritura do 'pipe' conectado ao processo filho indicado:

```

void Monitor::sendChild(int nr, string const &line)
{
    map<int, Child *>::iterator it =
        find_if(d_child.begin(), d_child.end(), Find(nr));

    if (it == d_child.end())
        cerr << "No child number " << nr << endl;
    else

```

```

    {
        ofdnstreambuf ofdn(it->second->writeFd());
        ostream out(&ofdn);

        out << line << endl;
    }
}

```

- Quando o usuário entrar 'exit', o membro 'exiting()' é chamado. Este termina todos os processos filho, visitando todos os elementos de 'd_child', usando o algoritmo genérico 'for_each()' (seção 17.4.17). O programa em seguida é terminado:

```

void Monitor::exiting(int, string const &)
{
    for_each(d_child.begin(), d_child.end(), killChild);
    exit(0);
}

```

Finalmente a função 'main()' do programa é:

```

#include "monitor.h"

int main()
{
    Monitor monitor;

    monitor.run();
}

/*
Exemplo de uma sessão:

# a.out
? start
Child 1 started
? 1 hello world
? 3394: Child 1:1:  hello world
? 1 hi there!
? 3394: Child 1:2:  hi there!
? start
Child 2 started
? 3394: Child 1: standing by
? 3395: Child 2: standing by
? 3394: Child 1: standing by
? 3395: Child 2: standing by
? stop 1
? 3395: Child 2: standing by
? 2 hello world
? 3395: Child 2:1:  hello world
? 1 hello world
No child number 1

```

```

? exit3395: Child 2: standing by
?
#
*/

```

21.3.5.3: A classe 'Child'

Quando o objeto 'Monitor' parte um processo filho, tem que criar um objeto da classe 'Child'. A classe 'Child' é derivada da classe 'Fork', permitindo sua construção como um demônio, como discutido na seção anterior. Como o objeto 'Child' é um demônio, sabemos que seu processo pai pode ser definido como uma função vazia, seu 'childProcess()' precisa, ainda assim, ser definido. Aqui estão as características da classe 'Child':

- A classe 'Child' define dois membros de dados 'Pipe', para permitir a comunicação entre os processos pai e filho. Como estes 'pipes' são usados pelo processo filho 'Child', seus nomes estão dirigidos ao processo filho: O processo filho lê de 'd_in' e escreve em 'd_out'. Aqui está o cabeçalho da classe 'Child' e seus dados privados:

```

class Child: public Fork
{
    Pipe          d_in;
    Pipe          d_out;

    int           d_parentReadFd;
    int           d_parentWriteFd;
    int           d_nr;
}

```

- O construtor de 'Child' simplesmente guarda seu argumento, o número de ordem de um processo filho, em seu membro de dados 'd_nr':

```

Child(int nr)
:
    d_nr(nr)
{}

```

O processo filho 'Child' obtém sua informação de sua 'stream' de entrada estandarte e escreve sua informação em sua 'stream' de saída estandarte. Como os canais de comunicação são 'pipes', as re-direções precisam ser configuradas. Aqui está a implantação do membro 'childRedirections()':

```

void Child::childRedirections()
{
    d_in.readFrom(STDIN_FILENO);
    d_out.writeTo(STDOUT_FILENO);
}

```


- Apesar de que o processo pai não realiza ações, precisa configurar algumas re-direções. Como os nomes dos 'pipes' indicam suas funções no processo filho, 'd_in' é usado para escrever ao pai e 'd_out' é usado para ler pelo pai. Aqui está a implantação de 'parentRedirections()':

```
void Child::parentRedirections()
{
    d_parentReadFd = d_out.readOnly();
    d_parentWriteFd = d_in.writeOnly();
}
```

- O objeto 'Child' existirá até ser destruído pelo membro de 'Monitor' 'stopChild()'. Permitindo seu criador, o objeto 'Monitor', acessar os extremos do 'pipe' do pai, o objeto 'Monitor' pode se comunicar com o processo filho 'Child' via os extremos desse 'pipe'. Os membros 'readFd()' e 'writeFd()' permitem ao objeto 'Monitor' acessar esses extremos do 'pipe':

```
int readFd()
{
    return d_parentReadFd;
}
int writeFd()
{
    return d_parentWriteFd;
}
```

- O objeto 'Child' do processo filho tem basicamente duas tarefas a realizar:
 - Precisa responder à informação que apareça em sua 'stream' de entrada estandarte;
 - Se não aparecer informação depois de certo tempo (as implantações usam um intervalo de cinco segundos), então uma mensagem, de todas formas, será escrita em sua saída estandarte.

Para implantar este comportamento, 'childProcess()' define um objeto 'Selector' local, agregando 'STDIN_FILENO' ao seu conjunto de entradas de arquivos descritores monitorados.

Então, num laço eterno, 'childProcess()' espera por 'selector.wait()' retornar. Quando o alarme é disparado envia uma mensagem à sua saída estandarte (portanto ao 'pipe' de escritura). De outra forma ecoa as mensagens que aparecem em sua entrada estandarte em sua saída estandarte. Aqui está a implantação do membro 'childProcess()':

```
void Child::childProcess()
{
    Selector    selector;
    size_t     message = 0;

    selector.addReadFd(STDIN_FILENO);
    selector.setAlarm(5);
}
```

```

while (true)
{
    try
    {
        if (!selector.wait())          // timeout
            cout << "Child " << d_nr << ": standing by\n";
        else
        {
            string line;
            getline(cin, line);
            cout << "Child " << d_nr << ":" << ++message << ": " <<
                line << endl;
        }
    }
    catch (...)
    {
        cout << "Child " << d_nr << ":" << ++message << ": " <<
            "select() failed" << endl;
    }
}
exit(0);
}

```

- Os dois acessórios seguintes permitem ao objeto 'Monitor' acessar o processo 'Child' ID e número de ordem, respectivamente:

```

int pid()
{
    return Fork::pid();
}
int nr()
{
    return d_nr;
}

```

21.4: Objetos funções que realizam operações sobre bits

Na seção 17.1 vários tipos predefinidos de objetos funções foram introduzidos. Existem objetos funções que realizam operações aritméticas, relacionais e lógicas, correspondendo a uma multitude de operadores binários e unários.

Alguns operadores parece faltarem: Parece não haver objetos funções predefinidos que correspondam a operações sobre bits. Contudo, sua construção, dada a disponibilidade de objetos funções predefinidas, não é difícil. Os exemplos seguintes mostra um modelo de classe que implanta um objeto função que chama um operador sobre bits e ('operator&()), e um modelo de classe que implanta um objeto função chamando o operador unário de negação ('operator~()). Deixamos ao leitor a construção de

objetos funções similares para os demais operadores.

Aqui está a implantação do objeto função que chama o operador sobre bits 'operator&()':

```
#include <functional>

template <typename _Tp>
struct bit_and: public std::binary_function<_Tp,_Tp,_Tp>
{
    _Tp operator()(const _Tp& __x, const _Tp& __y) const
    {
        return __x & __y;
    }
};
```

Esta é a implantação do objeto função que chama 'operator~()':

```
#include <functional>

template <typename _Tp>
struct bit_not: public std::unary_function<_Tp,_Tp>
{
    _Tp operator()(const _Tp& __x) const
    {
        return ~__x;
    }
};
```

Estas e outros objetos funções predefinidos que faltam estão implantados no arquivo 'bitfuncional', que se encontra no arquivo 'cplusplus.yo.zip'.

Eis um exemplo que usa 'bit_and()' removendo todos os números ímpares de um vetor de valores inteiros:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "bitand.h"
using namespace std;

int main()
{
    vector<int> vi;

    for (int idx = 0; idx < 10; ++idx)
        vi.push_back(idx);

    copy
    (
        vi.begin(),
        remove_if(vi.begin(), vi.end(), bind2nd(bit_and<int>(), 1)),
```

```

        ostream_iterator<int>(cout, " ")
    );
    cout << endl;
}
/*
Saída Gerada:

0 2 4 6 8
*/

```

21.5: Implantando um 'reverse_iterator'

Anteriormente, na seção 19.11.1 a construção de iteradores e iteradores reversos foi discutida. Naquela seção o iterador foi construído como uma classe 'inner' numa classe derivada de um vetor de apontadores a 'strings' (abaixo essa classe derivada será referida como 'a classe derivada')

Um objeto dessa classe aninhada de iterador manipula a de-referenciação dos ponteiros guardados no vetor. Isto nos permitiu ordenar as 'strings' apontadas por esses elementos de 'vector' antes que os ponteiros.

Um retrazo da solução tomada na seção 19.11.1 é que a classe que implanta o iterador está ligada estreitamente à classe derivada como a classe 'iterator' estava implantada como classe aninhada. Que aconteceria se quiséssemos fornecer uma classe derivada de uma classe 'container' que guarda ponteiros com um iterador que manipula a de-referenciação dos ponteiros?

Nesta seção uma variante da solução anterior (classe aninhada) é discutida. A classe 'iterator' será definida como um modelo de classe, parametrizando o tipo de dados para os quais os elementos do recipiente aponta, bem como o tipo do iterador do recipiente. Mais uma vez implantaremos um 'RandomIterator' já que é o tipo de iterador mais complexo.

Nossa classe se chamará 'RandomPtrIterator', indicando que é um iterador aleatório operando sobre valores ponteiros. O modelo de classe define três modelos de tipos de parâmetros:

- O primeiro parâmetro especifica o tipo de classe ('Class'). Como a classe aninhada anterior, o construtor de 'RandomPtrIterator' será privado. Para permitir às classes clientes construírem 'RandomPtrIterator' necessitaremos, portanto, declarações de amiga. Contudo uma classe 'Class' amiga não pode ser definida: Modelos de tipos de parâmetro não podem ser usados em declarações de classes amigas. Mas isto não é um grande problema: Não todos os membros da classe cliente necessitam construir iteradores. Usando a declaração amiga do primeiro modelo de parâmetro pode ser especificado para 'begin()' e 'end()' dos membros da cliente;

- O segundo modelo de parâmetro parametriza o iterador do tipo de recipiente ('BaseIterator');
- O terceiro modelo de parâmetro indica o tipo de dados para o qual os ponteiros apontam (Type).

'RandomPtrIterator' usa um elemento de dados privado, um 'BaseIterator'. Eis a seção inicial, incluindo o construtor da classe 'RandomPtrIterator':

```
#include <iterator>

template <typename Class, typename BaseIterator, typename Type>
class RandomPtrIterator:
    public std::iterator<std::random_access_iterator_tag, Type>
{
    friend RandomPtrIterator<Class, BaseIterator, Type> Class::begin();
    friend RandomPtrIterator<Class, BaseIterator, Type> Class::end();

    BaseIterator d_current;

    RandomPtrIterator(BaseIterator const &current)
    :
        d_current(current)
    {}
}
```

Dissecando suas declarações de amiga, vemos que os membros 'begin()' e 'end()' da classe 'Class', retornam um objeto 'RandomPtrIterator' para os tipos 'Class', 'BaseIterator' e 'Type' garantem o acesso ao construtor privado de 'RandomPtrIterator'. Isto é exatamente o que queremos. Note que 'begin()' e 'end()' são declaradas como amigas limites.

Todos os demais membros de 'RandomPtrIterator' são públicos. Como 'RandomPtrIterator' é uma generalização da classe aninhada 'iterator' desenvolvida na seção 19.11.1, re-implantar as funções membros requeridos é fácil e só requer mudar 'iterator' por 'RandomPtrIterator' e mudar 'std::string' por 'Type'. Por exemplo, 'operator<()', definida na classe 'iterator' como:

```
bool operator<(iterator const &other) const
{
    return **d_current < **other.d_current;
}
```

É re-implantada como:

```
bool operator<(RandomPtrIterator const &other) const
{
    return **d_current < **other.d_current;
}
```

Como segundo exemplo: O 'operator*'() é definido na classe 'iterator' como:

```

std::string &operator*() const
{
    return **d_current;
}

```

E re-implantada como:

```

Type &operator*() const
{
    return **d_current;
}

```

Re-implantar a classe 'StringPtr' desenvolvida na seção 19.11.1 também não é difícil. Aparte de incluir o arquivo cabeçalho que define o modelo da classe 'RandomPtrIterator', requer somente uma única modificação já que sua definição de tipo de seu iterador agora deve estar associado a 'RandomPtrIterator':

```

typedef RandomPtrIterator
    <
        StringPtr,
        std::vector<std::string *>::iterator,
        std::string
    >
    iterator;

```

Incluindo a modificação do arquivo cabeçalho de 'StringPtr' no programa dado na seção 19.11.1 resultará num programa que se comporta identicamente à sua versão anterior, apesar de que 'StringPtr::begin()' e 'StringPtr::end()' agora retornam objetos iteradores construídos da definição de um modelo.

21.6: Um conversor para converter textos em qualquer coisa

A biblioteca estandarte da linguagem C oferece funções como 'atoi()', 'atol()' e outras que convertem cadeias de caracteres ASCII-Z em valores numéricos. Em C++ estas funções ainda estão disponíveis, mas uma forma de converter textos com maior segurança dos tipos é usar objetos da classe 'std::istringstream'.

Usando a classe 'std::istringstream' no lugar das funções estandartes de conversão da C se tem a vantagem da segurança de tipo, mas também parece uma alternativa incômoda. Depois de tudo, teremos que construir e iniciar um objeto 'std::istringstream' primeiro, antes de estarmos habilitados a extrair um valor de algum tipo dela. Isto requer usar uma variável. Então se o valor extraído só é requerido para iniciar algum parâmetro de uma função, se encontra que seria muito melhor um meio que evitasse todo esse trabalho.

Nesta seção desenvolveremos uma classe ('A2x') que evita toda desvantagem das funções da biblioteca estandarte da linguagem C, sem os requerimentos incômodos da definição dos objetos 'std::istream' uma e outra vez. A classe se chama 'A2x' que significa 'ascii para qualquer coisa'.

A classe 'A2x' é usada para obter um valor para qualquer tipo extraível de objetos 'std::istream' dada sua representação textual. Como 'A2x' representa uma variante dos objetos das funções C, não só tem segurança nos tipos senão que também são extensíveis. Conseqüentemente seu uso é muito preferível sobre os estandartes das funções C. Esi suas características:

- 'A2x' é derivada de 'std::istream', assim todos os membros da classe 'std::istream' estão disponíveis. Assim, extrações de valores sempre são feitas sem esforço:

```
class A2x: public std::istream
```

- 'A2x' possui um construtor padrão e um construtor que espera um argumento 'std::string'. O último construtor é usado para iniciar objetos 'A2x' com texto para ser convertido (p.ex., uma linha de texto obtida de um arquivo de configuração):

```
A2x()
{
}
A2x(std::string const &str)
:
    std::istream(str)
{
}
```

- O poder real de 'A2x' vem de seu operador de conversão 'Type()' que é o modelo de um membro. Como é o modelo de um membro, automaticamente se adapta ao tipo de variável à que se deve dar um valor, obtido pela conversão do texto armazenado no objeto 'A2x' para o tipo de variável desejado. Quando a extração falha, o membro herdado 'good()' por 'A2x', retorna falso:

```
template <typename Type>
operator Type()
{
    Type t;

    return (*this >> t) ? t : Type();
}
```

- Ocasionalmente o compilador pode não estar habilitado em determinar para que tipo converter. Neste caso um tipo explícito pode ser usado para o modelo:

```
A2x.operator int<int>();
// ou só:
A2x.operator int();
```

Como a sintaxe não se vê atrativa, também é oferecido o modelo de membro 'to()', que permite construções como:

```
A2x.to(int());
```

Eis sua implantação:

```
template <typename Type>
Type to(Type const&)
{
    return *this;
}
```

- Uma vez disponível um objeto 'A2x', pode ser re-iniciado usando o membro 'operator=()':

```
#include "a2x.h"

A2x &A2x::operator=(std::string const &str)
{
    clear(); // muito importante!!! se uma conversão falha, o objeto
            // fica inusável até que se execute este comando
    str(txt);
    return *this;
}
```

Eis alguns exemplos de seu uso:

```
int x = A2x("12");           // inicia int x de uma string "12"
A2x a2x("12.50");           // cria explicitamente um objeto A2x

double d;
d = a2x;                    // adjudica uma variável usando um objeto A2x

a2x = "err";
d = a2x;                    // d é 0: a conversão falhou,
                          // e a2x.good() == false

a2x = " a";                 // re-adjudica a a2x um novo texto
char c = a2x;               // c agora 'a': internamente operator>>() é usado
                          // espaços iniciais são saltados.

extern expectsInt(int x);    // inicia um parâmetro usando um
expectsInt(A2x("1200"));    // objeto anônimo A2x

d = A2x("12.45").to(int()); // d é 12, não 12,45
```

Além da classe 'A2x' uma classe complementar 'X2a' facilmente pode ser construída também. A construção de 'X2a' é deixada como exercício ao leitor.

21.7: Envoltórios para os algoritmos STL

Muitos algoritmos genéricos (cf. Capítulo 17) usam funções objetos para operar sobre dados

os quais são referidos por seus iteradores, ou requerem funções objetos predados que usam algum critério para decidir sobre esses dados. A solução standarte seguida pelos algoritmos genéricos é passar a informação a que iterador se refere em chamadas a funções sobrecarregadas operadoras (i.e., `'operator()()` das funções objetos que são passadas como argumentos aos algoritmos genéricos.

Usualmente esta solução requer a construção de uma classe dedicada que implanta a função objeto requerida. Contudo, em muitos casos a classe onde os iteradores já existem oferece a funcionalidade requerida. Alternativamente a funcionalidade pode existir como função membro dos objetos aos quais os iteradores se referem. Por exemplo, para encontrar o primeiro objeto `'string'` vazio num vetor de `'strings'` se poderia aproveitar o membro `'string::empty()'`.

Outra situação freqüentemente encontrada é relativa a um contexto local. Considere, outra vez, a situação onde os elementos de um vetor de `'strings'` são todos visitados: Cada objeto precisa ser inserido numa `'stream'` cuja referência só é conhecida pela função onde os elementos `'string'` são visitados, mas alguma informação adicional precisa ser passada à função de inserção também, usando o `'ostream_inserter'` menos apropriado.

A parte mais frustrante do uso de algoritmos genéricos é que as funções objeto dedicadas freqüentemente se parecem muito, mas a solução padrão (usar funções objetos predefinidas, usar iteradores especializados) raramente realizam a tarefa requerida: suas interfaces fixas de funções (p.ex., `'equal_to'` que chama `'operator==()'` dos objetos) a miúdo são muito rígidas para serem úteis e, mais, são incapazes de usar qualquer contexto local adicional ativo quando usadas.

Não obstante, podemos nos maravilhar que se possa construir modelos de classes que podem ser usadas uma e outra vez e criar funções objetos dedicadas. A instanciação de tais modelos de classes poderiam oferecer facilidades para chamar funções membro configuráveis, usando um contexto local configurável.

Nas seções vindouras diversos modelos de envoltórios que suportam estes requerimentos estão desenvolvidos. Para suportar um contexto local uma estrutura de contexto local é introduzida. Mais, o modelo de envoltório nos permitirá especificarmos a função membro que pode ser chamada em seu construtor. Assim a rigidez das funções membro fixas como as usadas nas funções objetos predefinidas é evitada.

Como um exemplo de algoritmo genérico que usualmente requer uma simples função objeto considere `'for_each()'`. O `'operator()()'` do objeto função passado a este algoritmo recebe como argumento uma referência ao objeto ao que o iterador se refere. Geralmente o `'operator()()'` fará uma ou duas coisas:

- Pode chamar uma função membro do objeto definido em sua lista de parâmetro (p.ex.,

'operator()(string &str)' pode chamar 'str.length());

- Pode chamar uma função, passando seu parâmetro como argumento(p.ex., chamar 'somefunction(str)).

Claro que o exemplo anterior está um pouco exagerado, já que o endereço de 'somefunction()' poderia ter sido passado diretamente ao algoritmo genérico, portanto para que usar um procedimento assim complexo? A resposta é contexto: Se 'somefunction()' requeresse outros argumentos, representando o contexto local onde 'somefunction()' for chamada, então o construtor dos objetos da função teriam recebido o contexto local como seu argumento, passando-o a 'somefunction()', junto com o objeto recebido pela função 'operator()' do objeto . Não há forma de passar qualquer contexto local a uma variante simples do algoritmo genérico, onde o endereço de uma função é passado à função genérica.

A primeira impressão, contudo, o fato que o contexto local difere de uma situação a outra faz difícil estandarizar o contexto local: um contexto local pode consistir de valores, ponteiros, referências, que diferem em número e tipos de uma situação para outra. É claramente impraticável definir para todas as situações possíveis e o uso de funções 'variadic' também não é muito atrativo, já que os argumentos passados ao construtor de objetos de uma função 'variadic' não podem ser simplesmente passados ao objeto da função 'operator()'.

O conceito de estrutura do contexto local é introduzido para estandarizar o contexto local. É baseado nas seguintes considerações:

- Usualmente uma função que requer um contexto local é uma função membro de uma classe.
- Em lugar de usar uma implantação intuitiva, onde é dado à função membro os parâmetros requeridos que representam um contexto local, ela recebe um só argumento: Um 'const &' para uma estrutura do contexto local.
- A estrutura do contexto local é definida na interface de classe.
- Antes da função ser chamada, uma estrutura do contexto local é iniciada, que então é passada como argumento à função.

Está claro que a organização das estruturas locais diferirão de uma situação para outra, mas sempre haverá só um contexto local requerido. O fato de que a organização entrante do contexto local difere de uma situação para outra não causa dificuldades ao mecanismo de modelagem da C++. Dispondo de um tipo genérico ('Context') junto com diversas instanciações concretas desse tipo genérico é uma mera matéria de como usar os moddos.

21.7.1: Estruturas do contexto local

Quando uma função é chamada, o contexto onde é chamada é, então, reconhecido pela função entregando-lhe a lista de parâmetros. Quando a função é chamada estes parâmetros são iniciados pelos argumentos da função. Por exemplo, uma função 'show()' pode esperar dois argumentos: Uma 'ostream &' onde está a informação e um objeto que será inserido na 'stream'. Por exemplo:

```
void State::show(ostream &out, Item const &item)
{
    out << "Here is item " << item.nr() << ":\n" <<
        item << endl;
}
```

As funções diferem claramente em suas listas de parâmetros: Ambos, o número e os tipos de seus parâmetros variam.

Uma estrutura do contexto local é usada para estandarizar a lista de parâmetros das funções, em benefício dos modelos de construção. No exemplo acima, a função 'State::show()' usa um contexto local de uma 'ostream &' e um 'Item const &'. Este contexto nunca muda e pode muito bem ser oferecido por uma estrutura definida como segue:

```
struct ShowContext
{
    ostream &out;
    Item const &item;
};
```

Note que esta estrutura imita a lista de parâmetros de 'State::show()'. Como está diretamente ligada à função 'State::show()' seria melhor definida na classe 'State'. Uma vez definida esta estrutura, a implantação de 'State::show()' é modificada, pois espera uma 'ShowContext &':

```
void State::show(ShowContext &context)
{
    context.out << "Here is item " << context.item.nr() << ":\n" <<
        context.item << endl;
}
```

(Alternativamente uma 'State::show(ShowContext &context)' sobrecarregada poderia ser definida, chamando o membro original 'show()').

Usando uma estrutura com o contexto local qualquer lista de parâmetros (exceto as das funções 'variadic') podem ser estandarizadas como uma lista de parâmetros de um só elemento. Agora que temos um único parâmetro para especificar qualquer contexto local, estamos prontos a modelar as

funções objetos das classes envoltórios.

21.7.2: Funções membro chamadas por funções objetos

A função membro chamada pela função objeto é a função `'operator()()`', que pode ser definida como uma função com muitos parâmetros. No contexto dos algoritmos genéricos, estes parâmetros são usualmente um ou dois elementos, representando os dados para os quais os iteradores dos algoritmos apontam. Desafortunadamente do ponto de vista do construtor do modelo de classe, não se sabe com anterioridade se estes elementos de dados são objetos, tipos primitivos ou ponteiros. Assumamos que gostaríamos de criar uma função objeto para mudar todas as letras em objetos `'strings'` para maiúsculas. Nesse caso nossa função `'operator()()`' recebe uma `'string &'` (p.ex., quando operando sobre um `'vector<striung>'`), mas nossa função `'operator()()`' também pode receber uma `'string *'` (p.ex., quando iterando sobre os elementos de um `'vector <string *>'`). Outros tipos de parâmetros também podem ser concebidos.

Portanto, como podemos definir uma função genérica que possa ser chamada de `'operator()()`', se não sabemos (ao definir o modelo) se usamos `'.*'` ou `'->*'`? O problema de se chamar uma função membro em combinação com um objeto ou um apontador a objeto não tem que ser resolvido pelo modelo. Em lugar disso pode ser manipulado pela própria classe, se a classe fornece um membro estático.

Uma vantagem adicional de se usar uma função estática é que um membro estático não possui atributos constantes. Conseqüentemente não pode cair em ambigüidades na chamada de uma função membro estática desde `'operator()()`' de uma função objeto.

Os algoritmos genéricos, contudo, diferem no uso na função objeto do valor retornado por `'operator()()`'. Como será ilustrado na seção seguinte, o tipo de retorno das funções chamadas também pode ser parametrizado.

21.7.3: Modelo de função objeto com um argumento configurável

Como um exemplo introdutório assumamos que temos uma classe `'Strings'` que armazena um membro de dados `'vector<string> d_vs'`. Queremos mudar todas as letras nas `'strings'` guardadas em `'d_vs'` para maiúsculas e inserir as `'strings'` originais e as modificadas num objeto `'ostream'` configurável. Para isso nossa classe oferece o membro `'uppercase(ostream &out)`.

Para realizar esta tarefa necessitamos usar o algoritmo genérico `'for_each()'` ou uma dada

função objeto. Está claro, como temos um contexto local (o objeto 'ostream' configurável), a função objeto é requerida aqui. Por isso, construímos a seguinte classe de suporte:

```
class Support
{
    std::ostream &d_out;

public:
    Support(std::ostream &out)
    :
        d_out(out)
    {}

    void operator()(std::string &str) const
    {
        d_out << str << " ";
        transform(str.begin(), str.end(), str.begin(), toupper);
        d_out << str << std::endl;
    }
};
```

Um objeto anônimo da classe 'Support' pode agora ser usado na implantação da classe 'Strings'. Eis um exemplo de sua definição e uso:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

#include "support.h"

class Strings
{
    std::vector<std::string> d_vs;

public:
    void uppercase(std::ostream &out)
    {
        for_each(d_vs.begin(), d_vs.end(), Support(out));
    }
};

using namespace std;

int main()
{
    Strings s;

    s.uppercase(cout);
}
```

Para modelar a classe 'Support', usando as considerações discutidas previamente, desenvolvemos as seguintes etapas:

- O contexto local poremos numa estrutura, que, então, é passada ao construtor do modelo, assim 'Context' se torna um dos modelos de tipo dos parâmetros.
- A implantação do 'operator()()' de modelo é estandarizada. No modelo chamará uma função, que recebe o argumento de 'operator()()' e uma referência ao contexto como seus argumentos. O endereço da função a chamar pode estar guardado numa variável local do modelo da função objeto. Na classe 'Support' 'operator()()' usa um tipo de retorno 'void'. Este tipo é a miúdo o tipo requerido, mas quando são definidos predicados pode ser um valor booleano. Por isso, o tipo de retorno de 'operator()()' do modelo (e, portanto, o tipo de retorno da função chamada) é feito configurável também, oferecendo por conveniência um tipo padrão 'void'. Assim, chegamos à seguinte definição da variável que armazena o endereço da função a chamar:

```
ReturnType (*d_fun) (Type &argument, Context &context);
```

e a implantação do 'operator()()' do modelo (que o passa a outro modelo de membro de dados: 'Context &d_context) fica:

```
ReturnType operator() (Type &param) const
{
    return (*d_fun) (param, d_context);
}
```

- Ao construtor do modelo são dados dois parâmetros: um endereço de função e uma referência a uma estrutura com o contexto local. Pondo o nome na classe de 'Wrap1' (de função objeto unária (1) envolvente), sua implantação fica:

```
Wrap1(ReturnType (*fun) (Type &, Context &), Context &context)
:
    d_fun(fun),
    d_context(context)
{ }
```

Agora estamos quase prontos a construir o modelo completo da classe 'Wrap1'. Duas situações adicionais merecem maiores considerações:

- Os argumentos passados a 'operator()()' do modelo do membro podem ser de várias naturezas: valores, referências modificáveis, referências imutáveis ('const'), apontadores a entidades modificáveis ou apontadores a entidades imutáveis. O modelo deve oferecer facilidades de uso a todos esses tipos diferentes de argumentos.
- Os algoritmos definidos na biblioteca estandarte de modelos, notavelmente aqueles que requerem funções objetos com predicados (p.ex., 'find_if()'), assumem que estes objetos definem

tipos internos, nomeadamente tipos de resultados para seu membro 'operator()()' e tipos de argumentos para seus tipos de dados (funções objetos com predicado binário (veja seção 20.7.4)) tipo do primeiro argumento e tipo do segundo argumento para os tipos respectivos dos argumentos de 'operator()()' são esperados. E mais, estes tipos têm que ter nomes plenos, sem ponteiros nem referências.

Vários tipos de parâmetros do 'operator()()' do modelo podem ser manipulados por versões sobrecarregadas do modelo de construtor e de seu membro 'operator()()', definindo quatro implantações que manipulam referências 'Type const' e apontadores a 'Type const'. Para cada uma destas situações um ponteiro a função à função correspondente, chamada pelo 'operator()()' do modelo precisa ser definido também. Como em cada instanciação do modelo só um tipo das funções sobrecarregadas (construtor e o 'operator()()' associado) será usado, uma união pode ser definida que acomode os ponteiros dos vários tipos (i.e., quatro) de funções que podem ser passadas ao construtor do modelo. Esta união pode ser anônima, já que somente seus campos serão usados. Note que os valores dos argumentos podem ser manipulados pelos parâmetros 'Type const &': Não se requer versões sobrecarregadas para manipular tipos dos valores dos argumentos.

O tipo interno esperado por algumas funções 'STL' pode ser disponibilizado definindo 'typedefs'. Como os vários tipos de argumentos (constantes, ponteiros, referências) são manipulados pelos construtores sobrecarregados de modelos e funções membro, os 'typedefs' podem colocar simples apelidos nos modelos de tipos de parâmetros.

Eis a implantação do modelo da função objeto de um argumento configurável:

```
template <typename Type, typename Context, typename ReturnType = void>
class Wrap1
{
    Context &d_context;
    union
    {
        ReturnType (*d_ref) (Type &, Context &);
        ReturnType (*d_constref) (Type const &, Context &);
        ReturnType (*d_ptr) (Type *, Context &);
        ReturnType (*d_constptr) (Type const *, Context &);
    };
public:
    typedef Type          argument_type;
    typedef ReturnType    result_type;
                                // reference
    Wrap1(ReturnType (*fun) (Type &, Context &), Context &context)
    :
        d_context(context),
        d_ref(fun)
    {}
    ReturnType operator() (Type &param) const
```

```

{
    return (*d_ref)(param, d_context);
}

// const reference
Wrap1(ReturnType (*fun)(Type const &, Context &),
      Context &context)
:
    d_context(context),
    d_constref(fun)
{}
ReturnType operator()(Type const &param) const
{
    return (*d_constref)(param, d_context);
}

// pointer
Wrap1(ReturnType (*fun)(Type *, Context &), Context &context)
:
    d_context(context),
    d_ptr(fun)
{}
ReturnType operator()(Type *param) const
{
    return (*d_ptr)(param, d_context);
}

// const pointer
Wrap1(ReturnType (*fun)(Type const *, Context &),
      Context &context)
:
    d_context(context),
    d_constptr(fun)
{}
ReturnType operator()(Type const *param) const
{
    return (*d_constptr)(param, d_context);
}
};

```

Para usar este modelo, a implantação original dedicada de 'Support::operator()()' é agora definida numa função membro estática da classe 'Strigs', que define também a estrutura do contexto local. Aqui está a nova implantação da classe 'Strings', usando o modelo 'Wrap1':

```

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

#include "wrap1.h"

class Strings
{
    std::vector<std::string> d_vs;

```



```

struct Context
{
    std::ostream &out;
};

public:
    void uppercase(std::ostream &out)
    {
        Context context = {out};
        for_each(d_vs.begin(), d_vs.end(),
            Wrap1<std::string, Context>(&xform, context));
    }

private:
    static void xform(std::string &str, Context &context)
    {
        context.out << str << " ";
        transform(str.begin(), str.end(), str.begin(), toupper);
        context.out << str << std::endl;
    }
};

using namespace std;

int main()
{
    Strings s;

    s.uppercase(cout);
}

```

Para ilustrar o uso do modelo de parâmetro 'ReturnType', assumamos que a transformação só é requerida na primeira 'string' vazia. Neste caso o algoritmo genérico 'find_if' está à mão, já que para uma vez que um predicado retorne verdadeiro. A função 'xform()' retorna um valor booleano e a implantação de 'uppercase()' especifica um tipo explícito ('bool') para o modelo do parâmetro 'ReturnType':

```

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

#include "wrap1.h"

class Strings
{
    std::vector<std::string> d_vs;

    struct Context
    {

```

```

        std::ostream &out;
    };

public:
    void uppercase(std::ostream &out)
    {
        Context context = {out};
        for_each(d_vs.begin(), d_vs.end(),
            Wrap1<std::string, Context, bool>(&xform, context));
    }

private:
    static bool xform(std::string &str, Context &context)
    {
        context.out << str << " ";
        transform(str.begin(), str.end(), str.begin(), toupper);
        context.out << str << std::endl;

        return str.empty();
    }
};

using namespace std;

int main()
{
    Strings s;

    s.uppercase(cout);
}

```

Note que só a classe 'Strings' necessitou ser modificada. O modelo de 'Wrap1' foi usado para criar os dois, o retorno 'void' da função e o predicado unário.

Uma nota final: Algumas vezes não é necessário um contexto, mas a solução feita com o modelo de classe 'Wrap1' pode ser considerada útil. Nesses casos, ou um contexto vazio pode ser definido ou uma classe envolvente alternativa que não use um contexto pode ser definida. Pessoalmente, uso a primeira opção.

21.7.4: Modelo de função objeto com dois argumentos configurável

Tendo construído o envoltório unário, a construção do modelo de envoltório binário não oferece surpresas. O 'operator()()' da função objeto agora é chamado com dois argumentos no lugar de um. A construção de uma classe 'Wrap2' é quase idêntica à da 'Wrap1'. Ei-la aqui:

```

template <typename Type1, typename Type2,
    typename Context, typename ReturnType = void>

```

```

class Wrap2
{
    union
    {
        ReturnType (*d_ref)(Type1 &, Type2 &);
        ReturnType (*d_constref)(Type1 const &, Type2 const &);
        ReturnType (*d_ptrs)(Type1 *, Type2 *);
        ReturnType (*d_constptrs)(Type1 const *, Type2 const *);
    };
    Context &d_context;

public:
    typedef Type1      first_argument_type;
    typedef Type2      second_argument_type;
    typedef ReturnType result_type;

                                // references
    Wrap2(ReturnType (*fun)(Type1 &, Type2 &), Context &),
        Context &context)
    :
        d_ref(fun),
        d_context(context)
    {}
    ReturnType operator()(Type1 &param1, Type2 &param2) const
    {
        return (*d_ref)(param1, param2, d_context);
    }

                                // const references
    Wrap2(ReturnType (*fun)(Type1 const &, Type2 const &), Context &),
        Context &context)
    :
        d_constref(fun),
        d_context(context)
    {}
    ReturnType operator()(Type1 const &param1,
                          Type2 const &param2) const
    {
        return (*d_constref)(param1, param2, d_context);
    }

                                // pointers
    Wrap2(ReturnType (*fun)(Type1 *, Type2 *), Context &),
        Context &context)
    :
        d_ptrs(fun),
        d_context(context)
    {}
    ReturnType operator()(Type1 *param1, Type2 *param2) const
    {
        return (*d_ptrs)(param1, param2, d_context);
    }
}

```

```

    }

    // ponteiros constantes
    Wrap2(ReturnType (*fun) (Type1 const *, Type2 const *), Context &),
        Context &context)
    :
        d_constptrs(fun),
        d_context(context)
    {}
    ReturnType operator() (Type1 const *param1,
                          Type2 const *param2) const
    {
        return (*d_constptrs)(param1, param2, d_context);
    }
};

```

Como com o modelo do envoltório unário (veja seção 20.7.3), um classe adicional, que não espere o contexto local, pode ser definida.

21.8: Usando *'bison++'* e *'flex'*

O exemplo discutido nesta seção entra em peculiaridades no uso de geradores parser e geradores scanner, que geram fontes C++. A entrada de de um programa que exceda certo nível de complexidade é vantajoso o uso de geradores parser e scanner para criar o código que faça o reconhecimento de entrada.

O exemplo que aqui se encontra assume que o leitor sabe como usar um gerador de scanner 'flex' e um gerador de parser 'bison'. Ambos, bison e flex estão bem documentados em outra parte. Os predecessores de bison e flex, chamados yacc e lex, respectivamente, estão descritos em diversos livros, p.ex., no livro de O'Reilly *'lex & yacc'*.

Contudo, geradores de parser e scanner também estão (talvez mais comumente nos dias atuais) disponíveis como software livre. Ambos bison e flex são parte de distribuições de software ou podem ser obtidos em: <ftp://prep.ai.mit.edu/pub/gnu>

O flex cria uma classe C++ quando a opção `%option c++` for especificada.

Para geradores de parser o programa bison está disponível. No início dos anos 90, Alain Coetmeur (coetmeur@icdc.fr), criou uma variant C++ (bison++) que cria uma classe parser. Apesar de que o programa bison++ produz código que pode ser usado em programas C++ também mostra características mais apropriadas ao contexto de linguagem C que ao contexto C++. Em Janeiro de 2005 eu reescrevi partes do bison++ de Alain, resultando na versão original do programa bison++. Então em

Maio de 2005 se terminou uma completa reescritura do gerador de parser bison++, que está disponível na Internet com versão 0.98 e superior. O Bisonc++ pode ser baixado de <http://www.sourceforge.net> bem como de [ulr\(ftp.rug.nl\)](http://ftp.rug.nl) (<ftp://ftp.rug.nl/contrib/frank/software/linux/bisonc++>), onde como arquivo fonte e pacote binário (i386) Debian (incluindo a documentação do bison++). O bisonc++ cria uma classe parser mais limpa que bison++. Em particular, deriva a classe parser de uma classe de base que contém definições de fichas e tipos bem como funções membro que não podem ser (re)definidas pelo programador. A maioria destes membros deve ser definida também diretamente na classe parser. Dada esta solução, a classe parser resultante é bem pequena, declara somente membros que são definidos pelo programador (bem como outros membros, gerados por bisonc++, implantando o membro de parser 'parse()'). O membro 'parse()' é o único membro público de bisonc++ gerado pela classe parser. Os membros restantes são privados. O único membro que não está implantado (e que portanto deve ser implantado pelo programador) é 'lex()', que produz a próxima ficha (token) léxica.

Nesta seção das Anotações focalizaremos bisonc++ como nosso gerador de parser.

Usando-se flex e bisonc++ se pode gerar classes de baseadas em parsers e scanners. A vantagem desta solução é que a interface com scanners e parsers tende a se tornar mais clara que sem usar a interface de classe. Ainda mais, as classes permitem livrar-nos da maioria se não de todas as variáveis globais, facilitando o uso de múltiplos scanners e parsers num programa.

Abaixo foram elaborados dois exemplos. O primeiro exemplo usa só o flex. O scanner que gera monitora a produção de um arquivo de diversas partes. Este exemplo se focaliza no scanner léxico e chaveamento de arquivos enquanto se move através da informação. O segundo exemplo usa ambos, flex e bisonc++ para gerar um scanner e um parser, que transforma as expressões estandartes da aritmética em notação pós-fixa, comumente usada na geração de código em compiladores e nas calculadoras HP. No segundo exemplo a ênfase é sobretudo no bisonc++ e na composição de um scanner dentro de um parser gerado.

21.8.1: Uso do 'flex' para criar um 'scanner'

O scanner léxico desenvolvido nesta seção é usado para monitorar a produção de um arquivo de muitos sub-arquivos. O início é o seguinte: a linguagem de entrada conhece uma diretiva '#include', seguida pelo texto que especifica o arquivo (seu caminho) que deve ser incluído no local da diretiva '#include'.

Para evitar complexidades irrelevantes, no exemplo a seguir, o formato da diretiva '#include' está restrito à forma '#include <caminho do arquivo>'. O arquivo especificado entre parênteses angulares deve estar disponível no lugar indicado por caminho do arquivo. Se o arquivo não se encontrar ali, o

programa termina com uma mensagem de erro.

O programa começa com um ou dois nomes de arquivos como argumentos. Se o programa começar com um só nome de arquivo, a saída é escrita na 'stream' 'cout', a saída estandarte, cujo nome é dado como o segundo argumento do programa.

O programa define um máximo de profundidade aninhada. Uma vez excedido esse máximo, o programa termina com uma mensagem de erro. No caso, o nome do arquivo na pilha que indica onde o arquivo estava incluído é impresso.

Uma característica adicional é que as linhas de comentário (estandarte C++) são ignoradas. Portanto, as diretivas de inclusão em linhas de comentário também são ignoradas.

O programa é criado através dos seguintes passos:

- Primeiro o arquivo 'lexer' é construído, contendo as especificações da linguagem de entrada;
- A partir das especificações em 'lexer' os requerimentos para a classe 'Scanner' se desenvolve. A classe 'Scanner' é uma classe envolvente da classe 'yyFlexLexer' gerada por flex. Os requerimentos resultam na especificação da interface da classe 'Scanner';
- Em seguida é construída 'main()'. Um objeto 'Startup' é criado que inspeciona os argumentos da linha de comando. Se for bem sucedido, o membro de 'Scanner' 'yylex()' é chamado para construir o arquivo de saída;
- Agora que o contexto global do programa foi especificado, as funções membro das várias classes são construídos;
- Finalmente o programa é compilado e linkado.

21.8.1.1: A classe derivada 'Scanner'

O código associado às regras das expressões regulares está localizado na classe 'yyFlexLexer'. Mas queremos, claro está, usar os membros das classes derivadas neste código. Isto causa um pequeno problema: Como uma classe de base pode conhecer os membros de classes derivadas dela?

Afortunadamente a herança nos ajuda a realizar isto. Na especificação de classe 'yyFlexLexer', nos advertimos que a função 'yyflex()' é uma função virtual. O arquivo cabeçalho 'FlexLexer.h' declara o membro virtual 'yyflex()':

```

class yyFlexLexer: public FlexLexer
{
    public:
        yyFlexLexer( istream* arg_yyin = 0, ostream* arg_yyout = 0 );

        virtual ~yyFlexLexer();

        void yy_switch_to_buffer( struct yy_buffer_state* new_buffer );
        struct yy_buffer_state* yy_create_buffer( istream* s, int size );
        void yy_delete_buffer( struct yy_buffer_state* b );
        void yyrestart( istream* s );

        virtual int yylex();

        virtual void switch_streams( istream* new_in, ostream* new_out );
};

```

Como a função é virtual, pode ser suprimida numa classe derivada. Nesse caso, a função suprimida será chamada no código de sua classe de base (i.e., 'yyFlexLexer'). Como a classe derivada 'yylex()' é chamada, agora terá acesso aos membros da classe derivada e também aos membros públicos e protegidos de sua classe de base.

Como padrão, o contexto onde o scanner é gerado está posta a função 'yyFlexLexer::yylex()'. Este contexto muda se usamos uma classe derivada, p.ex., 'Scanner'. Para derivar 'Scanner' de 'yyFlexLexer', gerada por flex, faça o seguinte:

- A função 'yylex()' precisa ser declarada na classe derivada 'Scanner';
- As opções (veja abaixo) são usadas para informar sobre o nome da classe.

Observando-se as expressões regulares, notamos que necessitamos regras para reconhecer comentários, diretivas '#include' e todos os caracteres restantes. Esta é quase uma prática estandarte. Quando uma diretiva '#include' é detetada ela é examinada pelo scanner. Esta também é uma prática comum. Eis o que nosso scanner léxico fará:

- Como é usual, o pré-processamento das diretivas não é analisado pelo parser, ma pelo scanner léxico;
- O scanner usa um mini scanner para extrair o nome do arquivo da diretiva, lançando um valor de 'Scanner::Error' ('invalidInclude') se falha;
- Se o nome do arquivo pode ser extraído, é guardado em 'nextSource';

- Quando a diretiva de inclusão tenha sido processada, 'pushSource()' é chamada para fazer o chaveamento para outro arquivo;
- Quando chega ao fim do arquivo ('EOF'), a função membro da classe derivada 'popSource()' é chamada, voltando o arquivo anteriormente puxado e retornando verdadeiro;
- Uma vez que a pilha de arquivos esteja vazia, 'popSource()' retorna falso, resultando na chamada a 'yyterminate()', terminando o scanner.

O arquivo de especificação do scanner léxico está organizado similarmente ao usado pelo flex em contextos C. Apesar de que, para contextos C++, o flex cria uma classe ('yyFlexLexer') do qual outra classe (p.ex., 'Scanner') pode derivar. A especificação do próprio arquivo flex possui três seções:

- A primeira seção do arquivo 'lexer' é um preâmbulo C++, contendo código que pode ser usado na definição das ações a realizar uma vez que uma expressão regular combine. Na iniciação presente, onde cada classe tem seu próprio arquivo cabeçalho interno, o arquivo cabeçalho interno inclui o arquivo 'scanner.h', que em seu turno inclui 'FlexLexer.h', que é parte da distribuição do flex. Contudo, devido à complexa iniciação do arquivo anterior, pode não ser lido novamente pelo código gerado por flex. Assim, temos a seguinte situação:
- Primeiro examinamos o arquivo de especificação 'lexer'. Contém um preâmbulo que inclui 'scanner.h', como este declara, via 'scanner.h' a classe 'Scanner', portanto estamos habilitados a chamar os membros de 'Scanner' do código associado às expressões regulares definidas no arquivo de especificação 'lexer';
- No 'scanner.h, que define a classe 'Scanner', o arquivo cabeçalho 'FlexLexer.h', que declara a 'Scanner' como classe de base, devem ser lidos pelo compilador antes da definição da própria classe 'Scanner';
- O código gerado por flex já inclui 'FlexLexer.h' e, como mencionado, não pode ser incluído outra vez. Contudo, flex inserirá o preâmbulo do arquivo de especificação no código que gera;
- Como este preâmbulo inclui 'scanner.h' e, portanto, 'scanner.h' e 'FlexLexer.h', se incluirmos 'FlexLexer.h' serão duas vezes e isto deve ser evitado.

Para evitar incluir 'FlexLexer.h' diversas vezes, sugerimos o seguinte:

- Apesar de que 'scanner.h' inclui 'scanner.h', 'scanner.h' é modificado de tal modo que inclui 'FlexLexer.h', a menos que a variável do pré-processador `_SKIP_FLEXLEXER_` esteja definida;

- No arquivo de especificação do flex, `_SKIP_FLEXLEXER_` é definida justo antes de incluir 'scanner.ih'.

Usando este esquema, o código gerado por flex agora volta a incluir 'FlexLexer.h'. Ao mesmo tempo, ao compilar os membros de 'Scanner', procede independentemente das especificações do arquivo de preâmbulo 'lexer', assim aqui 'FlexLexer.h' é incluído com propriedade. Eis o preâmbulo de especificação dos arquivos:

```
%{
    #define _SKIP_YFLEXLEXER_
    #include "scanner.ih"
}%
```

- A segunda seção do arquivo de especificação é uma área de símbolos do flex usada para definir símbolos, como um mini scanner ou opções. As seguintes opções são sugeridas:
 - - '%option 8bit': permite ao scanner léxico ler caracteres de 8 bits (antes que 7 bits, que é o padrão);
 - - '%option c++': O flex gerará código C++;
 - - '%option debug': Inclui o código de depuração do flex. Ao chamar a função membro 'set_debug(true)' ativará o código de depuração em tempo de execução.

Quando ativo, informações a respeito quais regras são respeitadas são escritas na 'stream' de saída de erro. Para suprimir a execução do código de depuração chamamos 'set_debug(false)';

- - '%option noyywrap': Quando o 'scanner' chega ao fim de um arquivo, como padrão, chama a função 'yywrap()' que realiza o chaveamento para processar outro arquivo. Como existem alternativas que fazem esta função supérflua (veja abaixo), sugerimos especificar esta opção também;
- - '%option outfile="yylex.cc"': Define o nome do arquivo gerado como fonte C++, 'yylex.cc';
- - '%option warn': Esta opção é fortemente recomendada pela documentação do flex, portanto é mencionada aqui também. Veja a documentação do flex para detalhes;
- - '%option yyclass="Scanner"': define 'Scanner' como o nome da classe derivada de 'yyFlexLexer';

- - '%option yylineno': Causa que o 'scanner' léxico controle o número das linhas dos arquivos que lê. Quando processar arquivos aninhados, a variável 'yylineno' não é automaticamente posta no número da última linha de um arquivo, ao retornar a um arquivo parcialmente processado. Nesses casos, 'yylineno' deve ser posta no valor antigo. Se especificada, o número de linha corrente é retornado pelo membro público 'lineno()' como um inteiro.

Eis a área de especificação de símbolos do arquivo:

```
%option yyclass="Scanner" outfile="yylex.cc" c++ 8bit warn noyywrap yylineno
%option debug
```

```
%x      comment
%x      include
```

```
eolnComment    "//".*
anyChar        .|\n
```

- A terceira seção do arquivo de especificação é a seção das regras, onde são definidas as expressões regulares e suas ações. No exemplo desenvolvido aqui, o 'lexer' deve copiar informações da 'istream' '*yyin' para a 'ostream' '*yyout'. Para isto, a macro pré-definida 'ECHO' é usada. eis a especificação da seção de regras:

```
%%
/*
    The comment-rules: comment lines are ignored.
*/
{eolnComment}
"/*"          BEGIN comment;
<comment>{anyChar}
<comment>"*/"  BEGIN INITIAL;

/*
    File switching: #include <filepath>
*/
#include[ \t]+"<"      BEGIN include;
<include>[^ \t>]+      d_nextSource = yytext;
<include>">"[ \t]*\n    {
    BEGIN INITIAL;
    pushSource(YY_CURRENT_BUFFER, YY_BUF_SIZE);
    }
<include>{anyChar}     throw invalidInclude;

/*
    The default rules: eating all the rest, echoing it to output
*/
{anyChar}              ECHO;

/*
```

```

    The <<EOF>> rule: pop a pushed file, or terminate the lexer
    */
<<EOF>>          {
                    if (!popSource (YY_CURRENT_BUFFER) )
                        yyterminate();
                    }
%%

```

Como os membros da classe derivada podem acessar a informação armazenada no 'scanner' léxico (pode, mesmo, acessar a informação diretamente, já que os membros de dados 'yyFlexLexer' são protegidos e, portanto, acessíveis às classes derivadas), a maioria do processamento pode ser deixado às funções da classe derivada. Isto resulta numa definição muito clara das especificações do arquivo, que não requer um código pesado no preâmbulo.

21.8.1.2: Implantando a classe 'Scanner'

A classe 'Scanner' é derivada da classe 'yyFlexLexer', gerada pelo 'flex'. A classe derivada tem acesso aos dados controlados pelo 'scanner' léxico. Em particular tem acesso aos seguintes membros de dados:

- - 'char *yytext': Que contém o texto que combina com uma expressão regular. Os clientes podem aceder esta informação usando o membro 'YYText()' do 'scanner';
- - 'int yyleng': É o tamanho do texto, em bytes, em 'yytext'. Os clientes podem aceder esta informação usando o membro 'YYLeng()' do 'scanner';
- - 'int yylineno': Número atual da linha. Esta variável só é mantida se '%option yylineno' for especificada. Os clientes têm acesso a este valor usando o membro 'lineno()' do 'scanner'.

Outros membros estão disponíveis também, mas são menos usados. Os detalhes podem ser encontrados em 'FlexLexer.h'.

Os objetos da classe 'Scanner' realizam duas tarefas:

- Colocam informações sobre o arquivo atual na pilha de arquivos;
- Retiram a última informação colocada na pilha quando é detetado 'EOF' de um arquivo.

Diversas funções membro são usadas para realizar estas tarefas. Como são auxiliares do 'scanner', são membros privados. Na prática, estes membros são desenvolvidos quando aparecem suas necessidades. Note que aparte das funções membro privadas, diversos membros de dados privados

também são definidos. Vejamos mais de perto a implantação da classe 'Scanner':

- Primeiro, vejamos a seção inicial da classe, que mostra a inclusão condicional de 'FlexLexer.h', a abertura da classe e seus dados privados. Sua seção pública começa definindo a enumeração 'Error' que define várias constantes simbólicas para erros que podem ser detetados:

```
#if ! defined(_SKIP_YYFLEXLEXER_)
#include <FlexLexer.h>
#endif

class Scanner: public yyFlexLexer
{
    std::stack<yy_buffer_state *>    d_state;
    std::vector<std::string>         d_fileName;
    std::string                     d_nextSource;

    static size_t const             s_maxDepth = 10;

public:
    enum Error
    {
        invalidInclude,
        circularInclusion,
        nestingTooDeep,
        cantRead,
    };
};
```

- Como são objetos, os membros de dados da classe são iniciados automaticamente pelo construtor de 'Scanner'. Ativa o arquivo de entrada inicial (e saída) e coloca o nome do arquivo inicial de entrada na pilha. Eis sua implantação:

```
#include "scanner.ih"

Scanner::Scanner(istream *yyin, string const &initialName)
{
    switch_streams(yyin, yyout);
    d_fileName.push_back(initialName);
}
```

- O processo de escaneamento procede como segue:
 - Uma vez que o 'scanner' extrai o nome do arquivo da diretiva '#include', um chaveamento para outro arquivo é realizado por 'pushSource()'. Se o nome não pode ser extraído, o 'scanner' lança uma exceção 'invalidInclude'. O membro 'pushSource()' e sua função apareada 'popSource()' manipulam o chaveamento do arquivo. O chaveamento do arquivo procede assim:
 - Primeiro a profundidade das inclusões aninhadas é analisada. Se 'maxDepth' foi

alcançada, a pilha é considerada cheia e o 'scanner' lança a exceção 'nestingTooDeep';

- Em seguida 'throwOnCircularInclusion()' é chamada para evitar inclusões circulares quando se chaveia para um novo arquivo. Esta função lança uma exceção se o nome do arquivo for incluído duas vezes, usando um exame literal simples. Eis sua implantação:

```
#include "scanner.ih"

void Scanner::throwOnCircularInclusion()
{
    vector<string>::iterator
        it = find(d_fileName.begin(), d_fileName.end(), d_nextSource);

    if (it != d_fileName.end())
        throw circularInclusion;
}
```

- Então um novo objeto 'ifstream' é criado para o nome de arquivo em 'nextSource'. Se falha, o 'scanner' lança uma exceção 'cantRead';
- Finalmente, é criado um novo 'yy_buffer_state' para a 'stream' recém aberta e o 'scanner' léxico é instruído a chavear para essa 'stream' usando a função membro 'yy_switch_to_buffer()' da classe 'yyFlexLexer'.

Eis a implantação de 'pushSource()':

```
#include "scanner.ih"

void Scanner::pushSource(yy_buffer_state *current, size_t size)
{
    if (d_state.size() == s_maxDepth)
        throw nestingTooDeep;

    throwOnCircularInclusion();
    d_fileName.push_back(d_nextSource);

    ifstream *newStream = new ifstream(d_nextSource.c_str());

    if (!*newStream)
        throw cantRead;

    d_state.push(current);
    yy_switch_to_buffer(yy_create_buffer(newStream, size));
}
```

- A classe 'yyFlexLexer' fornece uma série de funções membro que podem ser usadas no chaveamento dos arquivos. A capacidade de chavear arquivos de um objeto 'yyFlexLexer' está na estrutura 'yy_buffer_state', que contém o estado do 'scan_buffer' do arquivo atualmente sendo

lido. Este bufer é posto na pilha 'd_state' quando é encontrada uma '#include'. O conteúdo de 'yy_buffer_state' é substituído pelo bufer criado para o processamento do próximo arquivo. Note que no arquivo de especificações do 'flex' a função 'pushSource()' é chamada como:

```
pushSource (YY_CURRENT_BUFFER, YY_BUF_SIZE);
```

'YY_CURRENT_BUFFER' e 'YY_BUF_SIZE' são macros disponíveis só na seção de regras do arquivo de especificações 'lexer', assim são passadas a 'pushSource()'. Atualmente não é possível usar essas macros diretamente com as funções da classe 'Scanner';

- Note que 'yylineno' não é atualizada quando o chaveamento de um arquivo é realizado. Se o número da linha deve ser monitorado, então o valor corrente de 'yylineno' deve ser posto em zero por 'pushSource()', visto que 'popSource()' renova um valor anterior de 'yylineno', retirado da pilha. A implantação atual de 'Scanner' mantém uma pilha só de ponteiros 'yy_buffer_state'. Mudando para uma pilha de elementos 'pair<yy_buffer_state *, size_t>' permitiria-nos salvar (e recuperar) os números de linha também. Esta modificação é deixada como exercício ao leitor;
- A função membro 'popSource()' é chamada para recuperar o bufer previamente guardado na pilha, permitindo ao 'scanner' continuar sua tarefa justo depois da diretiva '#include'. O membro 'popSource()' primeiro inspeciona o tamanho da pilha 'd_state': Se não está vazia, então o bufer atual é eliminado, para ser substituído pelo estado que espera no topo da pilha. O chaveamento do arquivo é feito pelos membros de 'yyFlexLexer' 'yy_delete_buffer()' e 'yy_switch_to_buffer()'. Note que 'yy_delete_buffer()' cuida de fechar a 'ifstream' e eliminar a memória alocada para esta 'stream' em 'pushSource()'. Ainda mais, o nome do arquivo entrado por último no vetor de nomes de arquivos é removido. Feito tudo isto a função retorna verdadeiro:

```
#include "scanner.ih"
```

```
bool Scanner::popSource(yy_buffer_state *current)
{
    if (d_state.empty())
        return false;

    yy_delete_buffer(current);
    yy_switch_to_buffer(d_state.top());
    d_state.pop();
    d_fileName.pop_back();

    return true;
}
```

- Dois membros de serviço são oferecidos: 'stackTrace()' que descarrega os nomes dos arquivos na pilha para a 'stream' de erro. Pode ser chamada pelo lançamento de uma exceção. Eis sua implantação:

```
#include "scanner.ih"
```

```
void Scanner::stackTrace()
{
    for (size_t idx = 0; idx < d_fileName.size() - 1; ++idx)
        cerr << idx << ": " << d_fileName[idx] << " included " <<
            d_fileName[idx + 1] << endl;
}
```

- 'lastfile()' retorna o nome do arquivo atualmente em processamento. Pode ser implantada em linha:

```
std::string const &lastFile()
{
    return d_fileName.back();
}
```

- O próprio 'scanner' léxico é definido em 'Scanner::yylex()'. Por isso 'yylex()' deve ser declarada pela classe 'Scanner', já que ela suprime o membro virtual 'yylex()' de 'FlexLexer'.

21.8.1.3: Usando um objeto 'Scanner'

O programa que usa nossa 'Scanner' é bem simples. Espera um nome de arquivo, indicando onde começar o processo de escaneamento. Inicialmente o número de argumentos é examinado. Se pelo menos um argumento é dado, então um objeto 'ifstream' é criado. Se este objeto pode ser criado, então um objeto 'Scanner' é construído, recebendo o endereço do objeto 'ifstream' e o nome do arquivo inicial como seus argumentos. Então o objeto membro de 'Scanner' 'yyflex()' é chamado. O objeto 'scanner' lança a exceção 'Scanner::Error' se falha em realizar suas tarefas com propriedade. Essas exceções são apanhadas no fim de 'main()'. Eis a fonte do programa:

```
#include "lexer.h"
using namespace std;

int main(int argc, char **argv)
{
    if (argc == 1)
    {
        cerr << "Filename argument required\n";
        exit (1);
    }
    ifstream yyin(argv[1]);
    if (!yyin)
    {
        cerr << "Can't read " << argv[1] << endl;
        exit(1);
    }
    Scanner scanner(&yyin, argv[1]);
```

```

try
{
    return scanner.yylex();
}
catch (Scanner::Error err)
{
    char const *msg[] =
    {
        "Include specification",
        "Circular Include",
        "Nesting",
        "Read",
    };
    cerr << msg[err] << " error in " << scanner.lastFile() <<
        ", line " << scanner.lineno() << endl;
    scanner.stackTrace();
    return 1;
}
return 0;
}

```

21.8.1.4: Construindo o programa

O programa final é construído em dois passos. Esses passos são para um sistema Unix, onde 'flex' e o compilador Gnu C++, g++, estão instalados:

- Primeiro, o 'scanner' léxico é criado usando o 'flex'. Para isto o seguinte comando deve ser dado:

```
flex lexer
```

- Em seguida todas as fontes são compiladas e linkadas. Em situações onde a função padrão 'yywrap()' é usada, a biblioteca 'libfl.a' deve ser linkada com o programa final. Normalmente isto não é requerido e o programa pode ser construído como p.ex.:

```
g++ -o lexer *.cc
```

Com o propósito de depuração do 'scanner' léxico, as regras de combinação e as fichas fornecidas são informações úteis. Quando se especifica as '%option', o código de depuração será incluído no 'scanner' gerado. Para obter as informações de depuração, este código precisa ser ativado. Assumindo que o objeto do 'scanner' é chamado 'scanner', o comando:

```
scanner.set_debug(true);
```

Produzirá a informação de depuração na 'stream' de erro padrão.

21.8.2: Usando ambos 'bisonc++' e 'flex'

Quando uma linguagem de entrada excede certo nível de complexidade, um parser freqüentemente é usado para controlar a complexidade da linguagem. Neste caso, um gerador de parser pode ser usado para gerar o código que verifica a correta aplicação da gramática de entrada. O 'scanner' léxico (de preferência composto dentro do parser) fornece pedaços da entrada, chamados fichas ('tokens'). O parser, então, processa as séries de fichas geradas pelo 'scanner' léxico.

O ponto de início de desenvolvimento de programas que usam parsers e 'scanners' é a gramática. A gramática define um conjunto de fichas que podem ser retornadas pelo 'scanner' léxico (comumente chamado léxico ('lexer')).

Finalmente, é fornecido um código auxiliar para 'preencher os espaços em branco': As ações realizadas pelo parser e pelo léxico não são especificadas, em geral, literalmente nas regras gramaticais ou expressões regulares léxicas, mas podem ser implantadas em funções membros, chamadas das regras do parser ou associadas às expressões regulares léxicas.

Na seção anterior vimos um exemplo de uma classe C++ gerada por 'flex'. Nesta seção nos concentraremos no parser. O parser pode ser gerado da especificação de uma gramática, processada pelo programa 'bisonc++'. A especificação da gramática requerida pelo 'bisonc++' é semelhante às especificações requeridas pelo 'bison' (e um programa existente 'bison++', escrito no início dos anos noventa pelo francês Alain Coetmeur), mas 'bisonc++' gera uma C++ mais próxima dos estandartes atuais que 'bison++', que se mostra ainda características da linguagem C.

Nesta seção é desenvolvido um programa que converte expressões infixas, onde os operadores binários são escritos entre seus operandos, em expressões pósfixas, nas quais os operadores binários são escritos depois de seus operandos. Mais ainda, os operadores unários, serão convertidos de sua notação préfixa a uma forma posfixa. O operador unário '+' é ignorado, já que não requer outras ações. Em essência nosso pequeno calculador é um micro compilador, transformando expressões numéricas numa linguagem semelhante às instruções assembler.

Nosso calculador reconhecerá um conjunto básico de operadores: Multiplicação, adição, parênteses e menos unário. Distinguiremos números reais de inteiros, para ilustrar uma sutileza nas especificações de gramáticas semelhantes ao 'bison'. Isto é tudo. O propósito desta seção é, depois de tudo, ilustrar um programa C++, usando um parser e um léxico e não construir um calculador completo.

Nas seções seguintes desenvolveremos a especificações de uma gramática para o 'bisonc++'. então, as expressões regulares para o 'scanner' estão especificadas de acordo com os requerimentos do 'flex'. Finalmente o programa é construído.

21.8.2.1: O arquivo de especificação do 'bisonc++'

O arquivo de especificação da gramática que o 'bisonc++' requer é comparável ao arquivo de especificações requerido pelo 'bison'. As diferenças são relativas à natureza da classe do parser resultante. Nosso calculador distinguirá números reais de inteiros e suportará o conjunto básico de operadores aritméticos.

O 'bisonc++' será usado como segue:

- Como usual, a gramática precisa ser definida. Com o 'bisonc++' não é diferente e as definições da gramática do 'bisonc++' são, para todo fim prático, idênticas às definições da gramática para o 'bison'.
- Tendo especificado a gramática e (usualmente) algumas declarações, o 'bisonc++' estará habilitado a gerar arquivos que definem a classe parser e a implantar a função membro 'parse()'.
- Todos os membros da classe (exceto aqueles requeridos para o funcionamento próprio do membro 'parse()') devem ser implantadas pelo programador. Claro que podem ser também declaradas no cabeçalho da classe 'parser'. No final o membro 'lex()' será implantado. Este membro é chamado por 'parse()' para obter a próxima ficha disponível. Mas 'bisonc++' oferece uma implantação estandarte da função 'lex()'. A função membro 'error(char const *msg)' tem uma implantação padrão simples que pode ser modificada pelo programador. A função membro 'error()' é chamada quando 'parse()' detecta erros sintáticos.
- O parser agora pode ser usado num programa. Um exemplo bem simples seria:

```
int main()
{
    Parser parser;
    return parser.parse();
}
```

O arquivo de especificações do 'bisonc++' consiste de duas seções:

- Seção de declarações: Nesta seção as fichas e regras de prioridades para os operadores são declaradas. Contudo, o 'bisonc++' também suporta muitas declarações novas. Estas novas declarações são importantes e são discutidas abaixo.
- Seção de regras: As regras gramaticais que definem a gramática. Esta seção é idêntica à requerida pelo 'bison', apesar de que alguns membros disponíveis no 'bison' e 'bisonc++' são

considerados obsoletos no 'bisonc++', enquanto outros membros podem ser usados num contexto mais amplo. Por exemplo, 'ACCEPT()' pode ser chamada de qualquer bloco de ação do parser para terminar o processo de análise.

Os leitores familiarizados com o 'bison' devem notar que já não há seção de cabeçalho. As seções de cabeçalho são usadas pelo 'bison' para fornecer as declarações necessárias para assegurar que o compilador possa compilar as funções C geradas pelo 'bison'. Em C++ as declarações são parte ou já usadas pelas definições de classe, assim, um parser que gera uma classe C++ e algumas de suas funções membro não requer de uma seção de cabeçalhos.

A seção de declarações:

A seção de declarações contém diversas declarações, entre as quais todas as fichas usadas na gramática e regras de prioridades dos operadores matemáticos. Além disso, muitas especificações novas podem ser usadas aqui. As relevantes para nosso exemplo e só disponíveis no 'bisonc++' são discutidas aqui. O leitor deve se referir à 'manpage' do 'bisonc++' para uma descrição completa.

- - '%baseclass-header header': Define o caminho do arquivo que contém a classe de base do parser. Padrões do nome da classe do parser mais o sufixo 'base.h'.
- - '%baseclass-preinclude header': Usa o cabeçalho como o caminho do arquivo pré-incluído no cabeçalho da classe de base do parser. Esta declaração é útil em situações onde o arquivo cabeçalho da classe de base ainda não é conhecido. P.ex., no campo '%union a std::string *' deve ser usado. Como a classe 'std::string' ainda não é conhecida pelo compilador, uma vez que processe o arquivo cabeçalho da classe de base necessitamos um modo de informar o compilador sobre essas classes e tipos. O procedimento sugerido é usar um arquivo cabeçalho de pré-inclusão que declare os tipos requeridos. Como padrão o cabeçalho estará entre aspas (usando, p.ex., #include"header"). Quando o argumento estiver entre parênteses angulares '#include <header>' será incluído. Nesse caso, são necessárias aspas para escapar à interpretação pela 'shell' (p.ex., usando -H'<header>').
- - '%class-header header': Define o caminho do arquivo que conterá (ou contendo) a classe parser. Padrões do nome da classe parser mais o sufixo 'suffix.h'.
- - '%class-name parser-class-name': Declara o nome da classe deste parser. Esta declaração substitui '%name' usada em 'bison++'. Define o nome da classe C++ que será gerada. Ao contrário que no 'bison++' '%name', a declaração '%class-name' pode aparecer em qualquer parte da

primeira seção do arquivo de especificação da gramática. Só pode ser definido uma vez. Se não for declarado um nome para a classe, o nome 'Parser' será usado.

- - '%debug': Provê 'parse()' e suas funções de suporte com código de depuração, mostrando o processo de análise na 'stream' padrão de saída. Quando incluída, a saída de depuração é ativada como padrão, mas sua atividade pode ser controlada usando o membro 'setDebug(bool on-off)'. Note que não são usadas mais macros '#ifdef DEBUG'. Re-executando 'bic()' sem a opção '--debug' um parser equivalente é gerado sem conter o código de depuração.
- - '%filenames header': Define o nome genérico de todos os arquivos gerados, a menos que seja anulada por nomes específicos. Como padrão os arquivos usam o nome da classe como o nome genérico.
- - '%implementation-header header': Define o caminho do arquivo que conterá (ou contendo) a implantação do cabeçalho. Os caminhos padrões da classe parser gerada mais o sufixo '.ih'. A implantação do cabeçalho conterá todas as diretivas e declarações usadas só na implantação das funções membro de 'parser'. É o único arquivo cabeçalho que é incluído pelos arquivos fonte que contêm a implantação de 'parse()'. Se sugere que outros membros da classe definidos pelo usuário use a mesma convenção, concentrando, assim, todas as diretivas e declarações requeridas para a compilação de outros arquivos fonte pertencentes à classe 'parser' num só arquivo cabeçalho.
- - '%parsefun-source source': Define o caminho do arquivo que contém o membro de 'parser' 'parse()'. Padrões de 'parse.cc'.
- - '%scanner header': Usa o cabeçalho como o caminho do arquivo de pré-inclusão do cabeçalho da classe 'parser'. Este arquivo pode definir uma classe 'Scanner', oferecendo um membro 'yylex()' que produz a ficha seguinte da 'stream' de entrada a ser analisada pelo parser gerado por 'bisonc++'. Quando esta opção for usada o membro de 'parser' 'int lex()' será pré-definido como:

```
inline int Parser::lex()  
{  
    return d_scanner.yylex();  
}
```

E um objeto 'Scanner' 'd_scanner' será composto dentro do parser. O objeto 'd_scanner' será

construído usando seu construtor padrão. Se outro construtor for requerido, a classe 'parser' pode ser provida com um construtor 'parser' apropriado (sobrecarregado) depois de ter construído o arquivo cabeçalho padrão da classe 'parser' usando 'bisonc++'. Como padrão o cabeçalho estará entre aspas (p.ex., #include "header"). Quando o argumento estiver entre parênteses quadrados '#include <header>' será incluído.

- - '%stype typename': O tipo de fichas semânticas. A especificação 'typename' deve ser o nome de um tipo desestruturado (p.ex., 'size_t'). Como padrão é inteiro. Veja 'YYSTYPE' no 'bison'. Não deve ser usado se uma especificação '%union' for usada. Com a classe 'parser', este tipo pode ser usado como 'STYPE'.
- - '%union union-definition': Atua identicamente à declaração do 'bison'. Como no 'bison', gera a união para o tipo de semântica do parser. O tipo da união é 'STYPE'. Se não for declarada uma união, um tipo de pilha simples pode ser definido usando a declaração '%stype'. Se não se usar '%stype', o padrão do tipo de pilha (inteiro) é usado.

Um exemplo de declaração '%union' é:

```
%union
{
    int      i;
    double   d;
};
```

Uma união não pode conter objetos como campo, já que os construtores não podem ser chamados na criação da união. Isto significa que uma 'string' não pode seer um membro da união. Uma 'string *', contudo, é um membro possível da união. O 'scanner' léxico não precisa conhecer tal união. O 'scanner' pode simplesmente passar o texto escaneado ao parser através da função membro 'YYText()'. Por exemplo usando um comando como:

```
$$ .i = A2x(scanner.YYText());
```

O texto concordante será convertido a um valor de tipo apropriado.

As fichas e não terminais devem ser associados com os campos da união. Isto é fortemente advertido, já que evita confusões de tipos e o compilador estará habilitado a examinar a exatidão dos tipos. Ao mesmo tempo, as variáveis específicas do 'bison' '\$\$', '\$1', '\$2', etc. podem ser usadas, no lugar de especificações de campos completos (como '\$\$.i'). Um não terminal ou ficha pode ser associado com um campo da união usando a especificação <fieldname>. P.ex.:

```
%token <i> INT           // ficha associada (obsoleta, veja abaixo)
      <d> DOUBLE
```

```
%type <i> intExpr // associação não terminal
```

No exemplo desenvolvido aqui, note que as fichas e os não terminais podem ser associados com os campos da união. Contudo, como notado antes, o 'scanner' léxico não tem que conhecer nada sobre tudo isto. Em nossa opinião, é mais claro deixar ao 'scanner' só uma coisa: Examinar textos. O parser que sabe tudo a respeito da entrada, pode, então, converter 'strings' como “123” num valor inteiro. Conseqüentemente, a associação de um campo da união a uma ficha é desencorajada.

A ilustração da descrição acima das regras da gramática será dada mais adiante.

Na discussão de '%union', a especificação de '%token' e '%type' pode ser notada. São usadas para especificar as fichas (símbolos terminais) que podem ser retornados pelo 'scanner' léxico e para especificar os tipos não terminais. Aparte de '%token', os indicadores de fichas:

```
%left,  
%right and  
%nonassoc
```

Podem ser usados para especificar a associatividade de operadores. As fichas mencionadas nesses indicadores são interpretadas como fichas que indicam operadores associados na direção indicada. A precedência dos operadores é dada por sua ordem: a primeira especificação possui a menor prioridade. Para sobrepassar a regra uma certa precedência num contexto determinada, '%prec' pode ser usada. Como todas estas são práticas estandartes do 'bisonc++', não estão mais elaboradas aqui. A documentação que acompanha a distribuição do 'bisonc++' deve ser consultada para maiores esclarecimentos.

Eis a seção de declarações do calculador:

```
%filenames parser  
%scanner ../scanner/scanner.h  
%lines  
  
%union {  
    int i;  
    double d;  
};  
  
%token INT  
DOUBLE  
  
%type <i> intExpr  
%type <d> doubleExpr  
  
%left '+'  
%left '*'  
%right UnaryMinus
```

Na seção de declarações, os especificadores '%type' são usados, associando o valor 'intExp' das regras (veja próxima seção) ao campo 'i' dos valores semânticos da união e o valor 'doubleExp' ao campo 'd'. A primeira vista isto parece complexo, já que as expressões das regras precisam ser incluídas para cada tipo de retorno individual. Por outro lado, se a própria união tivesse sido usada, ainda assim teríamos que haver especificado em algum lugar os valores de retorno semânticos que campos usar: um código com menos regras, mas mais complexo e propenso a erros.

As regras gramaticais

As regras e ações gramaticais são especificadas como usual. A gramática para nosso pequeno calculador é dada abaixo. Existem bem poucas regras, mas ilustram várias características oferecidas pelo 'bisonc++'. Em particular, note que não requer bloco de ação mais que uma linha de código. Isto mantém a organização da gramática relativamente simples e por isso aumenta a legibilidade e compreensibilidade. Mesmo a regra que define a terminação própria (a linha vazia na linha de regra) usa uma simples chamada a uma função membro 'done()'. A implantação dessa função é simples, mas interessante pelo que chama 'parser::ACCEPT()', mostrando que o membro 'ACCEPT()' pode ser chamado indiretamente de dentro do bloco de produção de ações das regras. Eis as regras de produção da gramática:

```
lines:
    lines
    line
|
    line
;

line:
    intExpr
    '\n'
    {
        display($1);
    }
|
    doubleExpr
    '\n'
    {
        display($1);
    }
|
    '\n'
    {
        done();
    }
|
    error
    '\n'
    {
```

```

        reset();
    }
;

intExpr:
    intExpr '*' intExpr
    {
        $$ = exec('*', $1, $3);
    }
|
    intExpr '+' intExpr
    {
        $$ = exec('+', $1, $3);
    }
|
    '(' intExpr ')'
    {
        $$ = $2;
    }
|
    '-' intExpr          %prec UnaryMinus
    {
        $$ = neg($2);
    }
|
    INT
    {
        $$ = convert<int>();
    }
;

doubleExpr:
    doubleExpr '*' doubleExpr
    {
        $$ = exec('*', $1, $3);
    }
|
    doubleExpr '*' intExpr
    {
        $$ = exec('*', $1, d($3));
    }
|
    intExpr '*' doubleExpr
    {
        $$ = exec('*', d($1), $3);
    }
|
    doubleExpr '+' doubleExpr
    {
        $$ = exec('+', $1, $3);
    }

```



```

    }
|
doubleExpr '+' intExpr
{
    $$ = exec('+', $1, d($3));
}
|
intExpr '+' doubleExpr
{
    $$ = exec('+', d($1), $3);
}
|
'(' doubleExpr ')'
{
    $$ = $2;
}
|
'-' doubleExpr          %prec UnaryMinus
{
    $$ = neg($2);
}
|
DOUBLE
{
    $$ = convert<int>();
}
;

```

A gramática acima é usada para implantar um calculador simples onde valores inteiros e reais podem ser negados, somados e multiplicados e onde as regras estandartes de prioridades podem ser evitadas usando-se parênteses. A gramática mostra o uso de símbolos não terminais tipificados: 'doubleExpr' corresponde aos valores reais ('double'), 'intExpr' corresponde aos valores inteiros. As associações de precedência e tipo são definidas na seção de definições do parser.

O arquivo cabeçalho de 'Parser'

Várias funções chamadas desde a gramática estão definidas como modelos de funções. O 'bisonc++' gera vários arquivos, entre os quais o arquivo que define a classe 'parser'. As funções chamadas dos blocos das regras de produção de ação são usualmente funções membro da 'parser' e estas funções membro precisam ser declaradas e definidas. Uma vez que o 'bisonc++' tenha gerado o arquivo cabeçalho que define a classe 'parser', automaticamente re-escreverá este arquivo, permitindo ao programador adicionar novos membros à classe parser. Eis o arquivo 'parser.h' usado para nosso pequeno calculador:

```

#ifndef Parser_h_included
#define Parser_h_included

```

```

#include <iostream>
#include <sstream>
#include <fbb/a2x.h>

#include "parserbase.h"
#include "../scanner/scanner.h"

#undef Parser
class Parser: public ParserBase
{
    std::ostringstream d_rpn;
    // $insert scannerobject
    Scanner d_scanner;

public:
    int parse();

private:

    void display(int x);
    void display(double x);
    void done() const;

    static double d(int i)
    {
        return i;
    }

    template <typename Type>
    Type exec(char c, Type left, Type right)
    {
        d_rpn << " " << c << " ";
        return c == '*' ? left * right : left + right;
    }

    template <typename Type>
    Type neg(Type op)
    {
        d_rpn << " n ";
        return -op;
    }

    template <typename Type>
    Type convert()
    {
        Type ret = FBB::A2x(d_scanner.YYText());
        d_rpn << " " << ret << " ";
        return ret;
    }
}

```

```

void reset();

void error(char const *msg)
{
    std::cerr << msg << std::endl;
}

int lex()
{
    return d_scanner.yylex();
}

void print()
{}

// funções de suporte para parse():

void executeAction(int d_production);
unsigned errorRecovery();
int lookup(int token);
int nextToken();
};

#endif

```

21.8.2.2: O arquivo de especificação do 'flex'

O arquivo de especificação do 'flex' usado pelo nosso calculador é simples: os espaços em branco são saltados, um único caracter é retornado e os valores numéricos como fichas 'Parser::INT' ou 'Parser::DOUBLE'. Eis o arquivo de especificação do 'flex' completo:

```

%{
#define _SKIP_YYFLEXLEXER_
#include "scanner.i"

#include "../parser/parserbase.h"
%}

%option yyclass="Scanner" outfile="yylex.cc" c++ 8bit warn noyywrap
%option debug

%%

[ \t]                ;
[0-9]+               return Parser::INT;

"."[0-9]*            |

```

```

[0-9]+ ("." [0-9] *) ?           return Parser::DOUBLE;

.|\\n                             return *yytext;

%%

```

21.8.2.3: Gerando o código

O código é gerado da mesma forma no 'bison' e no 'flex'. Para ordenar ao 'bisonc++' que gere os arquivos 'parser.cc' e 'parser.h' se dá o comando:

```
bison++ -V grammar
```

A opção '-V' gerará o arquivo 'parser.output' que mostra a informação sobre a estrutura interna da gramática fornecida, entre outros seus estados. É útil para fins de depuração e pode ficar fora do comando se a depuração não é requerida. O 'bisonc++' pode detetar conflitos ('shift-reduce conflicts' e/ou 'reduce-reduce conflicts') na gramática fornecida. Estes conflitos podem resolvidos explicitamente usando as regras de desambiguação ou são 'resolvidas' por padrão. Um conflito 'shift-reduce' é resolvido deslocando, i.e., a próxima ficha é consumida. Um conflito 'reduce-reduce' é resolvido usando as duas primeiras regras de produção que competem. O 'bisonc++' usa procedimentos de resolução de conflitos idênticos aos do 'bison' e 'bison++'.

Uma vez construídas a classe 'parser' e a função membro 'parse()', usamos o 'flex' para criar o 'scanner' léxico (i.e., o arquivo 'yylex.cc') com o comando:

```
flex -I lexer
```

Sobre os sistemas Unix, a compilação e linkagem das fontes geradas e a fonte para o programa principal (dado abaixo) é então realizada por um comando como:

```
g++ -o calc -Wall *.cc -s
```

Uma fonte onde a função 'main()' e o objeto 'parser' (tendo o 'scanner' léxico como um de seus membros de dados), finalmente, aqui está:

```

#include "parser/parser.h"
using namespace std;

int main()
{
    Parser parser;

    cout << "Entre expressões (aninhadas) contendo inteiros, duplos, *, + e "
          "operadores unários -\\n"
          ". Entre uma linha vazia, 'exit' ou 'quit' para sair.\\n";

    return parser.parse();
}

```

}

O 'bisonc++ pode ser baixado de, p.ex.,

<http://www.sourceforge.net/projects/bisoncpp>