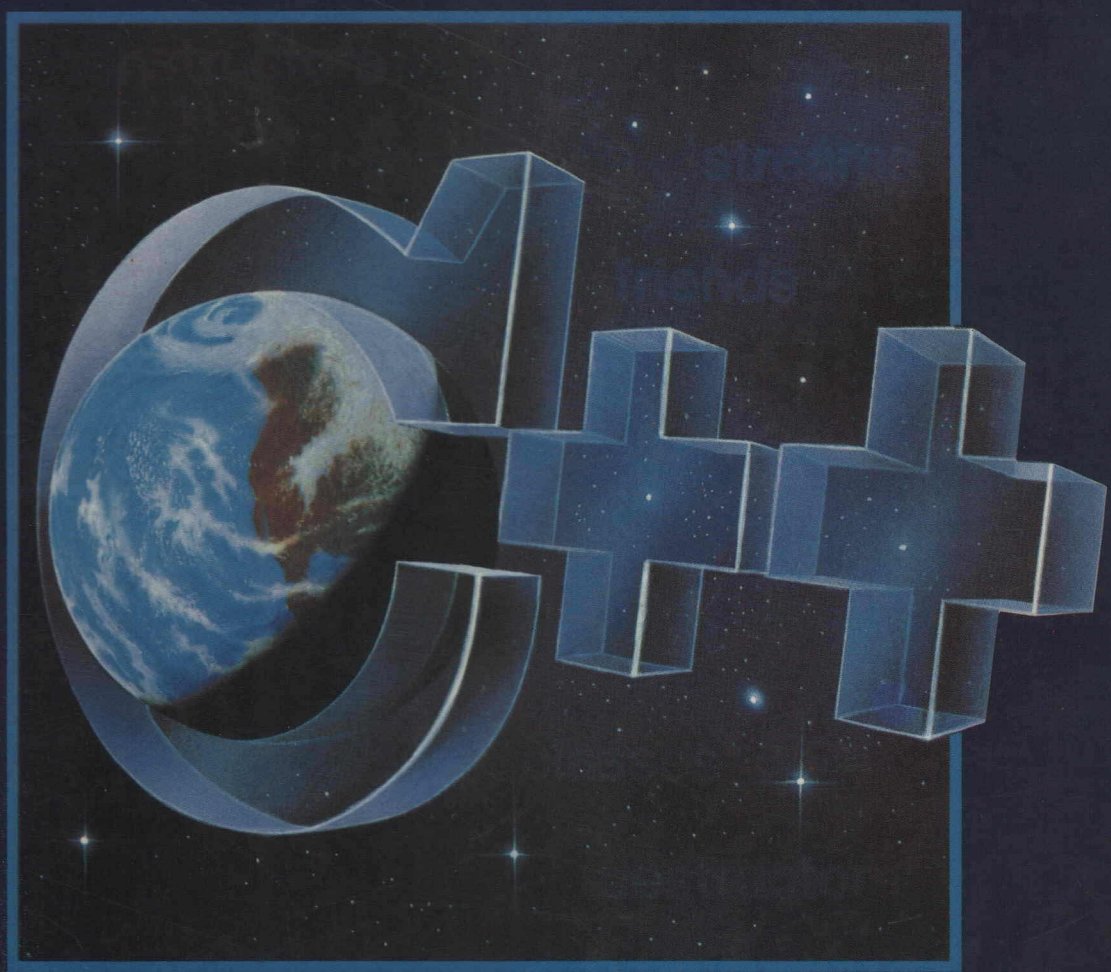
The Waite Group's

PROGRAMANDO EM C++

UNIX® e MS/DOS®



MAKRON Books

John Berry

6500,00

Programando em C++

Programando em C++

John Berry

Tradução
Mário Moro Fecchio

Revisão Técnica
Maria Estela S. Quintela

MAKRON *Books* do Brasil Editora Ltda.
Editora McGraw-Hill Ltda.
São Paulo
Rua Tabapuã, 1105, Itaim-Bibi
CEP 04533
(011) 829-8604 e (011) 820-8528

Rio de Janeiro • Lisboa • Porto • Bogotá • Buenos Aires • Guatemala • Madrid • México • New York • Panamá • San Juan • Santiago

Auckland • Hamburg • Kuala Lumpur • London • Milan • Montreal • New Delhi • Paris • Singapore • Sydney • Tokyo • Toronto

Do original

C++ Programming

EDITOR: MILTON MIRA DE ASSUMPÇÃO FILHO

Produtora Editorial: Daisy Pereira Daniel
Produtor Gráfico: José Rodrigues

Editoração Eletrônica: E.R.J. Informática Ltda.

Dados de Catalogação na Publicação (CIP) Internacional
(Câmara Brasileira do Livro, SP, Brasil)

Berry, John Thomas, 1947-
Programando em C++ / John Berry ; tradução Mário Moro Fecchio ; revisão técnica Maria Estela S. Quintela. -- São Paulo : Makron, McGraw-Hill,

1. C++ (Linguagem de programação para computadores) I. Título.

91-0555 CDD-001.6424

Índices para catálogo sistemático:

1. C++ : Linguagem de programação : Computadores : Processamento de dados 001.6424

Para minha esposa Nancy e minha filha Rebeca,
por sua paciência, ajuda e, sobretudo, amor.

Aos meus estudantes no Foothill College
que compreenderam se os seus projetos
algumas vezes estavam um pouco atrasados.

Makron Books

Agradecimentos

Só pensar na enormidade de uma tarefa como esta de escrever um livro é arrasador, mesmo agora, que ele está terminado. Este empreendimento não teria sido possível sem a ajuda de muitas pessoas que dedicaram mais tempo do que se deveria esperar; elas ajudaram a manter os erros em um nível mínimo.

Mitch Waite, Scott Calamar, e especialmente Jim Stockford, do Grupo Waite.

Harry Henderson e Gary Masters, meus editores.

Takanori Adachi, Ivar von Elsnitz, e especialmente John Carolan da Glockenspiel Limited, que forneceram o suporte técnico.

Além desses, eu gostaria de agradecer às seguintes empresas por sua cooperação e ajuda:

Glockenspiel Limited
Lifeboat Associates
Guidelines
Miwa Systems Consulting Company, Ltd.
Oasys

John Berry

O Grupo Waite estende seus agradecimentos às seguintes pessoas: Harry Henderson fez um belo trabalho como editor de conteúdo e reescrita; Gary Masters

fez o mais fluente trabalho que poderíamos esperar na cópia-edição; agradecimentos a John Carolan pelo tempo que despendeu proporcionando-nos uma cuidadosa revisão técnica; agradecimentos também a Joseph McIsaac, Ivar von Elsnitz, Craig Hubgley e Bob MacIlree por seu apoio.

Grupo Waite

Marcas Registradas

Makron
Books

Sumário

Prefácio

Mesmo durante o curto espaço de tempo em que este livro estava sendo escrito, a linguagem C++ transformou-se, de uma linguagem de programação relativamente desconhecida, usada primariamente sob o UNIX, numa ferramenta de desenvolvimento de software amplamente discutida. Há rumores em abundância sobre seu último domínio do desenvolvimento de software tanto em sua versão original UNIX bem como em MS-DOS e no OS/2 em desenvolvimento. Ao mesmo tempo em que este livro é impresso, alguns programadores profissionais já estão prevendo que C++ ultrapassará C como linguagem dominante na indústria de software.

Com todo o interesse em C++ e sua migração para os ambientes MS-DOS e OS/2, logo se tornou óbvio que havia uma grande necessidade de um livro-texto que servisse como introdução a essa linguagem de programação. Programadores C experientes, especialmente aqueles que se concentram no IBM-PC e seus clones, necessitam de uma introdução à linguagem C++ que dê ênfase a seus aspectos práticos. Eles precisam de algo para mostrar como a nova sintaxe complexa — mas poderosa — desse ambiente de programação oferece soluções eficientes aos problemas cotidianos do programador profissional. Estas necessidades foram a motivação para escrever este livro.

Embora C++ seja uma ferramenta mais poderosa do que C, a mudança da antiga linguagem para a nova é uma transição mais simples do que você espera. C++ é mais do que um movimento "evolucionário" de C. O núcleo básico de

proposições e operadores é o mesmo, incluindo elementos como definição de função, compilação separada e outras ferramentas de modularidade usadas comumente pelo programador profissional C. Muitos programas C são também válidos em C++. O que C++ acrescenta a essa sintaxe básica é a habilidade em criar *classes*; essas estruturas completas criam uma acentuada modularidade dentro de um programa. Elas são também o fundamento para a implementação de muitos dos conceitos de programação objeto-orientada. C++ contém também muitos aperfeiçoamentos adicionais em relação à sintaxe C original; muitos deles — mas nem todos — estão espelhados no novo padrão ANSI para C.

É importante encarar estas novas e complexas características usadas num contexto de programação e projeto não apenas como discussões teóricas que forçam o leitor a aplicar esse novo conhecimento como exercícios do tipo tentativa-e-erro. Neste livro, procuramos combinar as discussões com exemplos que mostram a característica que foi destacada, no contexto em que ela deverá ser usada. Isso é particularmente importante com uma ferramenta de programação tão nova e evoluída como C++. Note que os programas deste livro foram idealizados com o fim de ilustrar pontos específicos sobre a linguagem. Embora se abstenham das "perfumarias" que tendem a obscurecer o ponto em discussão, eles permanecem como firmes fundamentos para sólidas aplicações de C++.

Como você verá, o contexto é extremamente importante, porque nem tudo o que é possível em C++ é utilizável. As armadilhas esperam pelos incautos, como acontece com qualquer software. O próprio autor já caiu em muitas delas; assim, este livro pode tornar a sua jornada um pouco mais suave.

John Thomas Berry
San Francisco

Makron
Books

Capítulo 1
Escrevendo Seu Primeiro Programa C++

Este capítulo destaca os recursos da linguagem C++ e demonstra como utilizá-los em seus programas. Iniciando com conceitos familiares da linguagem C, mostra como a linguagem C++ proporciona tanto pequenas melhorias quanto grandes e novas facilidades que suportam um novo modelo para desenvolvimento de programa. O capítulo termina com uma visão geral do desenvolvimento de programa C++ nos ambientes UNIX e MS-DOS. Os capítulos seguintes trazem discussões profundas de cada um dos recursos apresentados aqui, juntamente com numerosos exemplos de programação.

C++ é derivada de C

Um dos poderes de C++ é que ela deriva diretamente da linguagem de programação C, e, portanto, tem suas bases em terreno familiar. Sem dúvida, C é

um subgrupo de C++. Devido ao fato de C++ ser usualmente implementada como um tradutor que produz o código regular de C a partir dos comandos-fonte C++, C++ não é uma linguagem de programação completamente nova que requer um longo aprendizado. Se você já sabe como programar em C, você conhece a maior parte da sintaxe básica de C++. Você tem pouco a aprender, de maneira que pode concentrar-se na riqueza de recursos que C++ acrescenta à linguagem anterior.

É impossível incluir neste livro uma discussão da sintaxe completa de C. Vamos assumir, portanto, que você tenha uma familiaridade básica com aquela linguagem. No entanto, o breve resumo a seguir destaca alguns importantes elementos de programação e aponta algumas das diferenças introduzidas por C++.

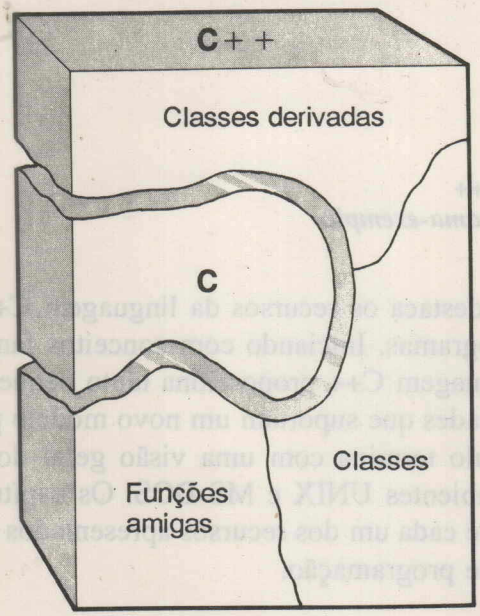


Figura 1.1 A relação entre C e C++.

A Forma de um Programa C++

Um programa C++ consiste em uma série de uma ou mais funções. Estas funções podem ser combinadas em um único arquivo, ou podem estar espalhadas através de vários arquivos em disco. Deve haver uma — e somente uma — função chamadas de *main()*. Embora a função *main()* esteja onde começa oficialmente a execução do programa, isso não é categórico, e discussões posteriores mostram que é possível executar código tanto antes que a função *main()* seja chamada quanto depois que ela termina a execução.

C++ permite também grande modularidade na programação através de várias suplementações interessantes na programação C usual. Um dos recursos — classe — integra funções em estruturas de dados; uma outra, permite a sobrecarga de nomes de funções, possibilitando que duas ou mais funções compartilhem de uma seqüência de chamada comum.

Semelhanças Globais entre C e C++

C++ está estreitamente ligada a C: contém as mesmas proposições e construções na programação. Ela tem essencialmente o mesmo escopo e as mesmas regras para armazenamento de classes, e até mesmo os operadores são idênticos. Naturalmente, C++ oferece mais, porém é importante que você reconheça as semelhanças. Estas semelhanças são melhor demonstradas por um programa-exemplo que mostra claramente a relação *sobregrupo-subgrupo*. Para resumir, *day.c* (Listagem 1.1) é um bom programa C e um programa C++ válido.

Listagem 1.1 Um programa válido C e C++: *day.c*

```
# include <stdio.h>

char* days[] = { "5a. feira", /* define os nomes dos dias */
                 "6a. feira",
                 "Sabado",
                 "Domingo",
                 "2a. feira",
                 "3a. feira",
                 "4a. feira"};
```

```
main(argc,argv)
int argc;
char* argv[];
{
  int jdate;
  char* day;

  if(argc == 1) { /* verifica argumentos da linha de comando */
    printf("formato: Listal <data juliana>\n");
    exit(1);
    }
  jdate = atoi(argv[1]); /* converte argumento em inteiro */
  day = days[jdate % 7]; /* pesquisa o nome correto do dia */
  printf("dia = %s\n",day);
}
```

Este simples programa aceita uma data do calendário Juliano (definida aqui como sendo o número de dias a partir de 1º de Janeiro) e dá como resposta o dia da semana. A chave deste programa é a cadeia de caracteres externa, *days()*, contendo os dias da semana. A data do calendário Juliano é submetida a uma divisão módulo 7. Isso dá como resultado um número entre 0 e 6, ou seja, o resto da divisão por 7. Esse resto é usado como um índice do arranjo. A disposição dos dias da semana no arranjo não é aleatória, mas sim definida para o ano corrente. A segunda célula no arranjo — índice 1 — é destinada ao dia da semana em que cai no primeiro dia do ano; e a contagem prossegue a partir daí. Outra característica interessante do programa é a interface da linha de comando. Se você rodar o programa sem o necessário argumento, ele exibe o formato da linha de comando correta e sai.

Embora *day.c* seja um legítimo programa C e C++ que compila tanto no tradutor C quanto C++, se você fosse escrever este programa puramente em C++ faria algumas coisas de maneira diferente. Conseqüentemente, quando você compilar este programa usando um compilador comercial C++, o programa gerará algumas mensagens de aviso. Não obstante, ele é uma boa ilustração da estreita relação entre estas duas linguagens de programação.

As características básicas e a sintaxe comuns a C e C++ incluem:

- a função *main()*
- o uso de argumentos de função

- os operadores aritméticos e lógicos
- as proposições de controle e looping
- a manipulação bit a bit
- os tipos de dados básicos

No entanto, tenha em mente que mesmo estas áreas similares têm características C++ distintas. O poder que reside na linguagem C++ torna-a muito mais do que apenas outra versão de C, e, como você aprenderá neste livro, o poder desta linguagem define uma nova aproximação para a programação. Assim, o exemplo da Listagem 1.1 é C++ válido, mas não bom C++.

C++ não é apenas C

Não superestime as semelhanças entre C e C++. Há diferenças significativas, algumas das quais são fáceis de perceber. E mais, qualquer comparação entre as duas linguagens é complicada pelo fato de que C está em período de transição. O padrão em nível industrial da linguagem C difere em alguns pontos da sintaxe da linguagem C tradicional. E este padrão geralmente espelha algumas das mudanças introduzidas pela linguagem C++. Portanto, este livro concentra-se em C++ e não a compara extensamente com a linguagem-padrão C. Desta forma, você adquire o sentimento do espírito de C++ sem ter que perder tempo considerando diferentes construções e estilos de programação. Ao mesmo tempo, examinaremos brevemente as semelhanças com o padrão ANSI para benefício daqueles programadores que já estão usando-o.

Declarações em C++

Uma das diferenças mais evidentes entre as duas linguagens está na declaração de variáveis. Embora um número inteiro seja ainda declarado como *int x* em C++, a ordem de declaração dentro do programa é mais flexível. Lembre-se de que, em C, todas as declarações devem ocorrer no início de um bloco de função

ou um bloco criado por um par de chaves ({ e }). Você precisa declarar todas as variáveis não apenas antes delas serem usadas, mas antes de qualquer proposição executável. Por exemplo, uma função que calcula uma média de uma série de números pode aparecer como *mean.c* na Listagem 1.2.

Listagem 1.2 Uma função para ilustrar declaração: *mean.c*

```
double mean(num,size)
double num[];
int size;
{
int loop;
double total = 0;

for (loop=0; loop<size; loop++)
  total += num[loop];

return total/size;
}
```

Dentro das chaves, a função é dividida em duas partes: uma seção de declaração e uma seção de ação.

Em C++, as declarações podem ser colocadas em qualquer lugar no programa. Na realidade, devido ao estilo da linguagem C++, você deverá manter as declarações tão perto do seu ponto de uso quanto possível. Isso serve para salientar sua relação com as proposições que as utilizam e lembra ao programador sobre o seu escopo. Uma revisão da função-exemplo *mean()* como *mean2.c* (Listagem 1.3) ilustra este ponto.

Listagem 1.3 A função *mean()* reescrita: *mean2.c*

```
double mean(double num, int size)
{
double total = 0;
for (int loop == 0; loop<size; loop ++)
  total += num[loop];

return total/size;
}
```

Em *mean2.c* o loop da variável inteira ocorre somente dentro das instrução *for*. Portanto, você pode postergar sua declaração até a execução da instrução *for*.

Observe que o formato das declarações de parâmetro também é novo. Esta mudança é discutida numa seção mais adiante neste capítulo.

O Escopo de uma Variável

As regras de escopo de C++ são semelhantes às de C. Há três possíveis campos para uma variável ou outro dado objeto: *local, file* e *class*. A discussão de *class* em C++ será feita posteriormente neste capítulo. Por enquanto, examinaremos os outros dois campos.

Uma variável local é usada exclusivamente dentro de um bloco. Os blocos são definidos por um par de chaves. Por exemplo, em

```
example()
{
int x,y;
   .
   .
}
```

as variáveis *x* e *y* são locais para a função *example()*.

Um bloco pode também ser definido dentro de um outro bloco, como mostramos a seguir:

```
example2()
{
   int x,y;
   {
   int p;
      .
      .
   }
   .
   .
}
```

Neste exemplo, *x* e *y* são locais para *example2()*, mas a variável *p* é local para o bloco interno definido pelo segundo par de chaves: isso acontece tanto em

C++ como em C. No entanto, esta forma é raramente utilizada em qualquer uma das linguagens.

O resultado principal com o escopo é a *visibilidade* ou *acessibilidade* da variável. Uma variável local é conhecida somente dentro de seu próprio bloco. No último exemplo, *p* pode ser acessada apenas dentro do bloco interno; ela é invisível aos outros blocos. A variável *y*, no entanto, ilustra outro aspecto das regras de visibilidade, ou seja, sua relatividade. A variável-objeto *y* está disponível dentro do bloco externo bem como no bloco interno.

Em contraste com uma variável local, uma variável do campo *file* é definida externamente a qualquer função ou classe. A disponibilidade deste tipo de dado-objeto se estende desde o ponto de definição até o fim do arquivo-fonte no qual ela é definida, independente do número de blocos envolvidos. Por exemplo, o seguinte código:

```
int count;
example()
{
    .
    .
}

example2()
{
    .
    .
}
```

define uma variável inteira, *count*, que pode ser acessada tanto em *example()* como em *example2()*. Isso corresponde à variável *global* de outras linguagens de programação.

Um aspecto importante do campo das variáveis está ligado à noção de dado ou ocultamente de variável. Considere o seguinte trecho de código:

```
int count = 5

example()
{
int count = 10
    .
    .
}
```

Parece haver um conflito entre os dois objetos com o mesmo campo. Qual é a versão de *count* usada pela função? A resposta é simples, e é a mesma tanto em C como em C++. O nome mais local tem a precedência. Aqui, a variável *count*, que está definida dentro de *example()*, é usada dentro da função. O que há de novo em C++ é a possibilidade de contornar esta restrição e referir-se a uma variável ou outro nome externamente ao campo corrente, mesmo que aquele objeto seja ocultado por um nome local. Você pode fazer isso usando o operador de resolução de abrangência (::).

O operador de resolução de abrangência muda a referência de um nome a partir de uma variável local para uma variável de abrangência do arquivo. O exemplo *scope.c* (Listagem 1.4) demonstra o uso deste operador.

Listagem 1.4 O operador de resolução de abrangência: *scope.c*

```
int count = 5;

int example()
{
int count = 10;

cout << "contagem interna = " << count;
cout << "contagem externa = " << ::count;
}
```

Colocando :: na frente da variável, você a força a se referir externamente à abrangência definida para o arquivo. Naturalmente, se não existir nenhuma variável com aquele nome na abrangência global, ocorre um erro. (A instrução *count* é uma instrução simples de saída que exibe texto e valores de variáveis.)

Classes de Armazenamento em C++

A abrangência é apenas um dos fatores com que o projetista deve tomar cuidado quando estiver criando um aplicativo. A *classe de armazenamento* de uma variável também afeta a maneira pela qual um programa usa a variável. Basicamente, a classe de armazenamento de uma variável ou outro dado-objeto determina o período de tempo em que aquele objeto existe. Embora as quatro classes de

armazenamento C++ sejam as mesmas, como em C, C++ acrescenta algumas nuanças ao seu uso.

Uma variável de classe de armazenamento *automático* existe somente quando o programa executa o bloco no qual a variável está definida. Por exemplo, em

```
example()
{
int i = 0;
    .
    .
}
```

i é uma variável automática. É alocado espaço para ela enquanto a função *example()* está sendo executada. No entanto, aquele espaço é retomado tão logo o controle retorna à função que chamou *example()*. Você pode definir uma variável automática usando o designador *auto* na definição, mas, devido ao fato de esta ser a condição normal para uma variável local, a definição explícita quase nunca é utilizada.

Você pode também definir uma variável como *static* (estática). Uma variável static passa a existir quando o programa começa a ser executado, e permanece até que ele termine. Isso permite que a variável conserve o seu valor mesmo se a execução do programa passar além de sua abrangência. Considere o seguinte trecho de código:

```
example()
{
static int x = 0;
x++
    .
    .
}
```

Neste caso, *static* modifica a declaração da variável na função. Observe que a abrangência da variável *x* não mudou; ela continua sendo acessível somente enquanto o programa está executando *example()*. No entanto, por ela ser static, suas localizações de memória são mantidas a cada vez que a função é chamada, e o valor armazenado em *x* é conservado indefinidamente.

Você pode também definir uma variável de abrangência de arquivo como *static*:

```
static int i ;
example()
{
    .
    .
}
```

Esta declaração restringe a abrangência de validade da variável *i* ao arquivo ao qual ela está definida. Em C++, bem como em C, grandes aplicações geralmente envolvem vários arquivos. Uma declaração estática restringe o escopo de uma variável global a seu arquivo de origem.

Observe que a instrução *int* nos dois exemplos anteriores é redundante. Uma definição de *static x* assume que o dado será do tipo *int*. No entanto, uma declaração static de qualquer outro tipo de dado deve ser feita explicitamente. Por exemplo, você tem que definir um número real como *static double x*.

Você define uma variável ou o nome de uma função num programa como sendo *extern* para referir-se a uma declaração encontrada em outro arquivo. Inicialmente isso foi idealizado para facilitar a compilação separada de arquivos, porém, em C++, qualquer variável ou função que você não declarar explicitamente como static será do tipo *extern* implicitamente.

Uma declaração de *register* (registro) funciona da mesma maneira que em C. Afinal, é apenas uma sugestão para o compilador, sugestão esta que é ignorada se não houver registros disponíveis. Um valor de registro deve ser um número inteiro — *char, int, long*, e assim por diante, cujo tamanho caiba no registro. (O tamanho máximo, naturalmente, depende do hardware.)

Objetos de Armazenamento Livre — Uma Quinta Classe de Armazenamento

C++ tem, na realidade, uma quinta classe de armazenamento — objetos criados usando o operador C++ *new*. Esse operador aloca dinamicamente espaços específicos de memória. Uma vez alocados, estes objetos permanecem existindo até que você use o operador *delete* para desalojá-los especificamente.

Objetos manipulados pelos operadores *new* e *delete* não são automáticos porque não são destruídos quando o programa deixa sua abrangência. Porém, também não são static. Um dado-objeto static existe durante toda a execução do programa. Uma variável criada pela instrução *new* pode ser destruída e seu espaço de memória retorna ao banco de memória disponível em qualquer instante no programa. Esta propriedade dá ao programador um controle completo sobre o uso de memória dinânimca.

Declarando o Valor de uma Constante em C++

Na linguagem C tradicional — não na linguagem-padrão ANSI — o recurso do pré-processador macro cria uma espécie de valor *constant* (constante). A instrução *# define RATE 1.5* em uma linha substitui a cadeia de caracteres *RATE* pelo valor *1.5* no programa inteiro. Isso é feito no estágio do pré-reprocessador, antes da compilação, e é uma substituição puramente textual.

C++ oferece uma constante-objeto verdadeira. Colocando antes de uma definição o determinador *const*, você pode criar um dado-objeto que tem abrangência e visibilidade semelhante à variável, mas também com um valor que o programa não pode alterar. Assim, no trecho de programa

```
const x = 1.2;
example()
{
    .
    .
int y = x*23;
    .
    .
}
```

x é uma constante inteira que pode ser usada em qualquer lugar onde se possa usar uma variável, exceto do lado esquerdo como instrução de atribuição. A principal vantagem que o modificador *const* tem sobre a *interpretação* mais tradicional *#define* é que a constante é uma localização de memória semelhante a uma variável e não é meramente um identificador substituível. Esta distinção é uma consideração importante quando se está usando ponteiros, porque você pode combinar este modificador com a declaração de uma variável de ponteiro.

Ponteiros em C++

Variáveis pointer, ou seja, variáveis que contêm o endereço de outra variável, operam de maneira análoga em C e C++. Para um ponteiro do tipo criado pelo usuário, ou embutido, você deve definir uma variável pointer como sendo um ponteiro. A variável pointer mais usada habitualmente em C++ é o ponteiro de caractere, que é o tipo base para a cadeia de caracteres. Quase tão comuns quanto estes são as variáveis pointer para os tipos de dados complexos que dominam um programa C++, incluindo estruturas, arranjos e classes (estas últimas somente em C++).

A sintaxe para pointer é a mesma em ambas as linguagens. Numa definição de variável, o operador * designa uma variável pointer. Por exemplo, a definição *char* x* cria uma variável de cadeia de caracteres. O operador * faz com que a definição da variável *x* aloque espaço, não para o valor do caractere, mas, sim, para o endereço da localização de memória que contém tal valor. Lembre-se, a definição de uma variável pointer não cria o espaço necessário para armazenar um valor, mas sim, somente o lugar para armazenar um endereço. Para armazenar aquele valor, você tem que usar o operador *new* ou atribuir à variável o valor de alguma variável já existente.

Existem também disponíveis em C++ os mesmos dois operadores que manipulam variáveis pointer em C: & e *. O operador &, chamado de *address-of* coloca num ponteiro o endereço de uma variável existente, da seguinte maneira:

```
int x = 123;
int *y = &x;
```

Após executar estas instruções, a variável *y* contém o endereço da variável *x*. O operador *, chamado de *ponteiro de referência indireta*, executa a operação complementar que consiste em obter o valor que está na localização de memória cujo endereço está armazenado no ponteiro. O trecho de programa a seguir:

```
int x, y = 123;
int z = &y;

x = *z;
```

copia em *x* o valor que está em *y*, através do ponteiro *z*. O operador * executa um acesso indireto de memória: ele lê o valor que está na localização de memória *z*

e então usa este valor como endereço para uma localização de memória que contém outro valor. Este último valor é aquele usado pelo operador de atribuição.

O acesso do ponteiro às funções é o mesmo, tanto em C++ como em C. Um par extra de parênteses indica a *referência indireta*. Por exemplo, a declaração de um ponteiro para a função deve ser semelhante a:

```
int (*f)();
```

Você deve colocar o nome da variável entre parênteses, para garantir a correta execução. Note que a declaração *int*f()* especifica uma função que dá como resultado um número inteiro para um ponteiro, e essa não é a mesma operação que mostramos no exemplo anterior. O operador *address-of* (&) atribui o valor ao ponteiro da seguinte maneira:

```
int (*f)();
f = &example();
(*f)();
```

Este trecho de programa atribui o endereço da função *example()* à variável *f*, que foi definida como sendo um ponteiro para a função. A terceira instrução chama a função através da variável pointer usando o operador *indirection* entre parênteses. Como na declaração, você tem que usar os parênteses para gerar a operação indireta correta. É claro que, se *example()* tiver muitos parâmetros, você tem que incluí-los também na chamada da função.

void Pointer em C ++

Uma novidade em C++ é o *void pointer*. O tipo de dado *void* existe tanto em C quanto em C++ e especifica funções que não retornam um valor à função que chamou. No entanto, C++ permite-lhe também definir variáveis pointer para esse tipo. Embora você não possa referenciar indiretamente estas variáveis, pode atribuir a elas outras variáveis pointer. Na verdade, elas são mais úteis porque são compatíveis com todos os tipos de variáveis pointer e, portanto, podem servir como variáveis pointer genéricas em situações onde o programa deve manipular diversos tipos diferentes de dados. Então, você pode usar uma forma-modelo ou simplesmente atribuí-las de volta a variáveis dos tipos originais para restituir a acessibilidade.

Declare um *void pointer* do mesmo modo como você definiria qualquer outra variável pointer — *void *x* — e use-a numa instrução de atribuição do mesmo modo como usaria um ponteiro normal. Embora você não possa usar nela os operadores de ponteiro & e *, pode atribuir a ela outra variável *void pointer* ou qualquer outro tipo de ponteiro. Por exemplo, no seguinte trecho de programa:

```
int y = 123;
int *p = &y, *q;
void *vp;
vp = p;
q = vp;
```

o ponteiro *p* contém o endereço de um número inteiro, *y*, neste caso. O ponteiro *vp* aceita o valor desse endereço e passa-o depois para a terceira variável de ponteiro *q*.

O ponteiro do tipo *void* proporciona-lhe uma vantagem importante, ou seja, a capacidade de escrever programas mais genéricos. Agora você pode criar códigos que manipulam uma variedade de diferentes tipos de dados sem ter que esquematizar esta informação precocemente no processo de desenvolvimento.

Ponteiros e const

Outra diferença importante em C++ é a interação entre ponteiros e o modificador *const*. Esta relação proporciona possibilidades de programação que não são encontradas em C. Por exemplo, você pode definir um ponteiro para um objeto *const* para dar ao programa outro caminho de acesso a esse valor inalterável. De maneira mais atraente, você pode definir um ponteiro constante ou mesmo criar um ponteiro constante para uma constante. Cada um dos exemplos anteriores é uma interpretação única que você pode acrescentar ao seu repertório de ferramentas de programação. Vamos explorar mais estas combinações.

Para definir um ponteiro para um valor constante, simplesmente acrescente o modificador *const* à declaração. Por exemplo, em

```
const char *x
```

const modifica *char* mas não o operador *. A variável *x* aponta para um objeto do tipo *char*, que é constante quando acessado através desta variável particular, mas

que pode ser variável através de algum outro nome! Em const char *x = "Isto é apenas um teste"; a variável *x* é inicializada com o valor literal da cadeia de caracteres.

A declaração de um ponteiro constante tem uma sintaxe ligeiramente diferente e também um significado diferente. A declaração:

```
char *const x ="esta string esta congelada";
```

cria uma constante que contém o endereço da cadeia de caracteres especificada. Seu valor não pode mudar porque o próprio ponteiro é uma constante. A cadeia de caracteres, no entanto, não precisa ser uma constante.

Você pode também criar uma referência constante a um valor constante usando o modificador *const* duas vezes, da seguinte maneira:

```
const char *const x = "Isto também é um exemplo";
```

Este exemplo define uma constante *x* que aponta para outra constante. Ponteiros constantes são valiosos numa série de situações. Eles lhe proporcionam a conveniência de notação de ponteiro, enquanto protegem os valores do programa para que não sejam alterados inadvertidamente.

Peculiaridades de C++

Até aqui, este capítulo focalizou as várias semelhanças entre C e C++ para ajudar o programador iniciante em C++ a aprender a linguagem. No entanto, há muitos elementos que são exclusivos de C++, os quais não são encontrados na sintaxe C tradicional, mesmo numa forma modificada. Isso inclui um sistema de entrada/saída melhorado, sobrecarga de função, e, acima de tudo, o conceito central de C++: a classe. Vamos agora examinar estes aspectos singulares da linguagem.

Simples I/O em C++

A linguagem C++ não tem quaisquer operações I/O (entrada/saída) embutidas. Ela mantém o máximo de flexibilidade usando funções na biblioteca-padrão para suportar tais atividades. C++ foi idealizada para ser compatível com C ascedentemente, e, portanto, contém todas as funções I/O familiares, incluindo *getchar(), putchar(), scanf()* e *printf()*. Na verdade, um programa em C que dependa fortemente destes recursos compila tanto em C++ quanto num compilador C comum.

No entanto, C++ melhora muito a entrada e saída de valores. Essa melhoria inclui um grande aperfeiçoamento em conveniência e uma interface mais direta com o meio exterior, eliminando a necessidade de usar grandes funções tais como *scanf()* e *printf()*, que trazem consigo muita bagagem extra. A natureza objeto-orientada da linguagem C++ levou à criação de rotinas destinadas a tomar valores do teclado e colocá-los na tela de vídeo, as quais são tão genéricas quanto suas correspondentes em C, e ainda são muito mais eficientes e simples de usar. Os detalhes da implementação deste subsistema estão num capítulo posterior, mas, basicamente, C++ permite-lhe o acesso a uma série de pequenas rotinas de entrada/saída de uso geral que podem ser adaptadas aos valores específicos que você precisa usar. Além disso, sua conveniência compete com as rotinas *printf()* e *scanf()* porque o sistema automaticamente escolhe a rotina adequada para a operação desejada.

As definições básicas de I/O estão no arquivo do cabeçalho-padrão *stream.h*, que você deve incluir em qualquer programa que usa as rotinas. Este arquivo do cabeçalho define uma interface para o subsistema de I/O que consiste em três arquivos-padrão — *cin, cout* e *cerr*. Quando um programa começa a ser executado, ele automaticamente abre estes arquivos e inicialmente conecta o primeiro ao teclado e os outros à tela de vídeo, embora todos possam ser redirecionados como qualquer um dos arquivos-padrão em C. Acoplada a estes arquivos está uma série de operadores I/O que são realmente únicos para C++. Dois operadores conectam os valores de saída especificados com o arquivo apropriado: << envia valores para *cout* ou *cerr* e >> recupera os valores de *cin*.

Em ambos os casos, o operador age como um mediador entre uma variável ou expressão e o mundo exterior. O trecho de programa

```
#include <stream.h>

main()
{
char x;
cin>>x;
  .
  .
}
```

lê o valor de um caractere do teclado e o coloca na variável *x*. O trecho a seguir:

```
#include <stream.h>

main()
{
char x = 'a';
cout <<x;
  .
  .
}
```

executa a operação complementar, ou seja, toma o valor de *x* e mostra-o na tela.

Há uma vantagem dupla em usar-se operadores em lugar da sintaxe de chamada de função encontrada em C. Primeiro, esta notação é mais natural para ser escrita e, portanto, mais fácil de ler. Os sinais duplos "maior que" levam você a ver a direção do fluxo de caracteres. Isso não acontece com a chamada de função ao estilo da linguagem C. E, o que é mais importante, os operadores dão às instruções de I/O um estilo mais fluido, possibilitando que sejam combinados em expressões I/O complexas. Assim, C++ oferece dois novos operadores com a mesma economia de linguagem e superioridade como o operador incremento (++), e torna-os parte da notação I/O.

Estes operadores especializados são uma parte integral da filosofia de programação C++. Os capítulos posteriores mostram como C++ dá ao programador uma capacidade quase ilimitada para redefinir os símbolos de operadores disponíveis na linguagem. Você pode usar esta aptidão para criar códigos (programas) exatos e concisos. Como resultado, seu programa será também de mais fácil manutenção.

Outro aspecto importante destes operadores I/O é que eles fazem conversão de dados. Isso resulta de sua flexibilidade em usar apenas duas operações

para tirar e colocar dados. Os dois exemplos anteriores usaram declarações dos caracteres mais básicos, mas, na realidade, você pode recuperar ou exibir qualquer tipo de valor com plena confiança de que o programa fará a conversão apropriada. Considere as linhas de *io.c* (Listagem 1.5).

Listagem 1.5 Usando os operadores I/O: *io.c*

```
#include <stream.h>

main()
{
double x = 1.23, y;

cin >> y;

cout << "a resposta e " << x * y << "\n";
}
```

Este é, na realidade, um programa completo — se não interessante — que passa pela compilação. Ele define duas variáveis reais — uma delas com um valor inicial e outra que recebe um valor do teclado. Esta última operação é completada usando-se o operador de entrada >> com o arquivo *cin*. A implementação do operador executa a necessária conversão para uma *double*. Observe que, num exemplo anterior, este mesmo operador manipulava o valor de um caractere. A saída também ilustra um pouco mais a capacidade desse subsistema de entrada/saída. O operador de saída << permite-lhe agrupar vários valores e enviá-los à tela como parte de uma operação única. Os dados não precisam ser do mesmo tipo porque cada uso do operador representa uma chamada diferente do código do operador. Por exemplo, *io.c* mostra uma cadeia de caracteres, um valor *double* e uma cadeia de caracteres especial. A cadeia de caracteres especial é simplesmente uma nova linha de caracteres que você deverá reconhecer pela função *printf()* de C. (C++ reconhece todos caracteres de controle do carro da impressora.)

Esta rápida introdução deverá ajudá-lo a iniciar a escrita de programas C++ que se comunicam com o mundo exterior. No entanto, você pode fazer muito mais com *cin, cout, cerr*, << e >>. O Capítulo 7 contém uma discussão completa destes elementos C++.

Definição de Função em C

Embora o módulo principal (*main()*) em C++ seja uma *classe* a ser discutida, funções são ferramentas importantes para a organização do código-fonte de um programa. Funções permitem que o programador crie regiões locais dentro de um programa e depois, cuidadosamente, controle a entrada e saída do programa nestas regiões. Mais importante ainda é o fato de que o programador pode criar bibliotecas de rotinas compiladas de uso geral e depois usá-las em muitos programas diferentes.

A sintaxe da definição de função em C++ segue o modelo do novo padrão C ANSI. A definição de uma função começa com uma linha de cabeçalho que contém uma declaração do valor de retorno da função e uma lista de declarações para qualquer um dos parâmetros, como segue:

```
int example(int x, int y)
```

Nesta declaração, a função retorna um número inteiro e toma dois números inteiros — x e y — como argumentos. Note que as declarações dos parâmetros são feitas num único passo dentro de um dos pares de parênteses.

A declaração de parâmetros na linha do cabeçalho difere da linguagem C tradicional, que requer o uso de uma linha separada. No entanto, isso é apenas metade do novo modelo de sintaxe. Você pode também definir uma função e seus parâmetros ao mesmo tempo. O exemplo a seguir demonstra esta sintaxe:

```
main()
{
int example(int,int);
    .
    .
}
```

Em contraste com o cabeçalho de declaração, aqui você não precisa mencionar explicitamente o nome do parâmetro, mas somente o tipo de dado, embora possa incluir nomes de parâmetros se você quiser usá-los. Porém, C++ requer que você declare cada função no programa que a chama. Este procedimento difere da linguagem de programação C normal, que não declara explicitamente dados-objetos inteiros ou funções.

Sobrecarga de Função

Sobrecarga de função é uma parte única da sintaxe C++ que permite-lhe criar uma família de funções similares. Todas as funções da família que foi criada compartilham do mesmo nome, mas cada uma tem uma codificação independente. Cada função sobrecarregada deve ter pelo menos um parâmetro que difere das outras funções, ou seja, o tipo de dado de pelo menos um parâmetro deve ser único para cada versão da função. Quando a função é chamada, a lista de parâmetros que é passada para ela faz com o sistema procure o corpo de código adequado. Portanto, você não precisa acrescentar instruções ao programa que escolham a função correta. A sobrecarga de função também elimina a necessidade de inventar nomes estranhos para indicar as diferenças e semelhanças de várias funções relacionadas.

A palavra-chave *sobrecarga* declara operadores sobrecarregados. Use-a como parte da declaração de função, da seguinte maneira:

```
overload cube;
int cube (int);
double cube (double);
```

Este exemplo define um grupo de duas funções sobrecarregadas chamado *cube()*. Estas funções diferem tanto no valor que elas retornam quanto no tipo de dado de seu único parâmetro. É este parâmetro que determina qual *cube()* é usado numa chamada qualquer de função.

Antes de encerrar este assunto, devemos acrescentar um cuidado importante. Sendo a sobrecarga de funções um recurso útil e poderoso, pode haver abusos que conduzem facilmente a uma codificação enganosa e incorreta. Muitos programadores de C++ sentem que esta parte da linguagem não foi ainda tão bem documentada quanto deveria ser. De qualquer modo, a sobrecarga de funções deverá ser usada com parcimônia a cuidado, especialmente por aqueles programadores que ainda são principiantes na linguagem C++!

O Tipo de Dado class

O foco da linguagem de programação C++ é *class*, um tipo de dado que atende tanto ao projeto modular quanto às novas e excitantes noções de programação objeto-orientada. É o tipo de dado *class* que define a funcionalidade da linguagem de programação C++ e a qualifica como um progresso significativo sobre a linguagem C.

A vantagem apresentada por *class* é manifestada na sua forma mais básica. Considere a seguinte definição:

```
struct square {
   int side;
   int area();
}
```

Esta é uma definição de uma classe simples. Obviamente, a forma básica deste tipo de dado deriva da notação familiar de uma estrutura, ou de uma coleção heterogênea de variáveis contíguas. Semelhantemente à *estrutura* C, a classe C++ tem uma parte variável que armazena um valor. No entanto, diferentemente do tipo *struct* em C, a classe tem também uma função associada. No exemplo acima, a função que foi definida dá como resultado um número inteiro. Em resumo, esta é a diferença. Uma classe é completa: contém tanto a localização de armazenamento para conter os valores quanto as funções que manipulam esses valores. Esse formato é uma condição necessária para a criação de programas objeto-orientados.

Grande parte deste livro é dedicada a explorar as ramificações da declaração de classe. Ele mostra como criar e depois usar estas classes para resolver problemas de programação comuns, embora complexos. A classe em C++ não está limitada à sua primeira declaração básica. Sem dúvida, você pode usá-la para criar arbitrariamente objetos complexos. A partir do processo de exploração deste novo e excitante terreno, você descobrirá o lado prático deste conceito abstrato.

Classes Simples e Não Tão Simples

Até aqui, você viu apenas o tipo mais simples de definição de classe — uma declaração. No entanto, mesmo esta forma simples pode ser melhorada e expandida. Agora mesmo, vamos falar sobre implementação, tal como criar um objeto que reflita uma classe definida.

Vamos expandir a declaração anterior de classe numa completa definição de um novo tipo de dado. Esse é um assunto fácil, com uma interpretação assim tão simples. Considere a definição *struct.h* (Listagem 1.6).

Listagem 1.6 Uma definição de *struct* ampliada: *struct.h*

```
struct square {
    int side;
    void set(int);
    int area();
};

void square::set(int x)
{
    side = x;
}

int square::area()
{
    return side * side;
}
```

Uma definição completa deve incluir não apenas a declaração mas também uma definição das funções que são membros da classe. Neste caso, as únicas funções-membro são as funções *set()* e *area(a)*. A primeira inicializa a parte variável da classe, e a última dá como resultado a área de uma figura. Note que, neste contexto, o operador de resolução de abrangência (::) associa uma definição de função com o nome da função declarado na estrutura.

Uma vantagem importante da classe aparece quando você usa a definição na codificação do programa:

```
main()
{
```

```
    .
square x,y;
x.set(2);
cout << "area da figura x= "<< x.area <<"\n";
y.set(3);
cout << "area da figura y = "<< y.area <<"\n";
    .
    .
}
```

Cada objeto que você cria declarado como uma variável class (geralmente chamada de *particularização* da classe) é uma região independente do programa. Por exemplo, quando a função *set()* é invocada, ela altera somente os valores no objeto que a chama. Isso liberta o programador do receio de passar variáveis para frente e para trás para uma função. A função *pertence* à variável; ela é uma função *membro.*

Outra vantagem da classe é que ela elimina a necessidade de usar a notação desajeitada *struct square x* na declaração de uma variável. A linguagem C++ é suficientemente inteligente para encontrar a definição da variável de classe.

Classe como Objetos

A classe definida pela palavra-chave *struct* é apenas o mais básico dos objetos. Falta-lhe uma das vantagens mais importantes da classe — a privacidade. Um objeto mais completo usa a definição de classe como ilustrado em *class.h* (Listagem 1.7).

Listagem 1.7 Uma definição completa de classe: *class.h*

```
class precord {
    char *name,
         *id;
    long salary;
public:
    precord(char*,char*);
    void set_sal(long);
    char* display();
};
```

```
precord::precord(char* nm, char* i)
{
    name = new char[strlen(nm)+1];
    strcpy(name,nm);
    id = new char[strlen(i)+1];
    strcpy(id,i);
}
void precord::set_sal(long s);
{
    salary = s;
}
char* precord::display()
{
    temp = new char[strlen(name) + strlen(id) + 36];

    sprintf(temp,"nome = %s\nidentificacao = %s\nsalario = %ld\n",
        name,id,salary;

  return temp;
}
```

Este exemplo é ainda uma classe simples, mas, diferentemente dos exemplos anteriores, tem todos os elementos desse tipo de dados. A coisa mais importante a ressaltar é que ele está dividido em duas seções — uma parte "pública" que contém as funções-membro e uma parte "privada". No exemplo, a parte privada contém somente membros variáveis. Note que apenas os membros que estão declarados na parte "pública" da definição de classe são conhecidos pelas outras partes do programa; os membros da parte "privada" estão ocultos. Além disso, a única maneira de acessar os membros variáveis neste exemplo é através das funções-membro, que formam uma espécie de *ligação* para a classe.

Uma função-membro de uma classe pode servir como uma ligação porque ele tem acesso especial à parte privada da classe. A função-membro simples *set_sal()* pode tomar seu parâmetro e armazenar aquele valor diretamente na variável membro *salary* (salário). Os membros de uma parte pública de uma classe servem como ponte entre o resto do programa e a parte oculta da classe-objeto. A classe simples criada pela definição *struct* não tem esta parte privada; tudo nela é público. Os membros de um tal objeto estão livremente disponíveis para serem fixados ou não por qualquer parte do programa.

A parte privada de uma classe acrescenta uma nova dimensão de modularidade a um programa C++. Agora você pode ocultar para o resto do programa não apenas itens de dados mas também as funções que os servem. Assim, você pode criar programas que se aproximem melhor do ideal de uma série de objetos independentes — o objetivo final de um projeto objeto-orientado.

Embora a codificação neste exemplo seja direta e fácil de entender, ela contém outro conceito com o qual você deve tomar cuidado antes de passar a outros tópicos. A função *precord()* pode parecer um pouco estranha para o programador C tradicional. Isso porque *precord()* é uma função-membro especial chamada de *construtora.* Você pode solicitar a função construtora, principalmente através de uma definição, cada vez que criar um objeto desta classe. Ela representa uma espécie de rotina de inicialização na qual o projetista fornece o código para colocar os valores na classe. Deverá ser usada uma função complementar chamada de *destrutora* sempre que for destruída uma classe-objeto. Para a definição de uma classe não são necessários construtores nem destrutores.

Mudando o Significado dos Operadores através de Sobrecarga

Outra propriedade importante que se manifesta pela definição de classe é a habilidade em sobrecarregar os operadores. Esse conceito muito simples é usado freqüentemente em linguagens de programação; a linguagem C++ simplesmente amplia seu campo de ação. A idéia básica que há por trás do conceito de sobrecarga de operador é redefinir um símbolo comumente usado de maneira que ele se aplique a uma nova série de valores. Por exemplo, o operador + representa a operação de adição. Além disso, este simples operador pode ser usado com qualquer tipo de dados numéricos.

Você pode somar dois números inteiros, ou pode somar "doubles" ou "longs". (Em C++ você pode somar até dois caracteres, desde que esta operação tenha sentido no contexto do programa!) A instrução C que soma dois valores "doubles" é bastante diferente daquela que soma dois números inteiros. Assim, a linguagem C usa um símbolo + sobrecarregado com seus tipos de dados internos. Se você define um tipo de dado *class*, pode também sobrecarregar quaisquer dos operadores comuns de maneira que eles se apliquem à nova classe. Portanto, C++

oferece a mesma possibilidade de usar tipos de dados definidos pelo usuário que a linguagem C oferece com os tipos de dados internos.

O ponto-chave para a redefinição de um *operador* é associar a função operador com uma classe-objeto. (Uma restrição importante na sobrecarga de operador é que ela deve envolver um objeto do tipo *class*.) Você pode conseguir essa associação definindo uma função operador e passando para ela uma *classe* como parâmetro, embora o método mais simples seja transformar a própria função operador num membro da classe. Uma vez declarada, a função operador pode ser definida da mesma forma que qualquer outra função. Por exemplo, suponha que você tenha definido uma classe com três componentes — *x*, *y* e *z*. Além disso, você quer definir a soma de dois objetos deste tipo como a soma de suas partes correspondentes, isto é, são somados os valores *x*, depois os valores *y*, e assim por diante. Embora isso possa ser levado a cabo pela definição e pela função de adição, é mais conveniente usar um símbolo tradicional para a soma e expandi-lo de maneira a abranger esta nova definição. Isso é conseguido facilmente em *overload.h* (Listagem 1.8).

Listagem 1.8 Uma classe com um operador sobrecarregado: *overload.h*

```
class triad {
    long x, y, z;
public:
    void operator+(triad);
};

void triad::operator+(triad p)
{
    x += p.x;
    y += p.y;
    z += p.z;
}
```

Para maior clareza, este exemplo define apenas uma função operador. A função *operator+( )* redefine o operador +, mas somente no contexto dessa classe. Observe o formato da redefinição da função. Há uma função operador correspondente para cada um dos símbolos disponíveis em C++. Uma vez entendido aquele símbolo, a redefinição não é mais estranha do que escrever uma função para cuidar de seus detalhes. Esse exemplo fornece um objeto do tipo *triad* como parâmetro e muda o objeto que pede +. Esta seqüência de chamada seria a mesma se você estivesse somando dois números inteiros:

```
.
.
triad a,b;
.
.
a+b;
```

É esta semelhança com os usos tradicionais do símbolo que torna a sobrecarga de operador uma ferramenta conveniente para a programação.

Como acontece com a sobrecarga de função, a sobrecarga de operador está cheia de perigos em potencial. É um recurso do qual se abusa facilmente, principalmente através do seu uso excessivo. Os perigos são devidos ao fato de que é muito fácil você sobrecarregar um operador num contexto e depois tentar usar seu antigo significado em outro contexto. Considere sempre com cuidado a sobrecarga de operador.

Classes Derivadas

Outra característica importante da linguagem C++ é que ela possibilita criar hierarquias de classes-objeto. Em outras palavras, você pode definir uma classe A e uma classe B, de tal maneira que tudo o que está em A está também em B, embora B contenha alguns elementos adicionais. A vantagem disso é óbvia para qualquer pessoa que tenha necessitado resolver problemas de programação em bancos de dados ou em assuntos semelhantes, como, por exemplo, sistemas técnicos. No entanto, essa habilidade oferece vantagens mesmo para as tarefas mais insignificantes de programação. Em primeiro lugar, C++ permite-lhe criar uma classe e compilá-la. Mesmo que você distribua somente a forma-objeto, outro programador pode criar uma classe derivada baseada na sua definição original sem ter nenhum acesso ao seu código-fonte. Isso aumenta enormemente a flexibilidade da linguagem de programação.

O mecanismo para a criação de uma classe derivada é fácil de ser usado. Primeiramente, você cria uma classe comum C++, como, por exemplo, em *base.h* (Listagem 1.9).

Listagem 1.9 Uma classe base: *base.h*

```
class new_rec {
    char *name,
         *id;
public:
    new_rec(char*, char*);
    char* display();
};
```

Esta classe torna-se a sua *base*. Uma classe derivada deve especificar esta como a sua base. No entanto, você pode depois acrescentar qualquer coisa à classe derivada para suplementar esta base, como em *derived.h* (Listagem 1.10).

Listagem 1.10 Uma classe derivada: *derived.h*

```
class rec : new_rec {
    char *address,
         *city,
         *state;
public:
    new_rec(char*,char*,char*,char*,char*);
    char* show();
};
```

No cabeçalho da definição da classe na Listagem 1.10, a frase *:new_rec* assinala esta classe como derivada de outra anterior definida como *new_rec*. Embora o objeto *rec* que foi declarado seja também um objeto *new_rec*, há limitações ao acesso que o objeto *rec* tem sobre sua classe de origem. Especificamente, ele não pode acessar a parte privada de *new_rec*. No entanto, seus membros podem acessar as funções-membro da classe-base sem quaisquer outras referências indiretas.

Criando um programa C++

Agora que você já teve uma visão geral do que a linguagem C++ tem a oferecer, o restante do livro serve para ajudá-lo a dominar as técnicas práticas e as considerações envolvidas na programação C++. Em primeiro lugar, exa-

minaremos os mecanismos de escrita de programas C++ e como você passa do código-fonte para os arquivos executáveis.

A linguagem C++ está disponível numa variedade de implementações para UNIX, MS-DOS, e outros sistemas operacionais. Obviamente, o sistema operacional afeta circunstâncias tais quais, por exemplo, como são denominados os arquivos e que tipos de serviços de I/O se dispõem. Devido ao fato de C++ ser ainda relativamente nova, você talvez tenha que usar muitos programas da biblioteca fundamental do compilador C com o qual você compila a saída do tradutor C++. Assim, o compilador C de suporte de uma dada implementação C++ é também uma consideração importante. Finalmente, cada implementação difere em sua mecânica para criar, editar, compilar, unir e depurar arquivos-fonte. Algumas implementações usam interfaces à base de menus, enquanto outras dependem principalmente de arquivos batch (MS-DOS) ou telas (UNIX). Seu recurso final nestes assuntos é a sua documentação C++ resultante. A seguir estão algumas considerações gerais e lembretes úteis para criar programas C++.

O Arquivo-Fonte e as Convenções de Denominação

Todos os programas começam como arquivos-texto ASCII comuns chamados de arquivos-fonte. O texto destes arquivos consiste em instruções na linguagem C++ que são arranjadas numa ordem lógica para produzir programas úteis. Assim, você gera o primeiro estágio de um programa C++ com um editor de texto. O tipo de editor que você usa depende tanto do seu gosto como do seu ambiente de desenvolvimento de software.

As convenções de denominação dos vários tipos de arquivos em um programa C++ geralmente são determinadas pelo tipo de implementação que você usa. No sistema operacional UNIX, tanto os programas tradicionais C como C++ compartilham de uma convenção comum: ambos terminam com a extensão ".c". No entanto, outros ambientes de programação fazem distinção entre estes dois tipos de programas. As duas opções mais populares para extensões C++ são ".cpp" (Guidelines C++) e ".cxx" (Advantage C++ e Oasys C++). A única exigência na denominação de arquivos do tradutor C++ é a especificação de uma extensão. Além disso, a sua escolha de nomes de arquivos deverá seguir os costumes locais — 14 caracteres para UNIX e 8 caracteres para MS-DOS, para citar as duas opções

mais comuns. Este livro adota a convenção UNIX, usando a extensão .c para programas C e C++.

Desenvolvendo um Programa-Exemplo

Pelo fato desta seção focalizar mais o lado prático da produção de programa executáveis a partir de um arquivo-fonte C++, vamos desenvolver um programa-exemplo e trabalhar com ele. O programa *averager.cpp* (Listagem 1.11) ilustra um exemplo simples.

Listagem 1.11 Um programa que calcula a média de uma série de números: *averager.cpp*

```
#include <stream.h>

main(int argc, char *argv[]) // toma argumentos da linha de
                             // comando
{
 double total = 0; // estabelece uma variavel acumuladora

 for (int i = 1; i<argc; i++)
    total += atof(argv[i]);  // usa stdlib para converter os
                             // argumentos

 cout << "media = " << total / (argc-1);
}
```

O programa *averager.cpp* aceita uma série de números na linha de comando, soma-os e depois calcula sua média. A codificação é imediata e representa um programa C++ simples que usa a extensão de arquivo especificada para Guidelines C++. Note que, devido ao fato de *argc* ter sempre o valor mínimo de um, o programa não precisa verificar a divisão por zero. Na Listagem 1.12 está um programa idêntico, *averager.cxx*, para Advantage C++. Observe a extensão diferente do cabeçalho do arquivo bem como a nova extensão do nome do arquivo.

Listagem 1.12 Um programa que calcula a média de uma série de números: *averager.cpp*

```
#include <stream.hxx>

main(int argc, char *argv[]) // toma argumentos da linha de
                             // comando
{
```

```
  double total = 0; // estabelece uma variavel acumuladora

  for (int i=1; i<argc; i++)
    total += atof(argv[i]); // usa stdlib para converter os
                            // argumentos

  cout << "media = " << total / (argc-1);
}
```

Compilando o Programa

É importante compreender que a maior parte das implementações correntes de C++ são mais tradutores do que verdadeiros compiladores. Isto é, eles tomam um arquivo-fonte C++ e produzem um arquivo de codificação C que deve ser então introduzido num compilador C existente. Para os tradutores C++ licenciados pela AT&T, esta tradução passa por um processo de dois estágios:

- Um programa chamado de *cpre* toma o arquivo-fonte C++ e executa todos os comandos do pré-processador, produzindo assim um arquivo-fonte C++ intermediário.
- Um programa chamado de *cfront* aceita o arquivo intermediário e produz um código C legítimo.

Note que a saída de *cfront* deve ainda ser processada por um compilador C e um linker para produzir um programa executável.

A maioria dos projetistas de software não executam diretamente estes programas tradutores. Em vez disso, usam um programa chamado de *front-end* para chamar cada estágio por vez. Os formatos das ligações *front-end* das várias implementações C++ variam muito. A Figura 1.2 mostra um diagrama "genérico" dos passos envolvidos no desenvolvimento de um programa C++.

Em UNIX, você compila um programa C++ muito semelhantemente ao modo como compila um programa comum C. Para compilar o programa-exemplo, você executa a seguinte linha:

```
CC averager.c
```

Com isso se produz um arquivo executável chamado *a.out*. (Note que, em UNIX, o processador C front-end é *CC*, para diferenciá-lo do compilador UNIX C, *cc*.)

Na época da publicação deste livro, as implementações mais populares MS-DOS de C++ são Guidelines, Advantage e Oasys. Em Guidelines, você compila o programa-exemplo com o seguinte comando:

```
cppexe averager.cpp
```

Tanto Advantage quanto Oasys requerem o seguinte comando:

```
ccxx averager.cxx
```

para produzir um arquivo executável. Em ambos os casos, o compilador C automaticamente compila o código final produzido pelo tradutor C++. Os futuros desenvolvimentos provavelmente levarão a ambientes C++ integrados que são tão fáceis de usar quanto os produtos QuickC e TurboC são para os programadores C.

Os exemplos de compilação desta seção são apenas três entre as muitas variações possíveis, e envolvem somente o programa mais simples. Em situações reais você deve também configurar e controlar o compilador C e o linker, e manter arquivos *make* para projetos com múltiplos arquivos-fonte. Para maiores detalhes sobre estes assuntos, você deve consultar o manual da sua implementação C++ específica.

Questões de Revisão

1. O conceito principal que torna C++ diferente de C é _____ _____.
2. A sintaxe de declaração de função em C++ é semelhante ao padrão _____ _____, que está sendo usado cada vez mais em C.
3. O uso de várias definições para a mesma função ou o mesmo operador é chamado de _____.
4. Muitas das implementações C++ não produzem diretamente linguagem de máquina; ao contrário disso, elas produzem _____ que deve então ser colocado num _____.

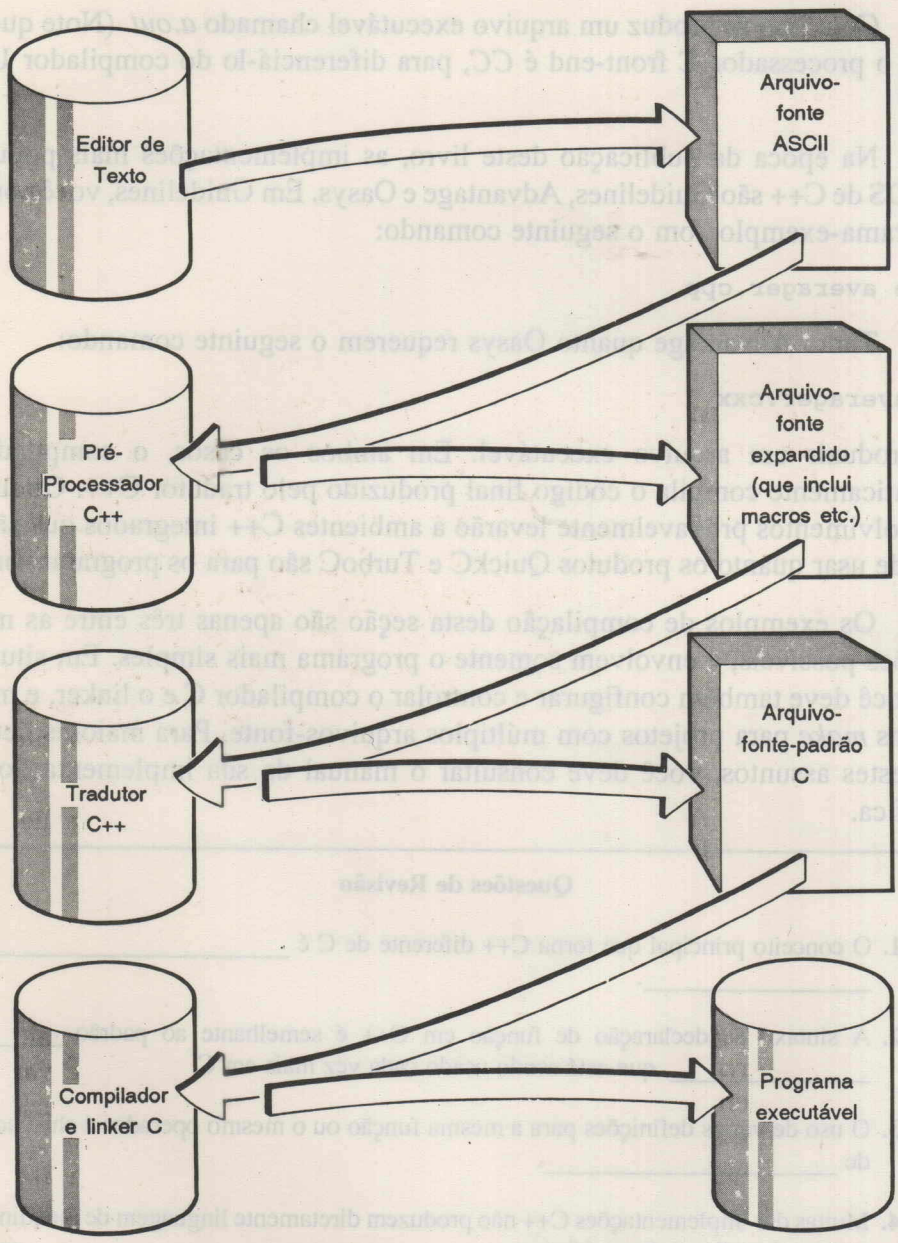


Figura 1.2 Desenvolvendo um programa C.

Problema de Programação

1. Usando os procedimentos adequados à sua implementação C++, digite e compile o programa *averager*.

Resumo

Este capítulo apresentou os aspectos principais de C++ e ofereceu uma variedade de exemplos de codificação C++. Você viu onde C++ é semelhante a C, onde ela expande alguns conceitos rotineiros de C, e onde se mantém em seu próprio campo para ser bem diferente de C. Embora as discussões e exemplos dêem apenas uma rápida idéia do poder desta linguagem de programação e os detalhes do seu uso, deverão ser suficientes para você começar a "pensar C++". Os capítulos que seguem oferecem discussões detalhadas sobre as características únicas de C++, incluindo sobrecarga de função, classes, sobrecarga de operador, classes derivadas e muitos exemplos práticos e completos.

Capítulo 2
Usando Funções em C++

As funções são o componente fundamental no projeto de um programa. Da mesma forma como acontece na linguagem C, as funções definidas pelo usuário são fundamentais em C++. No entanto, em C++, a sintaxe de função já familiar aos programadores C foi expandida para fornecer novas aptidões. Este capítulo mostra-lhe como declarar e definir funções e como usar os novos e potentes recursos que relacionamos a seguir:

- uma melhor interface entre funções
- valores assumidos opcionais para parâmetros de funções
- um parâmetro do tipo ponteiro de referência
- funções *inline*
- sobrecarga de nomes de funções (múltiplas definições)

- sobrecarga possibilidade de usar um número variável de argumentos de função

Se você já estiver familiarizado com o novo padrão ANSI proposta para C, reconhecerá algumas destas características de C++. Porém, C++ assume uma forma mais ampla e integrada para implementá-las. Todavia, o padrão proposto para a linguagem C está filosoficamente em sintonia com os princípios básicos de C++: ele ampara tanto o conceito geral de projeto estruturado quanto as características específicas de C++, tais como sobrecarga de função e modelagem de parâmetro.

Este capítulo também compara as definições de função C++ com as técnicas tradicionais da linguagem C. Embora seja importante compreender que C++ derivou de C, de certo modo ela é C, e é igualmente importante explorar as diferenças.

Sintaxe de Função Incrementada em C++

As funções são o ponto mais importante de qualquer programa C++. Na verdade, os programas são simplesmente grupos de funções; mesmo *main( )* — onde oficialmente começa a execução do programa — não é nada mais do que uma função. Portanto, o programador da linguagem C++ deverá ter sempre sua atenção focalizada para a criação e junção de funções. Funções construídas cuidadosamente levam à modularização de programas em subprogramas que são menores, legíveis e fáceis de depurar e dar manutenção. Além disso, a linguagem C++ é ainda mais adequada do que a linguagem C para reunir funções em bibliotecas que podem ser usadas muitas vezes em diferentes aplicações. Assim, o tempo que você gasta esquematizando e reesquematizando funções em C++ é um tempo bem empregado.

Declarações de Funções

Em C++ você declara *todas* as funções dentro da abrangência do programa no qual elas operam. Isso inclui funções que dão como resultado um número inteiro. Assim, a declaração:

```
int example( );
```

substitui a forma pré-ANSI *example( )*. Sem dúvida, você não apenas declara a função propriamente dita como também precisa incluir o tipo e o número de parâmetros na declaração. Em C++ uma declaração completa de função deve se parecer com alguma coisa assim:

```
int example(int);
```

O *int* dentro dos parênteses representa o tipo de dado do parâmetro que essa função utiliza — uma sintaxe essencialmente igual à modelagem de função no novo padrão ANSI C.

Quando uma função tem múltiplos parâmetros, você deve especificar o tipo de cada um separadamente. Por exemplo:

```
int example(int, char);
```

representa a declaração de uma função inteira que aceita dois parâmetros — um número inteiro e um caractere. Mesmo que todos os parâmetros tenham o mesmo tipo de dado, você deve especificar o tipo de cada argumento, da seguinte maneira:

```
int example(int,int,int);
```

Isso representa uma função que toma três parâmetros inteiros e dá como resposta um número inteiro. Note que você pode também incluir identificadores de parâmetro numa declaração:

```
int example(int x, char ch);
```

Tal inclusão não é necessária e não apresenta nenhuma vantagem particular.

Valores de Resposta de Função e o Tipo void

O primeiro item que você especifica numa declaração de função é o tipo de dado que a função dá como resultado. O resultado de uma função em C++ pode ser qualquer um dos seguintes tipos de dados: *char, short, int, long, float* ou *double*. Embora uma função não possa dar diretamente como resultado um vetor, ela pode dar uma estrutura ou uma classe. Naturalmente, um ponteiro para qualquer forma estruturada — mesmo um vetor — é sempre um resultado válido.

Em C++ (em ANSI C), o dado especial "*void*" indica explicitamente a *falta* de um resultado ou resposta. Você especifica valor de resultado *void* para uma função que é usada para efeitos secundários, como, por exemplo, uma função para escrever em um arquivo ou controlar um dispositivo periférico. Por exemplo, a função a seguir pode posicionar o cursor numa dada linha e coluna:

```
void cursor (int, int);
```

Em C++, diferentemente do que ocorre em C pré-ANSI, você deve usar explicitamente a declaração *void* para funções que não têm resposta. Se você não fizer isso, o compilador emite uma mensagem de aviso. Porém, implementações futuras podem tratar a falta de declaração *void* como uma condição real de erro. Outra razão para usar *void* como tipo de resposta é que ele permite-lhe usar o recurso da verificação da categoria em C++.

O que dizer das funções que não usam parâmetros? A prática da linguagem C tradicional utiliza parênteses vazios após o nome da função para indicar a falta de parâmetros.

ANSI C exige que você use o parâmetro *void*. Por exemplo,

```
example (void);
```

indica que *example* não usa parâmetros. Porém, a prática preferida em C++ é usar parênteses vazios:

```
example( )
```

Definições de Função

O formato da declaração de função que acabamos de ver é também o formato do cabeçalho para a definição de uma função. As únicas diferenças são: você precisa incluir os nomes dos parâmetros e você não usa o ponto-e-vírgula no final do cabeçalho. O esquema a seguir mostra um exemplo de um cabeçalho de definição de função:

```
int example(int x, int y)
{
  int z;
  .
  .
  return z;
}
```

(As linhas com um único ponto representam o código que implementa a função.) Observe o uso dos identificadores (*x* e *y* neste exemplo) para os parâmetros. Você não mais declara parâmetros em linhas separadas após o cabeçalho de definição. A nova sintaxe é mais compacta e fácil de ler.

Um Programa C++ Completo

O programa *calc.c* (Listagem 2.1) ilustra a sintaxe C++ em um programa simples que é uma calculadora de quatro operações. Para usar a calculadora, introduza uma expressão da seguinte forma no teclado:

<número><operador><número>

Esta introdução de dados é capturada pelo operador *cin* (veja mais detalhes no Capítulo 7) em três variáveis: uma variável numérica, uma variável caractere (para a operação) e outra variável numérica. Uma instrução chamada *switch* definida pela variável caractere seleciona a operação a ser executada, e o programa mostra o resultado na tela.

Listagem 2.1 Um programa que simula uma calculadora, para ilustrar o formato da declaração e definição de função: *calc.c*

```
#include <stream.h>

// define algumas substituicoes uteis

#define BLANK ' '
#define STOP 0

main()
{
  double x,y,
         radd(double,double), // declara uma funcao soma simples
         rsub(double,double), // ... uma funcao subtracao
         rmul(double,double), // ... multiplicacao ...
         rdiv(double,double); // ... e finalmente, uma divisao
   char opr = BLANK; // declara uma variavel para o operador

   while (opr != STOP) {     // continua ate o usuario dizer "exit"
      cout << "entre expressao "; // solicita digitacao ...
      cin >> x >> opr >> y;       // ... le ...

      switch (opr) {              // ... e, avalia
        case '+' : cout << "=" << radd(x,y); // cria uma condicao
                                             // para cada
                   break;                    // operacao
        case '-' : cout << "=" << rsub(x,y);
                   break;
        case '*' : cout << "=" << rmul(x,y);
                   break;
        case '/' : cout << "=" << rdiv(x,y);
                   break;
        case 'x' : opr = STOP;     // ... um pequeno lapso, "x" e a
                   break;          // segunda letra em "exit"
        default  : cout << "ainda nao implementado!\n";// otimismo
                                                      // exagerado!
      }
      cout << "/n/n";

      }
}
```

```
////////////////////////////////////////////////////////////////////

double radd(double a, double b)  // uma funcao para somar doubles
{
  return(a+b);
}

////////////////////////////////////////////////////////////////////

double rsub(double a, double b)  // ... uma para subtrai-los
{
  return a-b;
}

////////////////////////////////////////////////////////////////////

double rmul(double a, double b)  // ... multiplicar ...
{
  return a*b;
}

////////////////////////////////////////////////////////////////////

double rdiv(double a, double b)  // ... e, dividir
{
  if (b == 0)  // nao se esqueca de verificar divisao por zero
    return 0;
  return a/b;
}
```

O esboço do programa inclui uma função separada para cada operação possível. Na maioria dos casos, estas funções são estruturas de uma só linha, usando apenas uma instrução *return* para conter a expressão apropriada. (*rdiv( )* é uma exceção porque ela precisa verificar a divisão por zero antes de executar qualquer cálculo.) Este tipo de modularização se justifica aqui porque torna o programa mais legível. Em programas maiores, usando-se funções separada diminui-se o tempo de desenvolvimento e você terá mais espaço para expandir ou personalizar cada operação, por exemplo, para acomodar uma nova representação numérica como BCD (Decimal Codificado em Binário).

Observe que a função de cada operação é declarada dentro da função *main( )*. De maneira consistente com a sintaxe C++, são especificados tanto a forma da resposta quanto o tipo de cada parâmetro. Observe também que, em cada definição de função, a lista de parâmetros é colocada dentro dos parênteses do cabeçalho da função — e não por fora, como é comum na prática de programação C tradicional.

Vamos examinar mais atentamente este programa para ver como são utilizadas estas novas características. A seção que vem antes da função *main( )* executa três inicializações úteis: inclui as declarações de saída-padrão encontradas em *stream.h* e define dois identificadores: BLANK e *STOP*. Os identificadores melhoram a legibilidade da listagem. Note que o programa usa a sintaxe mais antiga do pré-processador — *# define* — e isso é adequado porque é necessária apenas substituição textual simples. [Mais adiante, este capítulo mostra como funções *inline* são melhores para definir macros (através de *define*) em muitas circunstâncias.]

A primeira parte da função *main( )* contém as declarações necessárias:

```
double x,y
   radd(double,double);
   rsub(double,double),
   rmul(double,double),
   rdiv(double,double),
char opr = BLANK:
```

Embora C++ não requeira mais que você faça todas as declarações no início de um bloco ou função, neste caso particular parece ser conveniente. Note que as funções que fazem o trabalho do programa — *radd( )*, *rsub( )*, *rmul( )* e *rdiv( )* — são declaradas no formato protótipo, que especifica tanto o resultado quanto os parâmetros que cada uma aceita.

O loop *while*, que controla o fluxo global do programa, define um loop simples do tipo "*entra-e-age*". Por meio dele, o programa emite um sinal de pronto, aceita valores do teclado, executa a operação apropriada, e mostra os resultados na tela. O programa usa também a flexibilidade do operador de entrada >> para concatenar três valores numa única linha de entrada. Assim, o usuário tem a ilusão de que está digitando uma única expressão.

A instrução *while*

```
while ( opr != STOP ) {
  cout << "enter expression";
  cin >> x >> opr >> y;
  .
  .
```

continua enquanto o operador numa expressão não for o caractere *STOP* definido no começo do programa.

O núcleo do loop *while* é a seguinte instrução *switch*:

```
switch (opr) {
 case '+' : cout << "=" << radd(x,y);
     break;
 case '-' : cout << "=" << rsub(x,y);
     break;
 case '*' : cout << "=" << rmul(x,y);
     break;
 case '/' : cout << "=" << rdiv(x,y);
     break;
 case 'x' : opr = STOP;
     break;
default : cout << "not yet implemented!\n";
}
cout << "\n\n";
```

Para cada operador válido é atribuída uma condição nesta função denominada *switch*. Na maioria dos casos, uma linha de código chama a função adequada e mostra o resultado. O programa utiliza também um artifício para desenvolver uma interface acessível ao usuário. Lembre-se de que, na realidade, um caractere é um número inteiro. O programa pode aceitar como entrada o valor *exit* porque a conversão implícita de forma é livre e sem restrições em C++. A letra *e* é convertida em um número real, a partir de sua forma como número inteiro. O mesmo acontece com a letra *i*. No entanto, o valor *x* está armazenado na variável caractere *opr*. Este valor então dispara a condição de saída para a função *switch*. Assim, o operador especial *STOP* na verdade não é necessário para se sair do loop; ele apenas ajuda a melhorar a legibilidade do programa.

Finalmente, as funções de serviço são quase idênticas, diferindo apenas na operação que são realizadas. Por exemplo, *radd( )*:

```
double radd(double a, double b)
{
   return(a + b);
}
```

aceita como parâmetro dois valores "double" e dá como resultado imediato sua soma. Somente *rdiv( )* difere significativamente:

```
double rdiv(double a, double b)
{
 if (b==0);
        return 0;
    return a/b;
 }
```

Observe a verificação de erro. O programa verifica *b* para ter certeza de que o seu valor não seja zero; este procedimento é necessário para qualquer expressão que envolva divisão.

Fornecendo Parâmetros com Valores Assumidos

Muitos programas contêm definições de funções de uso geral que usualmente trabalham como uma série de valores, mas ocasionalmente podem trabalhar com outra série. Por exemplo, uma função que eleva um número a uma potência pode ser definida como um subprograma de uso geral, mas muitas vezes será usada simplesmente para elevar um número ao quadrado. Em C, você pode executar estas diferentes operações de duas maneiras: pode definir uma série de funções especializadas para cuidar de cada ocorrência, ou pode explicitamente dar à função um parâmetro que indica a potência especificada. A primeira solução resulta num problema óbvio, ou seja, leva a uma proliferação de funções que pode ser muito pequena para um projeto eficiente. Estas funções apresentam mais recursos do que os realmente necessários para as instruções que são executadas. A segunda solução, que geralmente é a mais usada, leva a programas não muito legíveis devido à falta de um foco definido, principalmente se a função de uso geral tiver que servir a muitos casos especiais.

C++ oferece uma terceira alternativa. Em C++, você pode chamar uma função sem fornecer valores para alguns de seus parâmetros. Para isso, a definição da função fornece valores assumidos que são utilizados para estes parâmetros quando a função chamada não fornece um valor. Por exemplo, considere o seguinte trecho de programa:

```
int example(int, int=12),
x=23;
example(x);
```

O valor do segundo parâmetro será 12 (valor assumido especificado na declaração) porque a chamada da função *example( )* não especifica um valor para o segundo parâmetro. Se a função chamada fosse *example(x,4)*, então o segundo parâmetro teria sido 4.

O exemplo mostra claramente a sintaxe da especificação do argumento assumido. Na declaração de cada parâmetro que tenha um valor assumido, você deve colocar uma expressão da seguinte forma:

```
<tipo> = <valor>;
```

Você pode estabelecer desta maneira qualquer um ou todos os argumentos de uma função. Observe que a especificação assumida aparece na declaração da função *example( )* e não na definição propriamente dita. Na verdade, nesta definição não está claro que há um valor assumido definido; isso é evidente apenas quando você examina a declaração. O programa *calc2.c* (Listagem 2.2) é um exemplo de como definir uma função de uso geral, e como fornecer valores assumidos para os valores usados mais comumente.

Listagem 2.2 Programa de uma calculadora, com uma função que usa um parâmetro assumido *calc2.c*

```
#include <stream.h>

// define algumas substituicoes uteis

#define BLANK ' '
#define STOP 0

main()
{
  double x,y,
       radd(double,double), // declara uma funcao soma simples
       rsub(double,double), // ... uma funcao subtracao
       rmul(double,double), // ... multiplicacao ...
       rdiv(double,double), // ... e finalmente, uma divisao
       rpow(double,double=2); // cria uma operacao de potenciacao
                              // assumindo potencia de 2 - quadrado
  char opr = BLANK; // declara uma variavel para o operador
```

```
while (opr != STOP) {      // continua ate o usuario dizer "exit"
  cout << "entre expressao "; // solicita digitacao ...
  cin >> x >> opr >> y;      // ... le ...

  switch (opr) {             // ... e, avalia
    case '+' : cout << "=" << radd(x,y); // cria uma condicao
                                         // para cada
               break;                    // operacao
    case '-' : cout << "=" << rsub(x,y);
               break;
    case '*' : cout << "=" << rmul(x,y);
               break;
    case '/' : cout << "=" << rdiv(x,y);
               break;
    case '^' : if(y==2) // verifica se valor assumido (2)
               cout << "=" << rpow(x); // chama a potenciacao
                                       // com um parametro
               else
                cout << "=" << rpow(x,y);  // chama-a com ambos
               break;                     //especificados

    case 'x' : opr = STOP;      // ... um pequeno lapso, "x" e a
               break;           // segunda letra em "exit"
    default  : cout << "ainda nao implementado!\n"; // otimismo
                                                    //exagerado!

  }

  cout << "\n\n";

  }
}
////////////////////////////////////////////////////////////////

double radd(double a, double b)  // uma funcao para somar doubles
{
  return(a+b);
}
////////////////////////////////////////////////////////////////

double rsub(double a, double b)  // ... uma para subtrai-los
{
  return a-b;
```

```
}
////////////////////////////////////////////////////////////////////

double rmul(double a, double b)  // ... multiplicar ...
{
  return a*b;
}

////////////////////////////////////////////////////////////////////

double rdiv(double a, double b)  // ... e, dividir
{
  if (b == 0)  // nao se esqueca de verificar divisao por zero
    return 0;
  return a/b;
}

////////////////////////////////////////////////////////////////////

double rpow(double a, double e)  // uma funcao para potenciacao
{
  double t = a; // cria uma variavel para armazenamento temporario

  if (e == 0) // verifica o fato da potencia ser zero
    return 1;

  for (double i=1; i<e; i++) // um simples algoritmo para achar
    t *= a;                  // a potencia - isto nao funciona para
  return t;                  // qualquer caso
}
```

Este programa utiliza o programa anterior da calculadora e acrescenta um novo operador (^) que cuida da exponenciação. A função que manipula o operador — *rpow( )* — é declarada na mesma lista das outras funções. No entanto, a lista de parâmetros para a declaração de *rpow( )* não apenas lista cada parâmetro como também inclui um valor assumido para o último parâmetro. Examine a condição *switch* que controla a entrada desta função. Se a potência mencionada é dois, *rpow( )* é chamada com apenas um parâmetro, apesar de serem especificados dois na definição da função. Para o segundo parâmetro, é designado automaticamente o valor assumido de dois. Se você escolher um número diferente de dois, ambos os parâmetros são incluídos na chamada, e a função não usa o valor assumido.

Há algumas restrições sensatas quanto ao uso de argumentos assumidos. Os valores que você especifica deverão ser do mesmo tipo ou razoavelmente compatíveis com o parâmetro, quer dizer, a conversão implícita de um para outro deve ter sentido. Converter um valor *char* em um *int* faz sentido, porque *char* é, em essência, um pequeno número inteiro. Por outro lado, uma conversão de *double* para *int* comumente gera um valor incorreto.

Além disso, C++ não permite o uso de parâmetros assumidos no meio de uma lista de argumentos. Quando você chama uma função, pode omitir todos os argumentos (se a declaração especificar todos assumidos), mas não pode deixar de mencionar apenas os argumentos do meio e esperar que eles assumam os valores assumidos. Por exemplo, na seguinte declaração de função:

```
double example(int =123, char ='a', double =1.2);
```

você pode chamar a função com os parâmetros *int* e *char* (usando *double* como assumido); pode chamar a função somente com o parâmetro *int* (usando assumido para *char* e *double*); ou pode chamar a função sem nenhum parâmetro (usando valores assumidos para todos os parâmetros). No entanto, você não pode chamar *example( )* fornecendo somente os argumentos *int* e *double*, porque o pré-processador C++ não pode saber se você pretendia omitir o segundo ou o terceiro valor.

Questões de Revisão

1. Em C++ o costume de se declarar uma função com seus parâmetros na forma *example( )* é chamado de _____.

2. Você deve declarar o tipo de retorno de uma função. Portanto, você deve usar o tipo_____ para declarar que uma função que não dá um valor de retorno.

3. Se o programador fornece uma série de valores _____ como parâmetros de uma função, então se não for mencionado nenhum valor de parâmetro na chamada da função, estes valores serão usados.

Projetos de Programação

1. Amplie o programa *calc.c* para abranger operações que acham um resto e extraem raízes quadradas e cúbicas.

2. Acrescente uma operação de exibir para *calc.c*; ela só deve exibir valores, e não retorná-los.

Chamada por Referência

Outra característica C++ que torna mais versáteis as funções e as chamadas de funções é a possibilidade de "chamada por referência". Embora este termo possa ser estranho para muitos programadores C++ experientes, a ação resultante é utilizada todos os dias. Quando uma função associada a uma lista de parâmetros é chamada por outra função, devem ser dados valores às variáveis representadas pelos parâmetros. Ordinariamente, você faz isso chamando a função com uma série de variáveis, expressões, ou outros valores — um para cada parâmetro formal — embora a linguagem C++ também possibilite-lhe especificar um valor assumido que não precisa ser fornecido explicitamente. Geralmente, esses "argumentos formais" estão na forma de variáveis. Quando você usa a função a seguir:

```
x = example (x,y);
```

a variável *x* está associada com o primeiro parâmetro de *example( )* e *y* com o segundo.

Os parâmetros que são declarados na definição da função têm um campo de validade restrito à função. No exemplo a seguir:

```
int example (int x, int y)
{
   int z;
     .
     .
   return z;
}
```

x, *y* e *z* são variáveis locais. Usar uma variável na seqüência de chamada é o mesmo que inicializar uma variável na instrução de declaração dentro da função que está sendo chamada. Os valores passados para o parâmetro são "cópias" dos argumentos formais. Por exemplo, se você chama a função *example( )* da seguinte forma:

```
int p=2, q=4, w;
w= example (p,q);
```

o valor em *p* é copiado em *x* e, da mesma forma, *q* é copiado em *y*. Estas duas séries de variáveis estão relacionadas apenas no aspecto de que elas compartilham

dos mesmos valores. Os valores se movem numa só direção — da função que chama para a função chamada. Quaisquer alterações que ocorram nas variáveis na função não são passadas de volta para a função que chama. Essa sintaxe é conhecida como "chamada por valor".

Em C, a operação assumida é a "chamada por valor", mas você pode conseguir a "chamada por referência" usando um ponteiro como parâmetro. Numa chamada por referência, você passa a verdadeira variável (em outras palavras, o endereço da localização de memória) para a função. Quando você usa um ponteiro que representa um endereço, é enviado para a função o endereço do verdadeiro argumento, e não o seu valor. No exemplo que segue:

```
int example (int *x);
{
   int z;
      .
      .
   *x = 2*z;
}
```

uma alteração no parâmetro **x* tem um efeito permanente na função que o chama, porque ele muda o valor da variável *x*. Essa é uma maneira de recuperar valores de uma função sem usar a instrução *return*. Uma alternativa também comum é usar variáveis globais quando duas funções devem compartilhar dos mesmos dados.

Declarando Variáveis de Referência

C++ fornece uma forma de chamada por referência que é mais fácil de usar do que ponteiros. Primeiramente, vamos examinar o uso das variáveis de *referência* em C++. Da mesma forma como acontece em C, C++ permite-lhe declarar variáveis regulares ou variáveis de ponteiro. No primeiro caso, a memória é realmente alocada para o dado-objeto; no segundo, é reservada uma localização de memória para conter um endereço para um objeto que será armazenado posteriormente. C++ tem uma terceira espécie de declaração — *reference*. Semelhantemente a uma variável de ponteiro, ela faz referência a outra localização de variável, mas, do mesmo modo que numa variável regular, ela não necessita de operadores especiais.

A sintaxe da variável de referência é imediata:

```
int x;
int& y=x;
```

Este exemplo estabelece a variável de referência *y* e a designa para a variável existente *x*. Agora, a localização de memória tem associados a ela dois nomes — *x* e *y*. Como ambas as variáveis apontam para uma mesma localização de memória, elas são, na verdade, a mesma variável. Qualquer atribuição feita a *y* é refletida em *x*. O inverso também é verdadeiro, ou seja, mudanças em *x* ocorrem através de qualquer acesso a *y*. Portanto, com dados do tipo *referência,* você pode criar um pseudônimo para uma variável.

A variável do tipo referência tem uma restrição que serve para diferenciá-la da variável de ponteiro, que, afinal, faz algo semelhante. O valor da variável de referência deve ser estabelecido na declaração e não pode ser alterado enquanto o programa roda. Após você inicializar este tipo de variável na declaração, ela sempre se refere à mesma localização de memória. Assim, quaisquer atribuições que você faça a uma variável de referência alteram somente os dados na memória, e não o endereço da variável propriamente dita. Em outras palavras, você pode encarar uma variável de referência como sendo o ponteiro para uma localização de memória constante.

Chamada por Referência em C++

Por que usar referências? Considere a declaração de parâmetros de função. Usando uma variável tipo referência, numa declaração de parâmetro, você pode criar uma chamada por referência sem usar o mecanismo desajeitado do ponteiro, discutido anteriormente. A vantagem de usar uma variável de referência dentro de uma função é que a designação e manipulação de seus valores é imediata, não necessitando ser constantemente referenciada pelo operador ponteiro *, uma situação muito propícia a erros acidentais. O exemplo a seguir:

```
    .
    .
int value = 123;
example (value);
    .
```

```
    void example (int x&)
{
    x=2 * x;
}
```

dobra o valor corrente em *x* e devolve este valor para a função. Os parâmetros de referência eliminam a necessidade de declarar variáveis de ponteiro com sua sintaxe pertinente — **x* ou **rate* para executar uma chamada por referência.

O programa *mean.c* (Listagem 2.3) demonstra o tipo de dado chamado *reference* na lista de parâmetros de uma função. Este programa simples calcula a média de uma série qualquer de números reais. Ele é formado por três funções: *get_value( ), mean( )* e *main( )*. A função *get_value( )* recebe os valores do teclado e os acumula na variável accum. O parâmetro da função é definido na lista de parâmetros como sendo do tipo referência e é incrementado através do loop *for*. O valor especial *stop* encerra a operação desta função e retorna o controle à função que chamou. A função *mean( )* verifica a divisão por zero e então calcula o valor. A função *main( )* reúne as duas funções chamando cada uma e exibindo os resultados.

Listagem 2.3 Um programa que calcula a média de uma série de números e usa um parâmetro de referência: *mean.c*

```
#include <stream.h>
#include <string.h>

main()
{
  int temp;
  double get_value(int&),    // declara uma funcao de entrada
      mean(double,int),      // ... e uma para calcular a media
                             // aritmetica
      accum;
  accum = get_value(temp);   // obtem a entrada
  cout << "media = " << mean(accum,temp) << "\n"; // mostra o
                                                 // resultado
}

////////////////////////////////////////////////////////////////

double get_value(int& t)     // uma funcao para entrada dos dados
{
  char x[50];                // define um buffer de entrada
```

```
  double accum = 0;              // ... e uma variavel acumuladora

  cout << "entre valores abaixo:/n";

  for (t=0; ; t++) {             // inicia o loop que ira incrementar
                                 // um contador
    cin >> x;                    // preenche o buffer de entrada
    if (!strcmp(x,"stop")        // para ?
      break;
    accum += atof(x);    // converte o buffer em numero e acumula
    }
  return accum;
}
////////////////////////////////////////////////////////////////////
double mean(double x, int total)    // calculo da media
{
  if (total == 0) {              // verifica condicao de erro
    cout << "erro - divisao por zero!\n";
    return 0;
    }
  return x / total;          // retorna o valor calculado
}
```

As primeiras linhas de *mean.c* cuidam da necessária inicialização, incluindo os arquivos de cabeçalho *string.h* e *stream.h* a partir do diretório corrente. As declarações seguintes definem as funções que obtêm os números e calculam a média:

```
int temp;
double get_value (int&t),
mean (double,int),
accum;
```

O parâmetro *int&t* em *get_value( )* é um parâmetro de referência. Portanto, as alterações em *get_value* se refletem em *main( )*. A variável *accum* é uma variável real comum que armazena o total dos números introduzidos, e *temp* conta o número de valores introduzidos.

A parte fundamental de *main( )* consiste na chamada de duas funções. A primeira:

```
accum=get_value(temp);
```

recebe os valores numéricos do usuário através da função de entrada. A segunda:

```
cout << "mean=" << mean(accum, temp) << "\n";
```

mostra o resultado dos cálculos baseada nos dois valores de *mean( )*. Devido ao parâmetro *get_value* ter sido declarado como referencial, você não precisa usar o operador *address-of (&)*.

A definição de *get_val( )* resulta na declaração de referência:

```
double get_value(int&t)
```

Esta expressão declara a variável *t* equivalente a qualquer argumento formal que o solicita e passa todas as mudanças para a função solicitante. O restante do programa é imediato. Após as declarações, a função solicita ao usuário para que entre com os dados e utiliza um loop para tomar os valores do teclado:

```
for (t = 0;;t++){
cin >> x;
if(!strcmp(x,"stop"))
      break;
accum+= atof(x);
}
```

O programa trata os valores do teclado como se fossem cadeias de caracteres — através da instrução de entrada *cin* e do operador de entrada », porque constituem a forma mais conveniente de introduzir para testar. Aqui, a função testa apenas a cadeia especial *stop*, mas você poderia também fazer uma verificação de erro mais completa. Se o loop não encontrar a condição de saída, *get_value* usa a função da biblioteca, *atof( )*, para converter a cadeia de caracteres em seu valor numérico. Ela acrescenta este valor ao total corrente e envia o valor acumulado à função *main( )* através da instrução *return*. Como o contador *t* é uma variável de referência, todas as mudanças que ocorrem no loop também ocorrem na função de origem. Observe que esta função é muito mais legível do que um programa C convencional porque não requer ponteiro de referência indireta.

A função *mean( )* faz os cálculos baseada em dois valores fornecidos por *get_value*. Após testar a divisão por zero, ela executa uma divisão simples e dá o resultado.

Este exemplo é particularmente interessante devido à interação entre o parâmetro *t* em *get_value( )* e a variável *temp* que o solicita a partir da função

main( ). Devido a *t* ser um parâmetro variável, a seqüência de chamada inicializa-o com o valor de *temp*, ou seja, ambas as variáveis apontam para a mesma localização de memória. Quaisquer alterações em *t* — no caso, um incremento de um — são alterações permanentes.

Há várias razões para você usar uma variável de referência como um parâmetro, em lugar de uma variável de ponteiro. As designações são diretas e não precisam ser referenciadas por um operador indireto. Mais importante ainda é que a função solicitante não precisa usar o operador de endereçamento & juntamente com o parâmetro; ela usa diretamente o nome da variável. O uso de variáveis tipo *reference* tornam menores as possibilidades de erros, principalmente o erro de solicitar uma função com uma variável quando se deseja uma variável de ponteiro.

Funções Expandidas inline

Vamos mudar agora o foco de nossa discussão. Deixemos o assunto das melhorias nos mecanismos de chamada de função para tratar da questão se devemos absolutamente usar uma chamada de função. Na linguagem C tradicional, você pode usar macros para codificar um programa de maneira conveniente. Em qualquer lugar do programa onde aparece uma macro, o pré-processador substitui o nome da macro pelas instruções que ela contém. Esse é um recurso muito útil porque melhora a legibilidade e muitas vezes ajuda a evitar o excesso de chamadas de funções. O problema com o pré-processador macro é a maneira como ele ingenuamente interpreta uma definição macro — ele executa diretamente uma substituição textual. Por exemplo:

```
# define RATE 1.5
```

faz como que a cadeia de caracteres RATE seja substituída por 1.5. Isso funciona bem, mas o que acontece quando você usa uma instrução *# define* com parâmentros para prnduzir uma sintaxe semelhante à função? Essa macro é mais poderosa porque pode gerar código de programa para uma variedade de situações; mas pode também acarretar problemas. A definição a seguir:

```
# define tax(x) x*RATE;
```

funciona para algumas substituições, mas não para outras. Por exemplo, se você chama a macro com uma expressão em vez de uma variável simples:

```
tax(p+2);
```

a expansão da macro gera o seguinte código:

```
p+2*RATE
```

em vez do correto *(p+2)*RATE*. Os programadores C devem ter muito cuidado quando usam # *define* desta maneira. Outros problemas com a interpretação ressaltam o fato de que uma macro não é uma função: ela não tem variáveis locais; ela nem mesmo define um bloco. Os macros também não permitem verificação de parâmetro quando são usados modelos. Em resumo, uma macro # *define* é uma série de instruções mascaradas como se fossem uma função.

Funções inline em C++

A linguagem C++ compensa a desvantagem da macro C com um recurso chamado de *inline expanded function* (função inline expandida). A codificação expandida inline proporciona ao programador a oportunidade de usar um recurso semelhante à macro sem quaisquer dos problemas associados com a instrução # *define* do pré-processador.

O modificador *inline* permite-lhe assinar uma função em particular para ser expandida em lugar de ser compilada como uma função comum. Assim, toda vez que a função é solicitada, ela é substituída literalmente pelas instruções que se encontram na definição da função. (Na realidade, a expansão *inline* é meramente uma solicitação ao tradutor C++; ela pode ser ignorada se a sua função for muito complicada ou muito longa, e neste caso ela se torna uma função ordinária.) A grande vantagem oferecida por uma função *inline* é que ela mantém todos os atributos de uma verdadeira função: define um bloco; pode ter variáveis locais; e permite o mesmo tipo de verificação de erro que uma função comum. A criação de funções *inline* múltiplas também não apresenta problemas especiais.

Outra vantagem importante do uso das funções *inline* é que elas auxiliam a escrever programas modulares bem projetados que mantêm a eficiência ao nível de codificação. Para facilitar, a maioria dos programadores divide um programa

em funções pequenas e individuais. Quando são definidos muitos módulos, o programa geralmente é mais fácil de ser seguido e entendido e, portanto, é melhor projetado. O ideal é que cada módulo execute uma tarefa simples. Um programa bem estruturado é também mais fácil de ser atualizado ou modificado, porque são eliminados os efeitos secundários causados pelos dados-objeto com vasto campo de definição, e quaisquer erros criados pela nova codificação do programa podem ser facilmente isolados.

No entanto, as funções simples podem também criar um problema — podem ser ineficientes. Uma função que contenha uma única linha pode requerer mais tempo e recursos para ser definida do que realmente para rodar. Essa sobrecarga de trabalho inclui operações tais como memorizar o estado da função solicitante, copiar valores de parâmetros para a função, e solicitar a função. Recarregar a função original após executar a função chamada envolve o inverso desse procedimento. Assim, a sobrecarga da função toma seu tempo de execução. Geralmente, o projetista pode ignorar estes fatores, mas há ocasiões em que um programa precisa ter um alto desempenho, especialmente se ele tiver que executar muitas operações repetitivas. C++ permite-lhe eliminar este tipo de sobrecarga. Use simplesmente o modificador *inline* para declarar pequenas funções inline expandidas.

Definindo uma Função inline

Quando você usa o modificador *inline* como primeiro elemento da linha de definição de função, assim:

```
inline example(int x);
{
   .
   .
}
```

ele exige que o compilador coloque a função no programa sempre que este solicita a função. Nas chamadas de funções comuns, os valores são memorizados e o controle é passado para um subprograma. Com o modificador *inline* ativo, o código da função é copiado no programa cada vez que a função é solicitada. Na compilação, há tantas cópias quantas vezes a função é solicitada no programa, e

nenhuma delas fica sobrecarregada. No entanto, devido a codificação ser reproduzida várias vezes — aumentando assim o tamanho do programa — somente as funções muito pequenas podem beneficiar-se desta modificação.

A função *mean2.h* contém a função *mean( )* do programa exemplo anterior (Listagem 2.4) declarada *inline*. Nada mais mudou na definição. Na verdade, a função é um caso limite para a expansão *inline*, e é um tanto grande demais para se beneficiar deste tratamento. Carregar um arquivo com programa criado a partir de funções *inline* pode causar problemas.

Listagem 2.4 Uma função declarada como *inline*: *mean2.h*

```
inline double mean(double x, int total) // calculo da media
{
  if (total == 0) {                    // verifica condicao de erro
    cout << "erro - divisao por zero!\n";
    return 0;
    }
  return x / total;          // retorna o valor calculado
}
```

Questões de Revisão

1. Uma variável de referência contém o _____ de uma localização de memória e não o seu _____.
2. Uma variável de referência é usada geralmente como parâmetro de uma função para criar uma situação de chamada por _____.
3. Uma função declarada como _____ é semelhante ao comando pré-processador macro.

Projetos de Programação

1. Redefina *mean.c* de maneira que todas as funções sejam *inline* expandidas.
2. Refaça *mean.c* usando variáveis de ponteiro em lugar dos parâmetros de referência.

Criando Funções Versáteis com Sobrecarga

Ocorre freqüentemente que duas funções num programa executam a mesma operação, mas em tipos de dados diferentes. Por exemplo, uma função pode executar uma série de cálculos com números reais, enquanto outra só utiliza números inteiros. Os programas C usualmente fazem diferenciação entre estas funções com código especial. Por exemplo, no seguinte trecho de programa:

```
if(real_flag)
  x = f_std(z);
else
  i = i_std(z);
```

a função *f_std( )* executa cálculos com números reais, e a função *i_std( )* faz os mesmos cálculos com números inteiros. Você tem que criar uma função única e separada para cada tipo.

C++ proporciona uma solução melhor. Ela permite-lhe criar uma função única que sirva para ambos os propósitos. O recurso de sobrecarga manipula as diferenças.

Você deve utilizar a técnica de sobrecarga de função com muito cuidado. Quando estiver ainda começando o programa em C++, você deve se restringir a exemplos óbvios e bem escolhidos. Durante o processo de reutilização de um nome, você pode facilmente se perder e não saber qual versão está sendo utilizada num contexto específico. Para alguém que está lendo o programa, é ainda mais difícil entender estes diferentes contextos. Portanto, trabalhe com cuidado!

Declarando uma Função Sobrecarregada

Quando você declara o nome de uma função da seguinte forma:

```
overload example;
double example(double);
long example(int);
```

você alerta o compilador para o fato de que o programa contém mais do que uma série de códigos chamados pelo mesmo nome *example( )*. Neste caso, *example( )* dá como retorno *double* ou *long*, dependendo do tipo do parâmetro usado na sua chamada. Observe que, após a declaração *overload*, deve seguir uma declaração específica de cada forma. A única exigência para uma função sobrecarregada é que as listagens de parâmetros sejam diferentes pelo menos em um argumento. É assim que o compilador seleciona o código adequado para distinguir uma chamada de função. O número total de parâmetros envolvidos e o tipo de resultado, se houver algum, não são importantes, desde que pelo menos um parâmetro seja diferente em cada definição de função.

A exigência de "singularidade" para a lista de argumentos de uma função sobrecarregada inclui algumas restrições adicionais. Deve haver uma diferença clara e sem ambigüidades entre as listas de parâmetros de funções sobrecarregadas. Você não pode confiar sempre na flexibilidade de conversão de dados que a linguagem C++ faz implicitamente. Para solicitar uma função sobrecarregada, C++ primeiramente tenta comparar o tipo de argumentos com o tipo de parâmetros. Por exemplo:

```
int example(int,int)
```

é igual a

```
x = example(1,2);
```

Se C++ não encontra uma exata igualdade, ela aplica uma conversão interna, tentando reconciliar o argumento real com o argumento formal. No entanto, estas conversões nem sempre têm sentido. Por exemplo, *double* em *int* não pode ser feita pelo mecanismo de sobrecarga apesar de C++ geralmente permitir tais operações. Como último recurso, C++ tenta operações de coversão definidas pelo usuário para escolher a função apropriada. (Esse tipo de conversão é discutido juntamente com a sobrecarga de operador no assunto que trata de classes.)

O programa *mean2.c* (Listagem 2.5) é um exemplo de uma função sobrecarregada. Este programa permite que o usuário introduza uma série de números na linha de comando da seguinte forma:

```
mean 1.2 3.4 6.3 7
```

e dá como resposta a média desses números. Você pode usar números reais ou números inteiros. A resposta sempre corresponde aos números introduzidos — *double* para números reais e *long* para números inteiros.

Listagem 2.5 Um programa que calcula a média de uma série de números e usa a sobrecarga de função: *mean3.c*

```
#include <stream.h>
#include <string.h>

overload mean;          // declara "media" sobrecarregada
double mean(double, char*[], int); // declara uma versao double
long mean(long, char*[], int);     // ... e outra retornando long

main(int argc, char* argv[])   // prepara-se para obter os
                               // argumentos da linha de comando
{
  if (argc == 1) {       // verifica se foram introduzidos valores
    cout << Entre com os valores em uma linha de comando\n";
    exit(1);
    }

  if (strchr(argv[1],'.') == 0) {  // o primeiro numero e real ?
    long i = atol(argv[1]),        // chama mean() para long
    long x = mean(i,argv,argc);
    cout << "media = " << x << "\n";
    }
  else {                           // chama mean() para double
    double j = atof(argv[1]),
    double x = mean(j,argv,argc);
    cout << "media = " << x << "\n";
    }
}

////////////////////////////////////////////////////////////////

long mean(long x, char* buffer[], int len)  // a mean() para
                                            // valores long
{
  long temp = x;               // inicializa com o primeiro valor
  for (int i=2; i<len; i++)    // extrai todos os demais
    temp += atol(buffer[i]);
  return temp / (len - 1);     // retorna a media
}
```

```
//////////////////////////////////////////////////////////////
double mean(double x, char* buffer[], int len)  // a mean()
                                                // para doubles
{
  double temp = x;          // o valor inicial ja esta representado
  for (int i=2; i<len; i++)
    temp += atof(buffer[i]); // obtem o restante
  return temp / (len - 1);   // retorna o resultado
}
```

Após as instruções # *include*, o programa declara a natureza da função *mean( )*.

```
overload mean;
double mean(double,char*[], int);
long mean(long, char*], int);
```

As declarações das duas versões da função devem vir após a instrução *overload*. Obsereve que somente o primeiro parâmetro é diferente em cada uma; os parâmetros restantes são os mesmos. O mecanismo de sobrecarga usa esse primeiro parâmetro para decidir qual versão será usada durante o curso do programa. (O segundo parâmetro de cada função, *char*buffer[]*, não aparece usual; ele converte um vetor de ponteiros em cadeias de caracteres.)

Observe que, na função *main( )*, o código que manipula os argumentos passa da linha de comando para *argv*. Esta é uma convenção comum C que conduz a C++. Se o usuário rodar o programa sem fornecer argumentos adicionais, o programa imprime uma mensagem de erro e termina. Isso acontece da seguinte maneira:

```
if(argc ==1) {
 cout <<"Enter values on command line\n";
 exit(1);
}
```

(Lembre-se, um programa sempre tem um argumento — o próprio nome do programa. Assim, se *argc* é 1, o usuário não introduziu nenhum número na linha de comando.) Outro método de manipular a entrada incorreta seria reescrever a função de maneira que ela solicite do usuário os valores faltantes.

Na execução normal, o programa verifica se o primeiro argumento contém um ponto decimal; se contiver, é um número real. O primeiro argumento é

convertido em um número, e os argumentos restantes — ainda na forma de cadeia de caracteres — são destinados à função *mean( )*. O programa exibe a resposta. Se a função não detectar um ponto decimal, ela assume que o valor é um número inteiro e converte o primeiro caractere em um valor *long*. Este valor e o restante de *argv[]* são direcionados a *mean( )*; no entanto, é solicitada a segunda definição de *mean( )*, pois este é o valor *long*. Novamente, o programa exibe a resposta. As duas funções diferem apenas na maneira pela qual elas executam a conversão cadeia-para-número.

Vamos examinar mais detalhadamente a mecânica do programa. Uma vez que o programa confirme que a linha de comando tem alguns argumentos, ele deve decidir qual função solicitará. Primeiramente, ele converte o primeiro argumento no valor numérico apropriado. Isso é conseguido tomando-se o primeiro argumento na lista de parâmetros e usando-se a função de biblioteca *strchr( )* para verificar se há algum ponto decimal embutido na cadeia de dígitos. Um número real terá um ponto decimal e *struchr( )* testa isso. Se não for encontrado nenhum ponto decimal, o programa converte o primeiro valor em *long* através da função de biblioteca *atol( )*. Em caso contrário, o programa usa *atof( )* para converter o valor em *double*. Em ambos os casos o valor resultante vai para *mean( )*, e o valor calculado será exibido. Note que a variável *x* é declarada como variável automática e é inicializada através da função *mean( )*. Esta declaração restringe-se a um dos blocos internos da instrução *if*. Assim, o uso das mesmas variáveis ajuda a tornar a codificação mais simétrica.

Ambas as funções *mean( )* têm a codificação quase idêntica. Após a execução passar para o cabeçalho de definição:

```
long mean(long x, char* buffer[], int len)
```

ou

```
double mean(double x, char* buffer[], int len)
```

a função manipula os argumentos da linha de comando convertendo cada um em um número enquanto soma-os simultaneamente ao total corrente. A única diferença entre os dois cabeçalhos está no tipo de dados da função. O primeiro parâmetro faz a escolha da definição correta sem ambigüidade: se for um valor *double*, então solicita a função *mean( )* double; se for um valor *long*, solicita a função *mean( )* long.

Funções com um Número Variável de Parâmetros

A mesma flexibilidade que permite a sobrecarga de função permite também especificar um número variável de parâmetros. Este recurso é necessário porque os valores de parâmetro em uma função não podem ser estabelecidos até o momento de usá-la; algumas vezes, mesmo o número de parâmetros é desconhecido até este instante.

Um bom exemplo deste tipo de função é *printf( )*, que aparece em muitos programas C. Anteriormente ao padrão ANSI, você podia usar funções com um número variável de parâmetros em C, principalmente porque aquela linguagem não verificava parâmetros. Você poderia omitir os argumentos de uma função e, contanto que fornecesse o código para reconhecer esta situação na definição da função, o programa ainda rodaria. C também contém uma série de macros que ajudam a manter funções sem parâmetros — a série macro *varargs* — embora *printf( )* não seja implementada com elas.

No entanto, quando foi estabelecida a modelagem de função — como em ANSI C — esta solução simples tornou-se impossível. Agora, se você declara uma função e uma lista de argumentos subordinada a ela, esta declaração verifica os argumentos, evitando que você use variáveis a mais, ou que faltem alguns argumentos. Para maior segurança, foi descartada a flexibilidade.

No entanto, C++ oferece uma sintaxe fácil de entender para este tipo de chamada de função não-especificada — sem sacrificar os benefícios da verificação de parâmetros. Você precisa apenas usar reticências (...) para indicar um número variável de parâmetro.

Especificando um Número Variárvel de Parâmetros

Um caso típico da sintaxe da elipse (...) envolve a declaração de uma função na qual alguns parâmetros fixos são seguidos pelo especificador do parâmetro variável. Por exemplo, o trecho de programa a seguir:

```
int example(int,int,double ...)
```

especifica uma função que requer dois argumentos inteiros um *double* e um número desconhecido — no momento da compilação — de outros parâmetros. Estas é uma declaração adequada de uma função C++; entretanto, ela preserva a liberdade que os programadores C admitem como certa — a fluidez do parâmetro transitando. C++ tem também associada uma série de macros (encontrada no arquivo *stdarg.h*) que reúne quaisquer parâmetros reais que você possa destinar à função. Esta sintaxe permite a verificação total C++ dos parâmetros especificados.

O exemplo *prntvals.c* (listagem 2.6) é um programa simples que usa estas macros para solicitar uma função.

Listagem 2.6 Um programa para ilustrar uma função que usa um número variável de argumentos: *prntvals.c*

```
#include <stream.h>
#include <string.h>
// inclui os macros para extrair os parametros da funcao

#include <stdarg.h>

main()
{
  void print_many(int ...); // declara uma funcao com somente um
                            // conjunto de parametros

  print_many(3,1,2,3);      // chama-a com 3 argumentos
}
void print_many(int n ...)   // define a funcao
{
  va_list ap;          // chama a inicializacao das macros
  va_start(ap,n);
  for (int i=0; i<n; i++) {
    int temp = va_arg(ap,int);   // extrai cada parametro
    if (temp != 0)
      cout << temp << "\n";
    else
      break;
    }
    va_end(ap);      // chama a macro de encerramento
}
```

Neste exemplo, *main( )* declara uma função de *print many( )*, que toma um número variável de parâmetros. Note que o cabeçalho *stdarg.h* aparece no começo do arquivo do programa. Este arquivo contém as macros que auxiliam o uso de números variáveis de argumentos. Devido à função *print_many( )* tomar sempre um número inteiro como primeiro parâmetro, o programa passa à função primeiramente o argumento *3*, que indica o número de itens a serem impressos. Em seguida, o programa passa os valores a serem impressos — *1, 2* e *3*.

A primeira instrução em *print_many( )*

```
va_list ap;
```

declara o argumento list *ap*. A macro *va_start( )* usa esta lista de argumento como seu primeiro argumento, e o último parâmetro especificado antes da elipse, como seu segundo argumento (o inteiro *n* neste caso). Após estabelecer *va_list*, a macro *va_arg( )* separa cada argumento que é passado para a função. A macro:

```
va_arg(<va_list object>, <data type>);
```

dá como resultado uma variável do mesmo tipo que ela recebe como parâmetro. No exemplo, ela dá como resposta um número inteiro e atribui o valor à variável *temp*. Após *cout* exibir seu valor, a última linha da função solicita a macro *va_end( )* para "limpar" o objeto *val_list*.

Ponteiros e o Modificador const

Ponteiros sempre foram fundamentais na programação da linguagem C e conservam sua importância em C++. Embora C++ acrescenta o tipo de dado *reference* que pode dar valores como resposta (*chamada por referência*), você precisa ainda usar ponteiros para criar este tipo de situação de referência para cadeias de caracteres e outros vetores. Lembre-se de que um dos benefícios da chamada por referência é que o compilador não copia o argumento de uma função porque é passado somente o endereço. Com um grande conjunto de dados, isso pode representar uma grande economia de memória.

Outra possibilidade nova e vital é criada combinando o modificador *const* C++ com uma variável de ponteiro. Quando você usa o ponteiro constante

resultante numa declaração de parâmetro, você ganha a eficiência de memória proporcionada por uma chamada por referência, enquanto conserva a segurança de uma chamada por valor. Vamos examinar os detalhes desse procedimento.

Quando a maioria dos programadores idealizam uma função que usa um ou mais parâmetros na forma de cadeia de caracteres, eles geralmente as definem como ponteiros para um caractere. Por exemplo:

```
void example(char *s1, char *s2)
```

representa a linha do cabeçalho de uma função dessas. A única maneira diferente de declarar parâmetros na forma de cadeias de caracteres é declará-los como vetores de caracteres. No entanto, esse é um processo ineficiente — mesmo na linguagem C tradicional — porque o compilador simplesmente os converte em endereços de memória, ou, em outras palavras, em um ponteiro. Assim, a única maneira verdadeira de se manipular um parâmetro do tipo cadeia de caracteres é através de uma chamada por referência. Isso também é verdadeiro para outros conjuntos, porém, as cadeias de caracteres representam o caso mais comum.

Em C++, você pode criar aquilo que é efetivamente uma chamada por valor para um parâmetro do tipo cadeia de caracteres modificando a declaração do parâmetro com *const*. Este não é um verdadeiro parâmetro *chamada por valor* — pois não é local à função solicitante, e somente é transferido um ponteiro — mas, diferentemente de uma chamada por referência, o programa não pode mudar o valor da variável de parâmetro. Isso permite-lhe usar a sintaxe de ponteiro, sem ter que ter medo de modificar inadvertidamente aquilo que deve ser um valor constante.

A função *compare.c* (Listagem 2.7) demonstra uma comparação simples de duas cadeias de caracteres. Um resultado igual a zero indica que as cadeias são iguais; um número positivo indica a primeira posição na qual elas diferem; e um número negativo indica que a segunda cadeia é mais longa. (Esta função duplica a maior parte das características da função de biblioteca *strcmp( )*.)

Listagem 2.7 Uma função que compara duas cadeias de caracteres e usa parâmetros do tipo "const pointer": *compare.c*

```
#include <string.h>

int compare(const char *p1, const char *p2);
{
  int l1 = strlen(p1),      // observa o comprimento de cada cadeia
      l2 = strlen(p2);
```

```
  if (l1 == l2) {              // as cadeias sao de mesmo comprimento
    for (int cnt = 0; *p1 != '\0'; cnt++) {  // procura onde
                                             // elas diferem
      if (*p1++ != *p2++)
        return cnt;
      cnt++;                   // incrementa o contador
      }
      return 0;    // nenhuma diferenca detectada. Elas devem
                   // ser iguais
    }
  else if (l1 < l2) {
    for (int cnt = 0; *p1 != '\0'; cnt++) {    // procura por uma
                                               // diferenca
      if (*p1++ != *p2++)
        return cnt;
      cnt++;
      }
    return l1;   // o fim de l1 e a diferenca, porque ela e menor
    }
  else {
    for (int cnt = 0; *p2 != '\0'; cnt++) {  // procura mais
      if (*p2++ != *p1++)
        return -cnt;
      }
    return -l2;            // o fim de l2 marca a diferenca
    }
}
```

Esta função recebe dois parâmetros tipo cadeia de caracteres. No entanto, estes parâmetros são definidos como *const* porque os valores, nas cadeias de caracteres, nunca precisam ser alterados. A função apenas precisa examiná-los e fazer algumas comparações. Devido à função utilizar algumas funções de biblioteca-padrão, você deve usar a seguinte linha no começo do programa:

```
#include <string.h>
```

As instruções seguintes:

```
int l1 = strlen(p1),
l2 = strlen(p2);
```

armazenam os respectivos tamanhos de cada parâmetro, e isso prepara adequadamente os parâmetros para futuras comparações. Estas comparações ocupam o restante do programa.

Se as cadeias forem iguais, então um loop *for* simples verifica as posições dos caracteres que diferem nas duas cadeias. A variável *cnt* mantém a posição do caractere corrente, cujo valor retorna se os ponteiros tiverem valores diferentes em qualquer posição (indicando que existe uma diferença entre as cadeias). O teste termina quando o ponteiro do primeiro caractere alcança a marca *fim da cadeia '\0'*. Neste ponto, a função dá como resultado *0*, indicando que não há diferença. Observe que, neste exemplo, a impossibilidade de encontrar uma diferença indica igualdade.

Se a primeira cadeia for menor do que a segunda, a função continua a testar as cadeias à procura de diferença. A codificação para esta operação é idêntica àquela usada quando as cadeias são iguais. A única diferença é que, se a função chegar ao fim da primeira cadeia sem encontrar uma diferença, ela dá como resultado que a posição final é a primeira posição em que as duas cadeias diferem.

Finalmente, a função testa a situação na qual a segunda cadeia é maior do que a primeira. Aqui, o loop continua até o fim da segunda cadeia de caracteres. Isso é conseguido através do seguinte código:

```
else {
  for(int cnt = 0;*p2 ! = '\0'; cnt++){
  if (*p2++ != *p1++)
       return -cnt;
}
return -12;
```

Se uma diferença é encontrada, a função dá a resposta via *cnt*. Se o fim da cadeia é alcançado sem encontrar nenhuma diferença, a função dá como resposta o comprimento da cadeia e gera um valor negativo para indicar que esta posição se refere à segunda cadeia.

Questões de Revisão

1. Uma função _____ usa o mesmo nome para duas ou mais funções similares.

2. Uma função declarada como *example(int...)* indica uma função com um número _____ de parâmetros.

3. A declaração *const_char.p* produz uma variável de ponteiro _____ _____.

Projeto de Programação

1. Reescreva *calc.c* de maneira que ela funcione tanto para valores long quando doubles. Use sobrecarga de função.

2. Idealize uma função soma que possa somar qualquer número de valores. Declare a função com um número variável de parâmetros.

Resumo

Este capítulo explorou o assunto dos melhoramentos em C++ relacionados a funções. Alguns desses melhoramentos — não todos — encontram-se também no padrão ANSI C mais recente. No entanto, todos são compatíveis com a filosofia de programação C++. Estão incluídas neste grupo a modelagem de função e a especificação do parâmetro elipse. A primeira permite que o programador especifique não apenas o valor de retorno de uma função, mas também o número e o tipo de seus parâmetros. A segunda reconcilia a prática C tradicional de adiar a especificação de parâmetros até o instante de usar a função com a nova e rigorosa especificação por tipo. Os valores assumidos para parâmetros também foram outra melhora notada aqui.

O maior progresso discutido neste capítulo refere-se à sobrecarga de nome de função. Com esta sintaxe de definição de função melhorada, C++ permite que o projetista de software crie mais do que uma função com o mesmo nome. Um uso muito importante para este recurso surge quando a mesma série de operações deve ser executada em dados de diferentes tipos. Com uma função sobrecarregada o sistema automaticamente chama a função correta baseado nas diferenças em parâmetros que são transferidos ao nome da função.

O modificador *inline* é outro melhoramento importante na definição de função. Ele requer que o compilador tente substituir uma chamada de função pelo código real especificado pela definição de função. Esta é uma alternativa segura para o recurso macro criado pelo comando do pré-processador # *define*.

Capítulo 3
Definindo e Usando Classes

Representação dos dados
Modularidade e ocultamento de dados em C++
A mecânica da classe C++
Comparando soluções C e C++ para representação de dados
Classes completas em C++
Resumo

No último capítulo, discutimos como é organizado um programa C++ em funções separadas, e mostramos como C++ acrescenta nova versatilidade à definição e uso de funções. Agora, veremos como você pode combinar funções e declarações de dados em novas espécies de dados. Estas espécies de dados oferecem ao programador as ferramentes para criar dados-objeto internos que se aproximam mais estreitamente do comportamento dos objetos e sistemas do mundo real que o programa deve manipular. Não importa se o trabalho envolve física nuclear ou um relatório de vendas: a tarefa principal do projetista de software é criar maneiras cada vez mais precisas de representar o mundo real dentro da arquitetura do computador.

A linguagem C proporciona mecanismos básicos — arquivos de cabeçalho e funções compiladas separadamente — que ajudam os programadores a dividir um programa em partes ou módulos funcionalmente relacionados, ainda que sustentáveis separadamente. No entanto, o esboço do programa verdadeiramente modular requer mais do que isso. O programador precisa também controlar a forma como os módulos são acessados e determinar quais partes são internas (privadas)

e externas (públicas). Através do mecanismo da *classe*, C++ proporciona um novo grau de controle para especificar o relacionamento entre os vários objetos de um programa e o acesso a eles.

A *classe* C++ representa uma melhoria em C; ela combina representação flexível de dados com modularidade totalmente controlável, estebelecendo desta forma um novo conceito — o objeto — que exalta ambas as afirmações. Após uma discussão sobre as facilidades existentes em C, este capítulo apresenta a estrutura e projeto da classe em C++ focalizando, em primeiro lugar, os debates sobre representação de dados, e, depois, o assunto da modularidade.

Representação de Dados

A maioria dos programadores, quando pensam em espécies de dados, tende a pensar nas maneiras próprias como as linguagens C e C++ representam coisas como números inteiros, números em ponto flutuante e caracteres. No entanto, geralmente é necessário considerar objetos que consistem em muitas espécies diferentes de dados, embora relacionados. Um exemplo simples disso é um relatório de um cliente que pode conter caracteres (nome e endereço), números inteiros (inventários) e dados financeiros (cruzeiros). Embora o computador deva usar diferentes métodos para tratar destes tipos de dados básicos diferentes, o programador precisa de uma maneira de manipular o registro de dados como se fosse uma entidade única, como também os itens individuais.

As estruturas em C representam um primeiro passo em direção a esse alto nível de organização de dados. O recurso da definição de estrutura em C permite-lhe combinar muitas variáveis diferentes numa espécie de "super"-dado-objeto que tem dois aspectos. Por um lado, o objeto inteiro pode ser manipulado: você pode usá-lo como parâmetro para uma função, por exemplo. Por outro lado, você pode ter também acesso aos componentes individuais de tal estrutura de dados. Assim, a capacidade que as esturuturas têm de espelhar as entidades do complexo "mundo real" que elas representam torna-se uma parte indispensável do repertório de um programador C.

Porém, uma definição de estrutura em C refere-se somente a armazenamento de valores e uma estrutura de dados é simplesmente um grupo de variáveis. Para cada série de dados o programador deve providenciar uma série de ferramentas para manipular estes dados — por exemplo, para reservar memória para um novo registro, para colocar e retirar dados de um registro, para executar operações nos vários campos, e para liberar a memória quando o registro não for mais necessário. A linguagem tradicional C não tem nenhuma maneira fundamental de relacionar os dados e as ferramentas necessárias para trabalhá-los. O melhor que ela pode fazer é possibilitar a colocação tanto de estruturas quanto de funções relacionadas em um arquivo de cabeçalho ou em um módulo compilado separadamente.

A linguagem C++ toma o conceito de uma estrutura de dados e o expande de maneira a incluir não apenas os membros que armazenam valores, mas também as funções-membro que operam sobre aquelas variáveis. O resultado — um assunto completo chamado *classe* — é a base para que se possa chamar a linguagem C++ de linguagem de programação "objeto-orientada". Assim sendo, C++ pode ser vista como um avanço considerável para as linguagens "funcional" ou "processual" mais comumente usadas — como a linguagem C — na qual os dados e os algoritmos ficam estritamente separados.

O Problema da Representação de Dados em C

O problema da representação de dados é fundamental em todas as linguagens de programação. A maior parte dos esforços que se empregam no projeto de programa envolve a procura do formato ou representação adequados para os valores internos que o programa deve manipular. Algumas vezes esta representação é óbvia: por exemplo, um programa que faz análise estatística deve dar seus valores em números com ponto flutuante. Geralmente, porém, a escolha de um formato interno traz algumas surpresas. Por exemplo, você talvez decida usar números inteiros para representar importâncias em dinheiro em um programa, com o fim de evitar os erros de arredondamento associados com os números reais. Muitos problemas são tão complexos que o programador tem que definir uma espécie de dado de uso geral, como uma estrutura C, para combinar variáveis de diferentes tipos num conjunto coeso.

Vamos examinar o exemplo simplificado de uma aplicação típica de processamento de dados. Suponha que você tenha que escrever um programa de uso geral para manipular os dados financeiros de um cliente. Estes dados poderão estar em muitas formas diferentes — contas bancárias, ações, certificados de depósito, recibos e hipotecas. O problema torna-se ainda mais complicados porque os registros de informação não são homogêneos; alguns indivíduos têm mais dinheiro em cadernetas de poupança, outros investem mais em hipotecas, enquanto outros usam mais combinação diversificada de veículos financeiros. Como você representa no computador toda esta informação díspar?

Antes de tentar vários tipos diferentes de dados, pare e pense na informação que o programa deve manipular. Em primeiro lugar, você precisa escrever uma função que manipule todos os registros de um indivíduo — para talvez relatar-lhe o resultado líquido. Outras partes do programa, porém, devem tratar os vários componentes destes bens individualmente, de maneira que você não pode simplesmente jogar uma quantia numa simples variável. O dilema volta a ser a manutenção dos detalhes de um registro, enquanto se quer conservar a possibilidade de tratar o registro como um todo.

A solução-padrão da linguagem C é criar uma nova espécie de dados que contenha como componentes as partes individuais, ou campos, que formam a informação. Esta nova espécie é chamada de estrutura — *struct.* O exemplo *finance.h* (Listagem 3.1) mostra a definição de uma estrutura simples que pode conter os dados financeiros descritos anteriormente.

Listagem 3.1 Uma estrutura de registros financeiros em C: *finance.h*

```
struct precord {
  char id[9];    /* alguma maneira de identificar o proprietario */
  long savings, /* total das poupancas */
       checking, /* demanda de depositos */
       cd,       /* certificados de deposito */
       stocks,   /* o montante aplicado em acoes */
       bonds;    /* e titulos */
};
```

A definição da estrutura *precord* contém uma entrada (chamada de *membro* da estrutura) para cada categoria possível. Um membro de uma cadeia de caracteres pode conter instruções para identificar o indivíduo que possui o registro. Um membro contém o saldo numa conta bancária, outro contém o lucro total das

ações ao portador do indivíduo, e assim por diante. Para as situações mais complexas, você poderia expandir estes dados de maneira a abranger, por exemplo, os números dos certificados de depósito e até mesmo a contabilização dos cheques e poupanças.

Em C, uma vez que você define uma estrutura, você pode declarar uma variável daquele tipo, como mostramos no exemplo a seguir:

```
struct precord new_client;
```

Você acessa as partes individuais ou "membros" da variável da estrutura *new_client* através do nome da variável e do membro. Por exemplo, *new_client.id* acessa o membro *id*, *new_client.cd*, a variável do membro contendo *cd*, e assim por diante.

Um ponteiro para uma estrutura requer uma sintaxe especial, como mostramos no exemplo a seguir:

```
struct precord *p;
```

Com um ponteiro, também a notação de "flecha" refere-se às partes da esturutura para as quais estamos apontando. *p→cd*, por exemplo, refere-se ao membro *cd* da estrutura. Observe que ambos os exemplos conduzem à classe C++.

Agora, vamos usar a sintaxe da estrutura para manipular valores. Uma variável de estrutura é igual a qualquer outra variável: ela pode ter uma designação ou pode ser fornecida como um parâmetro. Além disso, semelhantemente a qualquer outro dado objeto, ela tem um escopo e uma classe de armazenamento. Ela pode ser global ou local; pode ser *automática, externa* ou *estática.* Para utilizar ou modificar seus valores, você tem que acessar a estrutura e seguir as regras de costume para variáveis em C. Infelizmente, os dois mecanismos proporcionados por C para acessar membros de estruturas (a notação "ponto" para membros individuais e a notação "flexa" para ponteiros) são um tanto desajeitados e geralmente dão origem a erros de programação.

Representando Dados com Objetos

Vamos desenvolver uma série de ferramentas de uso fácil para trabalhar com os valores em uma estrutura. Num caso ideal, estas ferramentas deverão acessar os valores individuais de uma maneira direta. Primeiramente, vamos examinar as características de um "objeto" interno do computador. De maneira mais clara, podemos dizer que ele é formado por duas partes:

- uma seção constituída por um valor ou um dado
- as operações que manipulam aqueles valores

O valor é necessário para manter a consistência de um uso do objeto para outro. Além disso, deve haver um esquema de armazenamento que guarda os resultados das transformações executadas sobre o objeto. Depois, deve haver as operações que executam estas transformações. Dados e operações, juntos, definem o objeto.

Esta definição de um objeto não deverá ser particularmente misteriosa. Embora você talvez não imagine, as variáveis ordinárias em C são formadas pela mesma estrutura bipartida. Um número inteiro, por exemplo, é armazenado como se fosse uma simples palavra de computador — o valor — e uma série de operadores como "x" e "*" — que definem como ele é usado. Vamos entender este tipo de construção em nível elementar, elevando-a até os mais abstratos assuntos de programação e aos objetos definidos pelo usuário, bem como às espécies de dados embutidos.

O exemplo *stack.c* (Listagem 3.2) mostra como você pode implementar um "objeto" em C.

Listagem 3.2 Uma espécie de dados *stack** implementada em um arquivo: *stack.c*

```
#define MAXSIZE 15

static stack[MAXSIZE], top;

push(x)
int x;
{
  if (top == MAXSIZE) {  /* verifica se a pilha estourou */
    err_rep(1);          /* imprime mensagem de erro */
```

```
    return 0;              /* nao empurrou */
    }
  stack[++top] = x;        /* estabelece a pilha */
  return 1;                /* empurrou */
}

pop(flag)
int *flag;
{
  if (top == 0) {          /* verifica se a pilha esta vazia */
    err_rep(2);            /* imprime mensagem de erro */
    *flag = 0;             /* envia de volta um sinal de erro */
    return 0;
    }
  return stack[top--];
}

static err_rep(n)
int n;
{
  static char *msg = {"stack overflow",
                      "stack underflow"}; /* lista das mensagens*/
                                          /* de erro */
  printf("%s\n",msg[n]);                  /* exibe a mensagem */
}
```

O exemplo *stack.c* consiste em um arquivo de definições de função que criam uma pilha (stack). O conceito-chave aqui é que o tipo dos dados é uma amálgama de funções e estratégias de armazenamento estático. Lembre-se de que uma pilha é uma estrutura de dados com um esquema de acesso do tipo "último que entra — primeiro que sai". O último valor que você coloca na pilha é o primeiro valor que você obtém de volta. Esse tipo de dado é definido pelas duas operações que são executadas nele: *push* e *pop*. *Push* é a operação que coloca um valor na pilha. *Pop* é a operação complemento, ou seja, tira um valor da pilha. Devido ao fato de as operações *push* e *pop* serem fundamentais à definição do tipo de dados, este é um exemplo ideal para se explicar a nova noção de um "objeto". Você pode facilmente implementar estas operações como se fossem funções, mas elas precisam ter uma estrutura de dados estática onde possam funcionar.

O exemplo *stack.c* contém uma solução C típica para este problema. Este arquivo define um vetor de dados chamado de *stack*, e define três funções para manipular este stack. A função *push()* acrescenta um número inteiro à pilha,

enquanto a função *pop* retira o último item. A função *err_rep()* imprime mensagem se a pilha apresenta estouro (overflow) ou esvazia (underflow). Juntos, vetor e funções proporcionam uma representação de dados e um grupo de ferramentas para manipular os dados.

Outro fator de projeto (que não é aparente na listagem) é um elemento fundamental à definição da pilha — estes itens são compilados separadamente num arquivo-objeto. Portanto, este arquivo binário é acessível a qualquer programa que necessite dos serviços desta *pilha* de dados. As regras de escopo restringem a "visibilidade" de váriaveis e funções, e criam um espaço, ou módulo separado, que contém a implementação desta espécie e controla estritamente o acesso a suas partes externas. O vetor *stack* e a variável *top* são definidos como *estáticas* para limitar sua visibilidade ao arquivo. Nenhuma instrução de fora — por exemplo, em outra parte do programa — pode manipular diretamente estas duas variáveis. O mesmo é válido para a função *err_rep()*; esta também é definida como estática e restrita ao arquivo. As funções *push()* e *pop()*, pelo contrário, representam a entrada e saída de valores deste módulo. Elas são as linhas de comunicação entre o módulo e o resto do programa e os únicos meios pelos quais os valores podem entrar ou sair.

A espécie de dados *stack* definida em *stack.c* é um bom exemplo de várias técnicas importantes de projeto dispensáveis para programador C. O ocultamento de dados é certamente um elemento-chave; outro é a noção de que uma espécie de dados pode ser mais do que uma estratégia estática para armazenar dados na memória — ela pode ser uma combinação dinâmica de valores e operações sobre aqueles valores. Esse exemplo mostra também, dramaticamente, as possibilidades inerentes em uma unidade, ou módulo, de um código que é autocontido. A modularidade é certamente uma técnica importante de projeto, e neste exemplo ela cresce a partir da solução de um problema em vez de ser imposta externamente como um princípio de organização.

Modularidade e Ocultamento de Dados em C

A vantagem mais evidente da modularidade é que ela dá ao programador a habilidade para lidar com grandes programas. É loucura criar um programa de

mil linhas sem dividi-los em partes menores e mais facilmente controladas; esse mínimo de sabedoria na programação é axiomático. Devido à função ser o módulo mais básico, a linguagem C apoia bem esse aspecto de modularidade. Você pode usar funções para implementar cada parte do seu projeto todo, terminando uma parte antes de prosseguir a próxima.

Arquivos como Módulos em C

C oferece algumas ferramentas individuais para apoiar o projeto de programação estruturada através da modularização. Manipulando as declarações de classe de armazenamento tanto de variáveis como de funções, e usando a capacidade C de fazer a compilação separada, você pode criar outro módulo — o arquivo — que é menos do que o programa inteiro, ainda que seja mais do que uma única função. A listagem anterior *stack.c* é um exemplo de um módulo de arquivo que contém múltiplas funções; é uma unidade de codificação mantida separadamente armazenada no seu próprio arquivo. Observe que as declarações controlam o acesso aos dados internos da pilha-objeto. Uma variável global declarada como estática (o vetor *stack*, por exemplo) tem um escopo restrito. Em vez de ser visível ao programa todo, seu escopo é restrito ao arquivo corrente. Assim, o arquivo torna-se uma "região" separada do resto do programa com seu acesso único através de funções e variáveis não estáticas.

O vetor *stack* e a variável inteira *pop* estão declaradas em *stack.c* como *estáticas.* Ambas são globais para as *push()* e *pop()* e, portanto, são acessíveis a partir destas funções. No entanto, devido a elas serem estáticas, seu escopo está restrito ao arquivo no qual elas estão definidas. Qualquer programa que use *stack* sabe que pode colocar valores e tirar valores da pilha (solicitando *push()* e *pop()* respectivamente), mas a implementação real está completamente oculta. Na verdade, um programa de usuário precisaria nada mais do que o arquivo compilado para usá-lo com eficácia.

Assim, a prática de espalhar definições de função entre muitos arquivos-fonte diferentes, compilando e depurando cada arquivo independentemente dos outros e combinando-os na fase de conexão, é bem do conhecimento dos programadores C. Os arquivos independentes combinam com as regras de escopo de C complicadas, mas bem articuladas para criar regiões dentro do programa que

mantenham suas próprias variáveis internas e funções bem como elos com o resto do programa. Estas funções e variáveis ligadas podem ser vistas como uma interface que permite o acesso a estas regiões, mas um acesso limitado e controlado pelo projetista. A compilação separada é baseada no fato de que uma variável global ou uma função que é declarada como *estática* não pode ser acessada externamente ao seu arquivo. A habilidade para criar estes objetos de acesso limitado é muitas vezes conhecida como *ocultação* ou *abstração de dados*, e é uma técnica útil para controlar a complexidade inerente nos projetos de programação no mundo real.

Módulos como Objetos

É fácil ver como os programadores usam ferramentas estruturais como as variáveis estáticas e funções estáticas para modularizar programas. O que pode não ser muito óbvio é que algo mais sutil ocorre quando você define estas regiões. A espécie de dado *stack*, por exemplo, é verdadeiramente um novo tipo adequado em tudo, menos na conveniência ao uso com os tipos tradicionais como *integer, long* ou *double*. Representando um valor e definindo operações específicas para manipulá-lo, você cria uma espécie de "objeto" que espelha a realidade que você está tentando implementar.

Mesmo quando os programadores conhecem a realidade toda do armazenamento e manipulação de números dentro de um computador, eles geralmente vêem uma conexão "natural" entre um número inteiro e uma variável *int*. Esta não é uma visão muito ingênua como a tentativa de controlar o grande número de detalhes pertinentes à programação. Os programadores não podem ter medo de deslocar bits à direita e à esquerda e movimentar valores na memória com a finalidade de somar dois números simples. O mesmo tipo de "controle de detalhe" é necessário para outros tipos de representação no computador, especialmente representações definidas pelo usuário como a definição de *struct*. Você precisa ver o mesmo tipo de conexão "natural" entre uma pilha e sua representação ou entre uma estrutura e a entidade da *vida real* que ela representa.

Os elementos-chave que definem um "objeto" são:

- uma estrutura-objeto.
- as operações que manipulam aquela estrutura.

Além disso, os detalhes da representação precisam estar ocultos ao usuário, que deverá saber somente que existe um tipo específico, ou, como no exemplo da pilha (Listagem 3.2), que há uma pilha que recebe ou devolve um valor. A situação ideal, inatingível em C, mas presente em C++, é a habilidade para declarar variáveis-objeto que, por sua vez, ocultam os detalhes de sua própria implementação.

Os primeiros passos em direção à implementação deste tipo de programação "objeto-orientada" foram delineados em *stack.c*. Este exemplo de modo algum é completo, mas mostra a direção certa. Assim, a criação de um objeto é simplesmente um processo em dois estágios que envolve:

- definir a implementação em termos de espécies de dados estáticos comuns e construção de programação C e
- "empacotar" esta implementação numa forma fácil de usar.

Você combina abstração de dados e modularidade para formar uma "unidade" que você pode ligar num programa sem dar nenhuma atenção à maneira como ela funciona. Se você escreve hoje um programa usando os objetos da pilha que acabamos de ver, você volta a ele na próxima semana, modifica-o — torna *stack* uma lista concatenada, por exemplo — e ainda usa o programa que você criou. Em termos práticos você teria que recompilar e limpar, mas não precisaria fazer mudanças reais no próprio programa. A medida deste "objeto" é sua credibilidade à definição bem estabelecida de uma pilha não ao método pelo qual você o criou. Os detalhes de implementação necessários para um objeto estão ainda no programa. No entanto, você precisa direcionar este esboço apenas uma vez — quando cria a definição de objeto; uma vez feito isso, você pode esquecer o projeto e usá-lo como se fosse uma parte interna da linguagem.

Este tipo de modelo, chamado de *esboce uma vez; use muitas vezes*, é exatamente o significado da programação modular. Cada objeto é um módulo e uma parte independente do programa. Se há uma diferença, está na seguinte ênfase: um dado-objeto é deliberadamente mais de uso geral do que um módulo precisa ser. Há outros aspectos da programação objeto-orientada que não são apontados aqui. Um dado-objeto é, em essência, um módulo, mas ele é também muito mais. O aspecto da comunicação entre objetos é crítico à definição, assim como é crítica a relação entre objetos chamada de *herança*. Estes dois itens são discutidos no Capítulo 4.

Embora uma pilha seja uma coisa intangível na vida real, ela ainda é diferente da representação criada no programa. As *operações* dessa entidade, no entanto, são espelhadas pela representação do computador. (Veja a Figura 3.1, que dá uma ilustração dessa relação.)

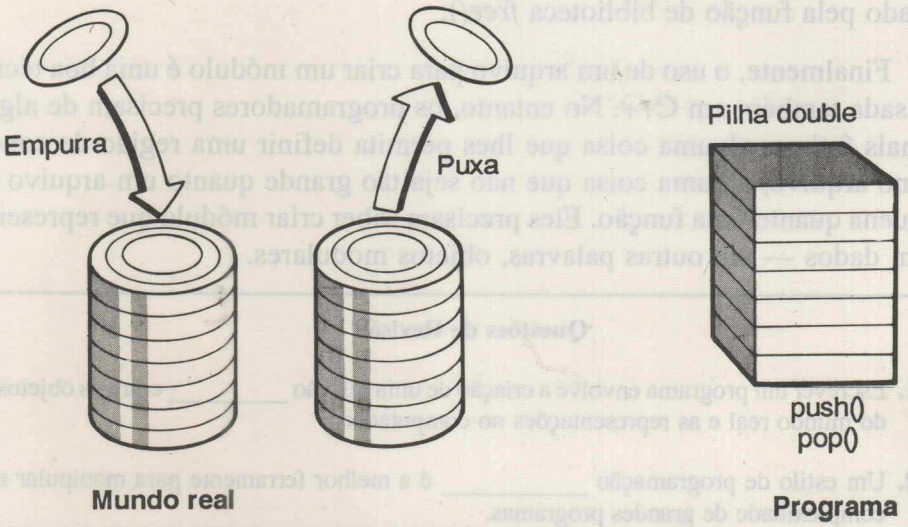


Figura 3.1 A relação entre o computador e as representações no mundo real.

A espécie de abstração mantida por C é certamente uma técnica útil e eficaz, mas não é isenta de problemas. Um dos problemas já foi apontado: as novas espécies de dados e outras formas de abstração de dados que você cria dentro de um programa nunca têm o mesmo status dos tipos internos. Por exemplo, nunca é tão fácil declarar uma variável *struct* quanto é declarar uma cadeia de caracteres. E, neste caso, conveniência significa eficiência, e, o que é mais importante, clareza. Clareza, por sua vez, significa menos enganos cometidos inadvertidamente e menos erros (*bugs*).

Esse ponto de conveniência parece ser uma operação complicada, mas não é. Com espécie de dados internos você pode fazer muitas coisas que com seus próprios projetos não poderia. Por exemplo, quando você cria uma variável inteira automática, tão logo o programa sai do seu escopo — talvez retornando da função — o computador toma de volta aquele espaço de memória. Em contraste, considere o que acontece quando você define uma estrutura e os ponteiros que a ela se referem. O que acontece com os ponteiros quando a estrutura para a qual eles apontam sai do escopo? A situação

é ambígua e o programador precisa ter cuidado com esta arrumação. Na verdade, a situação é ambígua exatamente quando a variável é declarada. Aqueles campos que são ponteiros requerem espaço alocado explicitamente, que é retido não apenas até que o programa saia do escopo da variável, mas, sim, até que seja explicitamente desalojado pela função de biblioteca *free()*.

Finalmente, o uso de um arquivo para criar um módulo é uma boa técnica, que é usada também em C++. No entanto, os programadores precisam de alguma coisa mais íntima, alguma coisa que lhes permita definir uma região de controle menor no arquivo, alguma coisa que não seja tão grande quanto um arquivo nem tão pequena quanto uma função. Eles precisam saber criar módulo que representem e alojem dados — em outras palavras, objetos modulares.

Questões de Revisão

1. Escrever um programa envolve a criação de uma relação _____ entre os objetos do mundo real e as representações no computador.

2. Um estilo de programação _____ é a melhor ferramente para manipular a complexidade de grandes programas.

3. As duas ferramentas principais para modularização em linguagem de programação C são _____ e _____.

4. Para usar uma nova espécie de dados você precisa criar uma maneira de armazenar _____ e _____ que funcione com elas.

Problemas de Programação

1. Refaça as definições de estruturas da Listagem 3.1 de maneira que elas fiquem completas e realistas.

2. Crie uma série de definições semelhantes para uma estrutura de dados simples, como por exemplo, estatística para um jogador de beisebol.

A Mecânica da Classe C++

A classe em C++ oferece maior modularidade do que os arquivos C compilados separadamente quando usam as propriedades de representação de dados da estrutura C. No entanto, em C++ esta modularidade é apenas a vantagem inicial. Você define uma classe de maneira muito semelhante àquela pela qual define uma estrutura em C. Em C++ porém, a classe não contém apenas dados-membro, possui também membros funcionais. O exemplo *circle.h* (Listagem 3.3) mostra a forma mais simples de uma classe.

Listagem 3.3 Uma classe C++ simples: *circle.h*

```
const double pi=3.1415;     // define pi com a precisao desejada
struct circle {
  double rad;               // armazena o raio de um circulo

  void set_radius(double);// declara uma funcao para inicializar
                          // uma classe
  double get_area();  // da como resultado a area com o raio atual
  double get_circum();// ... e a circunferencia
  };

void circle::set_radius(double r)
{
  rad = r;              // coloca o raio como valor do parametro
}

double circle::get_area()
{
  return pi * rad * rad;     // da o resultado do calculo usual
}

double circle::get_circum()
{
  return pi * 2 * rad;       // da o resultado da circunferencia
}
```

Devido à classe *circle.h* referir-se a uma figura geométrica, pi é definido como uma constante *double* com o valor apropriado. A palavra-chave *struct* indica que a definição que segue é a classe, denominada pelo *tag field* na mesma linha.

Neste caso *tag name* (rótulo) é *circle*. A variável *rad*, que é *double*, é uma variável estática comum que contém o valor do raio do círculo.

A definição de classe C++ é diferente da estrutura C porque contém também funções-membro. Em *circle.h*, estes membros são:

```
void set_radius();
double get_area();
double get_circum();
```

A primeira função define a variável *rad*, que representa o raio. A segunda função-membro dá como resultado o valor da área do círculo, que é calculada a partir do valor atual da variável *rad*. Finalmente, a última função-membro dá como resultado a circunferência. A definição destes membros é direta. Em cada caso, elas são funções de uma única linha que executa um cálculo e dá um resultado.

Observe que a relação evolutiva entre a estrutura C e a classe C++ é explícita. A definição mais simples de classe usa a palavra-chave "struct". No entanto, você deve ter cuidado ao usá-la; a classe definida por *struct* em C++ é muito diferente da estrutura C mais primitiva. Não apenas uma classe contém membros *funcionais* (uma estrutura C tem como membros apenas variáveis estáticas), estas funções-membro podem acessar diretamente a parte variável da classe, ainda que não tenha sido explicitamente enviado a elas como parâmetro. Este recurso é explorado em *set_rad()* para permitir que seja designado um parâmetro ao membro *rad*. As próprias definições de funções devem incluir uma referência à sua classe. Você consegue isso usando o operador de escopo (::).

A simples definição de uma classe não cria quaisquer objetos; você precisa ainda declarar algumas variáveis de classe. Por exemplo, a definição corrente:

```
circle x;
```

realiza esta tarefa. Observe que você não precisa usar a palavra-chave *struct* na declaração. Você acessa os membros da classe com a notação do ponto. Isso é imediato quando você acessa membros variáveis; por exemplo, *x.rad*, usa a mesma forma que na estrutura C. Entretanto, no início você pode achar estranho usar esta notação com funções. Por exemplo:

```
x.set_rad(1.23)
```

inicializa *circler x* com um raio de 1.23. Da mesma forma, *x.get_area()* e *x.get_circum()* dão como resposta os valores apropriados. É importante lembrar

que estas funções não são menos membros da classe do que o membro *rad*, e, como membros companheiros, têm uma inter-relação especial — cada um deles sabe e pode acessar o dado-membro implicitamente. Além do mais, o escopo da classe define uma região autônoma no programa que inclui todos os membros. O único meio de entrar nesta região é através da notação do ponto.

Do mesmo modo que acontece com as estruturas, você pode também declarar uma variável de ponteiro para uma classe, da seguinte maneira:

```
circle *p;
```

A notação da flecha manipula qualquer membro, como faria numa estrutura C. Assim, *p->rad*, *p->set_rad()*, *p->get_area()* e *p->get circum()* são expressões válidas.

Esta rápida visão geral da sintaxe básica da classe C++ permite-lhe usar toda a sua força, mas não a explica completamente. O restante deste capítulo e o próximo discutem mais as propriedades e limitações deste recurso de programação.

Comparando as Soluções C e C++ para Representações de Dados

Um problema que todo programador encontra sempre é como representar o tempo. O problema principal quando se trata de tempo é a conversão da hora do dia (hora e minutos) num intervalo de tempo. Geralmente você têm que responder à pergunta: "Quanto tempo se passou desde que o evento A ocorreu?" Mesmo com a simples pergunta: "Que horas são?", você tem que considerar dois sistemas: sistema AM/PM (manhã/tarde) e sistema de 24 horas. As seções que seguem mostram duas soluções para o problema de representar o tempo — uma em C e outra em C++. Para comparar estes dois sistemas de programação, a discussão também mostra como converter uma representação em outra, e isso aponta claramente as diferenças e semelhanças nestas linguagens.

Em C, você geralmente manipula a disparidade encontrada nas noções do dia a dia, tais como o tempo, criando aquilo que se congrega numa "família" de

tipos relacionados. Uma boa estratégia, nesta linguagem de programação tradicional, é encontrar algum sistema simples e fácil de representação de dados para servir como uma base. No caso do tempo, uma base natural é o número de segundos a partir da meia-noite. É adequada porque sempre gera um número inteiro positivo, e, portanto, o programa pode facilmente executar comparações de intervalos de tempo. Após determinar uma base, você pode escrever uma série de funções para transformar a representação básica em várias formas para os usuários — AM/PM, 24 horas, ou mesmo intervalos de tempo. Veja esta estratégia funcionando em *time.h* (Listagem 3.4).

Listagem 3.4 Funções hora do dia: *time.h*

```
static long secs; /* o tempo atual em segundos desde a meia noite */

/*************************************************************
 * set_time() acertara o tempo no formato 24 horas e acertara *
 *            o contador de segundos                          *
 *************************************************************/

set_time(tod24)
char *tod24;
{
  long hours, minutes;

  minutes = atol(&tod24[2]);  /* converte os ultimos dois digitos
                                 em um numero */
  tod24[2] = '\0';          /* apaga-os */
  hours = atol(tod24);      /* converte a primeira parte da cadeia */
  secs = (hours*3600)+(minutes*60);  /* define o total de
                                        segundos */
}

/*************************************************************
 * get_time() dara o tempo corrente -como esta armazenado no  *
 *            contador de segundos- no formato am/pm de 12 horas *
*************************************************************/

get_time(tod)
char* tod;
{
  static char *tday[] = {"MEIA NOITE","P.M.","MEIO DIA","A.M."};
  int hours, minutes, amflag = 0;
```

```
  hours = secs/3600;            /* calcula o total de horas */
  minutes = (secs%3600)/60;     /* ... e o total de minutos */

  if (hours == 24) {           /* fim de um dia - inicio de outro */
    hours = 0;
    amflag = (minutes == 0) ? 0 : 3; /* meia noite ou logo apos */
    }
  else if (hours > 12) {        /* o horario e pm */
    hours = hours - 12;
    amflag = 1;
    }
  else if (hours == 12) {       /* e meio dia */
    amflag = 2;
  else
    amflag = 3;                 /* se nao for nada mais, deve ser manha

  sprintf(tod,"%2d:%02d %s", hours, minutes, tday[amflag]);
}
```

Neste arquivo, a variável *seconds* é declarada *estática global*, para restringir seu escopo ao arquivo, proporcionando assim uma certa modularidade. A função C *set_time()* toma o tempo do formato 24 horas e define a variável *secs* (segundos) adequadamente. Por exemplo, se você colocar 0900 como parâmetros, *secs* torna-se 32.400; o horário de 2200 dá 79.200. Por outro lado, a função *get_time()* dá como resultado a hora do dia no formato 12 horas, convertendo o valor que se encontra na variável segundos no formato mais natural "horas e minutos". Neste caso, o valor retorna como um parâmetro de referência. Uma marca adicional no fim desta cadeia de caracteres indica a hora do dia e há quatro possíveis: AM, PM, NOON e MIDNIGHT (manhã, tarde, meio-dia e meia-noite).

O exemplo *time.c* é uma maneira perfeitamente razoável de lidar com a exibição do tempo, e representa uma boa solução em linguagem C para o problema. Você pode facilmente estender o seu alcance definindo funções separadas para calcular ou para mostrar qualquer outro aspecto desse dado que queira manipular. Por exemplo, você poderia criar uma função para converter a hora do dia em minutos transcorridos.

Vamos contrastar esta solução C para o problema com uma classe *time* definida em C++. A forma mais simples de uma classe, a estrutura, pode implementar este objeto, que, neste caso, é o tempo. A classe *time2.h* (Listagem 3.5) mostra esta nova implementação do objeto tempo.

Listagem 3.5 Uma classe para a hora do dia: *time2.c*

```
struct time {
  long secs;                  // o denominador comum para todos os
                              // formatos de tempo

  void set_time(char *);  // acertando a hora do dia

  void get_time(char *);  // recuperando o tempo corrente

  };

// Definicao das funcoes da classe

void time::set_time(char *tod24)
{
  long hours, minutes;

  minutes=atol(&tod24[2]);    // converte os dois ultimos
                              // digitos em numero
  tod24[2] = '\0';            // apaga-os
  hours = atol(tod24);        // converte a primeira parte da
                              // cadeia
  secs = (hours*3600)+(minutes*60); // acerta o total de segundos
}

void time::get_time(char *tod)
{
  static char *tday[] = {"MEIA NOITE","P.M.","MEIO DIA","A.M."};
  int hours, minutes, amflag = 0;

  hours = secs/3600;           /* calcula o total de horas */
  minutes = (secs%3600)/60;    /* ... e o total de minutos */

  if (hours == 24) {          /* fim de um dia - inicio de outro */
    hours = 0;
    amflag = (minutes == 0) ? 0 : 3; /* meia noite ou logo apos */
    }
  else if (hours > 12) {       /* o horario e pm */
    hours = hours - 12;
    amflag = 1;
    }
  else if (hours == 12) {      /* e meio dia */
    amflag = 2;
  else
```

```
    amflag = 3;          /* se nao for nada mais, deve ser manha */

  sprintf(tod,"%2d:%02d %s", hours, minutes, tday[amflag]);
}
```

Este pacote consiste em três partes: uma variável *long*, que armazena o número de segundos (*secs*) a partir da meia-noite, e duas funções-membro *set_time()* e *get_time()*. A função *set_time()* inicializa a variável *secs*; ela converte para segundos a cadeia de caracteres que contém o tempo no formato 24 horas. A função *get_time()* dá como resposta o tempo como uma cadeia de caracteres e mostra-o no formato conhecido AM/PM.

A codificação em ambas as listagens é idêntica. A diferença está no "empacotamento". Na Listagem 3.4, o denominador comum é a variável long *secs*. Para que esta variável seja comum a cada função do tempo, tem que ser global a ambas. Assim, ou ela deve ser global ao arquivo inteiro — como é aqui na declaração *static* — ou ao programa inteiro. A única maneira de controlar a visibilidade da variável *secs* é restringir estas funções a um arquivo e compilá-lo separadamente.

Agora, confronte a definição de *time.h* com a versão C++ em *time2.h*. A última versão preenche totalmente os requisitos para uma classe. Ela tem uma parte composta por dados, a variável long *secs*, e duas funções associadas. Estas funções têm um relacionamento especial com o objeto criado pela definição; elas podem referir-se diretamente à parte de dados, outro membro da estrutura, sem necessitar de nova expressão de referência indireta. Cada uma destas funções trata o membro *secs* como um localização comum de armazenamento. Ambas distinguem a variável e ambas podem acessá-la, seja para ler seu valor corrente seja para alterá-lo. Portanto, *secs* executa as mesmas funções que a variável long *static* no exemplo C anterior, na Listagem 3.4. Note também que este exemplo é declarado usando a palavra-chave *struct*. Uma estrutura em C++ é na realidade uma espécie simples de classe.

O escopo desta variável *secs* está limitado estritamente às duas funções definidas como membros da classe. Devido a esta variável ser reconhecida externamente à classe somente através de seus componentes nela contidos, você precisa declarar uma variável deste tipo para acessá-la. Por exemplo

```
    .
    .
char tstring[40];
```

```
time x;
    .
    .
x.secs = 23;
x.set_time("2000");
x.get_time(tstring);
    .
    .
```

são as únicas maneiras pelas quais você pode acessar os membros desta classe.

Em ambos os programas, você pode declarar tantos objetos da espécie *time* quantos precisar. Cada um é um objeto único, mas que compartilha seu formato e funcionalidade com outros objetos do mesmo tipo. Por exemlo, cada vez que você declara uma variável da classe *time*, é criada uma nova cópia ou modelo da classe com a estrutura especificada. Isso significa que as funções da classe agem sobre a parte específica de dados associada àquela cópia da classe. Chamar a função *set_time()* para um objeto *time* altera o contador de segundos (*secs*) naquela cópia somente. Chamar esta função para outra cópia altera os valores somente para aquela cópia.

Assim, cada vez que você cria um novo objeto da classe, cria novos membros e reserva memória para as variáveis. Observe, no entanto, que, se o membro for um ponteiro, é alocado somente espaço suficiente para alojar o endereço; o sistema não reserva memória para a variável apontada, um fato que é fácil esquecer quando você está trabalhando com cadeias de caracteres. As funções-membro de uma classe-objeto são uma história diferente. Devido a não ser eficiente copiar a codificação da função em cada objeto assim que ele é criado, as funções-membro são ligadas a cada objeto que estão controlando e mantêm registro da classe para a qual estão trabalhando. Somente uma cópia é mantida na memória. (A única exceção a essa regra é a função-membro *inline*, que é discutida no próximo capítulo.)

O programa *dtime.c* (Listagem 3.6) é um programa acionador que ilustra e testa a operação da classe *time2.h* definida na Listagem 3.5.

Listagem 3.6 Um programa acionador que testa as definições de tempo: *dtime.c*

```
// usa as instrucoes "include" apropriadas
// "time.h" contem as definicoes da classe "time"

#include <stream.h>
```

```
#include "time.h"

main(int argc, char *argv[])
{
  if (argc < 3) // se nao houver parametros suficientes, bater
                // em retirada ja
    exit(1);

  time t0,t1;      // nos precisamos de dois objetos time

  t0.set_time(argv[1]);      // acerta o tempo corrente
  t1.set_time(argv[2]);

  char tod0[15], tod1[15];  // declara duas cadeias de caracteres

  t0.get_time(tod0);        // busca o tempo
  t1.get_time(tod1);

  cout << form("tempo t0 = %s\ntempo t1 = %s\n",tod0,tod1); //...e
                                                           // o exibe
}
```

O programa é chamado por uma série de parâmetros do tipo linha de comando. Se o usuário não fornecer dois parâmetros, o programa pára. Com a entrada correta, o programa declara dois objetos do tipo *time-t0* e *t1* — e inicializa-os com os argumentos da linha de comando *argv(1)* e *argv(2)* respectivamente. A inicialização é conseguida chamando a função-membro apropriada:

```
t0.set_time(argv[1]);
t1.set_time(argv[2]);
```

Os dois argumentos são cadeias de caracteres que contêm o número de segundos com que a variável *secs* é definida. O programa não faz nenhuma verificação de erro; ele simplesmente assume que o valor introduzido é um número no intervalo adequado.

Após cada objeto tempo ter um valor, o membro *get_time()* mostra o valor corrente do tempo. Esta função-membro converte o tempo corrente (medido em segundos) no formato mais familiar de *horas:minutos*. Então, é usada a função de formatação de saída *form()* com a insturução *cout* para exibir o tempo e o designador apropriado — "AM", "PM", "MEIO-DIA" ou "MEIA-NOITE".

Embora a estrutura seja a classe mais simples do que pode ser definida em C++, ela não é a mais útil. E, apesar da parte dos dados ter um relacionamento especial com as funções-membro, ela está, na verdade, disponível a qualquer função dentro de seu escopo. Assim, se você cria um programa num único arquivo e declara uma estrutura global no seu início, quaisquer variáveis daquela estrutura podem ser alteradas por funções que não são membros da estrutura. Este não é um bom grau de modularidade. Assim, uma estrutura não se beneficia totalmente do mecanismo da classe.

Classes Completas em C++

Com a linguagem C++, você pode criar uma estrutura de classe mais completa usando a palavra-chave "class" em vez de "struct". O exemplo *time3.h* (Listagem 3.7) mostra a declaração da classe *time* anterior transformada numa *verdadeira* classe.

Listagem 3.7 Uma classe *time* verdadeira: *time3.h*

```
class time {
  long secs;// o denominador comum para todos os formatos de tempo
            // privativo da classe objeto

  public:
    void set_time(char *);     // acertando a hora do dia

    void get_time(char *);     // obtendo a hora corrente
  };
```

Além do uso da palavra-chave *class* em lugar de *struct*, a principal diferença entre esta versão e a classe mais simples definida na Listagem 3.5 é a palavra *public*, que aparece no meio da definição. Essa palavra-chave divide a classe em duas partes — a parte pública, ou externa, que está inteiramente disponível ao resto do programa, e o restante da definição, que está restrito às funções-membro da classe. Veja figura 3.2.

O acesso à parte privada de uma classe é barrado a todos, menos às funções-membro da própria classe, que têm a mesma relação especial daquelas

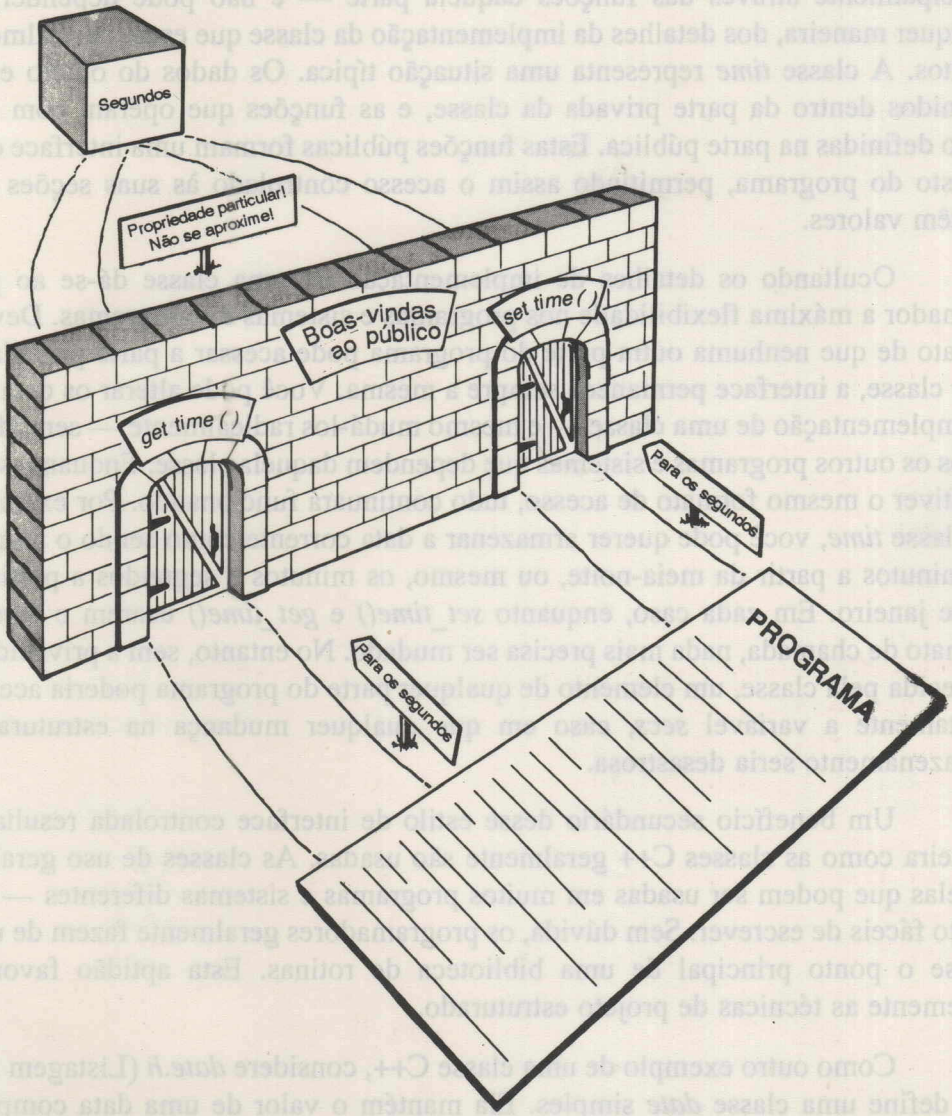


Figura 3.2 A diferença entre as seções pública e privada de uma classe.

que estão na estrutura. Funções que não são membros da classe podem acessar a parte privada somente através das funções ou variáveis na parte pública. Assim,

um programa só pode acessar uma classe através de sua parte pública — principalmente através das funções daquela parte — e não pode depender, de qualquer maneira, dos detalhes da implementação da classe que estão parcialmente ocultos. A classe *time* representa uma situação típica. Os dados do objeto estão definidos dentro da parte privada da classe, e as funções que operam com eles estão definidas na parte pública. Estas funções públicas formam uma interface com o resto do programa, permitindo assim o acesso controlado às suas seções que contêm valores.

Ocultando os detalhes de implementação de uma classe dá-se ao programador a máxima flexibilidade nos programas e sistemas de programas. Devido ao fato de que nenhuma outra parte do programa pode acessar a parte privada de uma classe, a interface permanece sempre a mesma. Você pode alterar os detalhes de implementação de uma classe — e mesmo mudá-los radicalmente — sem afetar todos os outros programas e sistemas que dependem daquela classe. Enquanto você mantiver o mesmo formato de acesso, tudo continuará funcionando. Por exemplo, na classe *time*, você pode querer armazenar a data corrente como sendo o número de minutos a partir da meia-noite, ou mesmo, os minutos e segundos a partir de 1º de janeiro. Em cada caso, enquanto *set_time()* e *get_time()* usarem o mesmo formato de chamada, nada mais precisa ser mudado. No entanto, sem a privacidade oferecida pela classe, um elemento de qualquer parte do programa poderia acessar diretamente a variável *secs*, caso em que qualquer mudança na estrutura de armazenamento seria desastrosa.

Um benefício secundário desse estilo de interface controlada resulta da maneira como as classes C++ geralmente são usadas. As classes de uso geral — aquelas que podem ser usadas em muitos programas e sistemas diferentes — são muito fáceis de escrever. Sem dúvida, os programadores geralmente fazem de uma classe o ponto principal de uma biblioteca de rotinas. Esta aptidão favorece fortemente as técnicas de projeto estruturado.

Como outro exemplo de uma classe C++, considere *date.h* (Listagem 3.8) que define uma classe *date* simples. Ela mantém o valor de uma data completa como uma série de números inteiros que representam o dia, o mês e o ano. Isso constitui a parte privada da classe. A função *new_date()* definida em *date.h* permite que o programador acerte a data, e a função *give_date()* mostra o valor corrente da data. Observe que a cadeia de caracteres que é o vetor *mname()*, o qual contém

os nomes dos meses indexados por seu número (janeiro é 1, junho é 6, e assim por diante), é uma constante porque nunca muda durante a execução do programa.

Listagem 3.8 Uma classe *date* simples: *date.h*

```
const char* mname = {"mes incorreto","Janeiro","Fevereiro",
                     "Marco","Abril","Maio","Junho","Julho",
                     "Agosto","Setembro","Outubro","Novembro",
                     "Dezembro"};

class date {
  int day,        // esquema interno de armazenamento
      month,      // isto e agora privativo da classe
      year;       // e desconhecido externamente

public:
  void new_date(int,int,int);  // uma funcao membro para
                               // mudar a data

  char* give_date();           // ... e uma para relata-la
};
void date::new_date(int d, int m, int y)  // toma os novos valores
{
  day = d;   // atribui os valores dos parametros para as
  month = m; // variaveis de armazenamento interno
  year = y;
}
char* date::give_date()
{
  char *buf = new char[80];

  sprintf(buf,"%s %d,%d",mname[month],day,year);
  return buf;
}
```

Questões de Revisão

1. Uma classe contém membros _____ e _____.
2. A forma mais simples de uma classe é a tradicional _____
3. Combinando _____ e _____ no mesmo objeto, você pode criar uma _____ mais completa.
4. Uma classe define um _____ dentro de um programa.

Membros Privados versus Públicos de uma Classe

Os exemplos anteriores demonstram que a classe em C++ é uma ferramenta poderosa para modularizar programas. De certo modo, sua operação é análoga ao uso dos arquivos e variáveis estáticas pelos programadores C. Embora uma classe seja muito mais do que um substituto para estas estruturas de arquivo, este é um ponto bom para iniciar as explorações dos aspectos práticos da idealização de classe.

Uma classe representa uma parte de um programa C++, que é isolado intencionalmente de todas as outras partes. Geralmente, esse é um ponto de confusão. Os itens da parte privada da classe têm um escopo que se estende à classe inteira — partes privadas e pública — mas não além. Você não pode, por exemplo, acessar uma variável nesta parte privada, de fora da classe; isso é semelhante à relação que há entre uma função e suas variáveis locais. Há duas coisas importantes, no entanto, que diferenciam a classe C++ das simples estruturas C. Em primeiro lugar, uma classe contém funções. Segundo, uma classe pode aparecer em qualquer lugar dentro de um programa — até mesmo no meio de uma função. Lembre-se, a função principal de uma classe é modularizar um programa; este é o segredo para usá-la com eficácia.

O que você deverá colocar na parte privada da classe? Algumas vezes, esta é uma das piores questões de projeto a resolver. O padrão geral para classes simples é colocar os valores dos dados na parte privada e as funções de acesso, na parte pública. Todos os exemplos anteriores de classes seguiram esta diretriz. Na classe definida em *time3.h* (Listagem 3.7), o membro variável que contém o

tempo corrente em segundos (*secs*) está oculto à visão, assim como também os três números inteiros na Listagem 3.8. No entanto, as funções que manipulam estas variáveis são públicas. Embora este padrão seja o costumeiro, você não deve considerá-lo como uma regra. Há situações em que uma função-membro privada é melhor, e algumas classes até requerem a parte dos valores como pública. No entanto, esta disposição típica geralmente faz sentido por razões práticas. Usualmente, os dados de qualquer conjunto são propensos a mudar. Por exemplo, colocar um programa numa máquina diferente pode causar diferença na magnitude de um valor. Ou você pode implementar uma estrutura de dados de nível superior como um vetor, numa certa situação e como uma lista concatenada em outra. Usualmente as funções são públicas, pela simples razão de que o programa deve ter acesso aos dados, se de qualquer modo elas forem úteis.

O exemplo *stack2.h* (Listagem 3.9) é uma definição de uma *stack* de dados como uma classe C++. Esta classe é mais complicada do que os exemplos anteriores, mas ainda representa uma definição típica de classe. Ela implementa a pilha, que é uma estrutura de dados LIFO (last-in, first-out — o último que entra é o primeiro que sai — como um pequeno vetor ou série (*stk()*) e usa uma variável inteira (*top*) para indicar o início corrente do vetor. Quando você coloca (*push*) um valor na pilha, a variável *top* é incrementada. Uma chamada à função *pop()* decrementa *top*, que se move entre 0 e um valor máximo definido pela constante *MAX_STACK*. A função *error_rep()* proporciona um meio simples de mensagem de erro. A função *error_rep()* e as duas variáveis-limite estão ocultas na parte privada da classe. A seção pública é formada por três funções de acesso à pilha: *init()*, que inicializa a classe dando valores apropriados para *top*; *push()*, que coloca um valor na pilha; e *pop()*, que retira um valor.

Listagem 3.9 Uma classe *stack*: *stack2.h*

```
#include <stream.h>

const char* msg[] = {"stack overflow\n","stack underflow\n"};

const MAX_STACK = 5;  // define o numero maximo de elementos
                      // da pilha
class stack {
  int top,            // o ponteiro da pilha
    stk[MAX_STACK];   // ... e os elementos

  void error_rep(int e_num) {cout << msg[e_num];} // mostra erros
```

```
public:
  void init() { top=0; }    // inicializa o ponteiro da pilha
  void push(int,int&);      // coloca um novo valor na pilha
  int pop(int&);            // retira o valor de cima
};

////////////////////////////////////////////////////////////////
//    stack::push()   coloca na pilha um valor especificado.
////////////////////////////////////////////////////////////////

void stack::push(int x,&flag=1)
{
  if (top == MAX_STACK) { // verifica se ha estouro na pilha
                          // antes de incrementar
    error_rep(0);         // ... mostra erro, se necessario
    flag = 0;             // ativa o flag de sucesso
    }
  else
    stk[++top] = x;
}

////////////////////////////////////////////////////////////////
//    stack::pop()   retira da pilha o elemento de cima.
////////////////////////////////////////////////////////////////

int stack::pop(int &flag = 1)
{
  if (top == 0) {           // verifica se a pilha esta vazia antes
                            // de decrementar
    error_rep(1);           // mostra erro, se necessario
    flag = 0;               // ativa o flag de sucesso
    return 0;
    }
  return stk[top--];
}
```

Observe a semelhança desta listagem com a codificação C que implementou a pilha em *stack.c*. Os algoritmos básicos para *push()* e *pop()* são os mesmos. Em primeiro lugar, *push* verifica se há estouro (overflow); depois incrementa a variável *top*. Finalmente ela usa esse valor como um índice da pilha, no qual ela atribui o valor passado pelo membro. A função *pop()* verifica se a pilha está vazia (underflow) e passa o valor do vetor para a função que solicitou. Depois, *top* é

decrementada. Estas descrições aplicam-se a ambas as versões. A classe *stack* faz algumas operações adicionais de indicação de erros e usa um formato diferente, mas estas são alterações superficiais.

A grande diferença entre a pilha definida em C e a pilha definida em C++ é que na primeira você pode ter somente uma pilha no programa. Para criar múltiplas pilhas você tem que fazer múltiplas cópias do arquivo que contém as instruções e depois redenominar cada função. Com a classe *stack* definida em C++, você pode declarar quantas pilhas precisar:

```
stack, x, y, z;
```

Todas elas são únicas e todas têm acesso a todos os recursos da pilha. Não é necessário nenhum código extra de seleção. Você coloca os valores (*push*) nas pilhas simplesmente especificando a pilha com a qual quer trabalhar, da seguinte forma:

```
x.push(2);
y.push(3);
z.push(4);
```

Para retirar valores da pilha (*pop*) a operação é semelhante:

```
int flagx, flagy, flagz;
cout << x.pop(flagx) << "\n";
cout << y.pop(flagy) << "\n";
cout << z.pop(flagz) << "\n";
```

Em ambos os casos, as mesmas funções e a mesma sintaxe aplicam-se a cada objeto.

Observe que na parte privada da classe residem a seção de dados e uma função, e todas as outras funções estão na parte pública. O critério para a parte pública é óbvio: o programador precisa ter acesso à pilha. Deve haver pelo menos uma espécie de operação *push* e *pop*, pois, em caso contrário, não é uma pilha de modo algum. Estas duas funções representam a interface com o resto do programa. Fica mais ou menos claro que o resto do programa deve ser mantido à distância da implementação da pilha. Isso evita instruções que manipulam diretamente o vetor *stk()* para melhor eficiência. Mais tarde você talvez queira reimplementar a pilha como uma lista concatenada e por isso deve manter o restante do programa na parte privada, de modo que outro programador ou um descuido não possam destruir seu trabalho. Mas o que dizer de *error_rep()*? Por que ela é interna e

oculta aò resto do programa, apesar de não ter um papel particularmente importante na operação básica da pilha? Somente as funções-membro usam a função de mensagem de erro na pilha; ela não é solicitada diretamente por qualquer outra parte do programa. Portanto, sendo um membro puramente interno, ela pertence à parte privada, ou interna, da classe. A parte pública, ou externa, deverá ficar restrita àqueles membros que controlam o acesso à classe.

Observe que ambas as funções *error_rep()* e *init()* são funções do tipo *inline*. Devido a elas serem muito pequenas, você deve declará-las assim para evitar o trabalho extra de chamar uma função. Você pode definir uma função-membro como *inline* de duas maneiras diferentes — pela maneira usual, declarando-a através da palavra-chave *inline*, ou incluindo o corpo da função dentro da declaração, como neste exemplo. Neste segundo caso, você não precisa usar o especificador *inline*.

Classes Aninhadas

Embora a classe seja uma espécie de dados poderosa, mesmo quando está restrita a membros simples, outras classes podem também ser definidas como sendo membros de uma dada classe. Assim, você pode criar espécies de dados arbitrariamente complexos alojando classes dentro de classes. Isso melhora ainda mais a modularidade do programa. Neste tipo de combinação, podem surgir alguns problemas, mas o resultado é tão poderoso que compensa o trabalho. O exemplo *phone.h* (Listagem 3.10) demonstra o alojamento de classe numa simples agenda telefônica.

Listagem 3.10 Definição de uma classe aninhada: *phone.h*

```
#include <string.h>

struct name {
  char first[40],
       mid[40],
       last[80];
};

struct phone {
  char area[4],
```

```
    exch[4],
    number[5];
};

class p_rec {
  name name;
  phone phone;

public:
  void fill_rec(char *[]);
  void display_rec();
};

void p_rec::fill_rec(char *info[])
{
  strcpy(name,first,info[0]);
  strcpy(name,mid,info[1]);
  strcpy(name,last,info[2]);
  strcpy(phone,area,info[3]);
  strcpy(phone,exch,info[4]);
  strcpy(phone,number,info[5]);
}

void p_rec::display_rec()
{
  cout << form("%s %s %s\n",name.first,name.mid,name.last);
  cout << form("(%s) %s-%s\n",phone.area,phone.exch,phone.number);
}
```

Este exemplo define uma classe chamada de *p_rec* que contém o nome e o número do telefone de uma pessoa. O nome da pessoa está dividido em: nome, segundo sobrenome e sobrenome. O número do telefone está também dividido em: código de área, prefixo e número. Para simplificar a tarefa, o membro *name* da classe é ele próprio uma classe: a estrutura *name*. O membro *phone* também é definido previamente desta maneira. Em ambos os casos, as subclasses são grupos ou cadeias de caracteres. A interface é formada pelas funções-membro *fil_rec()* e *display_rec()*. Esta última não tem parâmetros, e simplesmente exibe os valores correntes da classe. A função *fil_rec()* usa como parâmetro uma série de cadeias de caracteres e estabelece adequadamente a parte privada (ou interna) de *p_rec*.

O uso de uma classe *struct* como componente para outra classe é uma operação razoavelmente direta. A simplicidade das estruturas — particularmente aquelas que não possuem funções-membro — ajusta-se a muitas situações comuns

de projeto. Um perigo inerente à habilidade em definir classe é a tentação de criar estruturas de classe complexas. Geralmente, a complexidade resultante destas várias estruturas de classe aninhadas e interdependentes anula as vantagens do uso das classes, em primeira instância. Um elemento importante na filosofia da programação C++ é que você não deve sobrecarregar os programas com construções de classe muito complexas.

Outro ponto importante a ponderar quando você trata com estruturas aninhadas envolve o escopo da definição de classe. C++ ainda é, em essência, C, e o grau de ocultamento que você pode realizar é ainda restrito. Para entender esta restrição, considere a seguinte definição alternativa para a classe *p_rec* na Listagem 3.10:

```
class p_rec   {
    struct name  {
       char first [40]
            mid [40]
            last [80]
} name;

 struct phone  {
     char area [4]
          exch [4]
          number [5]
 }phone;
public:
 void fill_rec(char *[]);
 void display_rec();
};
```

Embora não pareça, esta codificação é equivalente à definição anterior. Não obstante as estruturas serem definidas na parte interna (privada) da classe e as definições parecerem muito bem aninhadas, as definições de classe, na realidade, não estão ocultas ao resto do programa. As estruturas *name* e *phone* estão, na verdade, disponíveis em qualquer outro lugar do programa como novas espécies de dados. Você poderia, por exemplo, declarar uma variável como sendo do tipo *name* posteriormente no programa. Devido ao fato de você não poder ocultar definições da mesma maneira como oculta dados, você é pouco motivado a usar esta forma. A definição original é mais elegante e mais fácil de ler. O ponto importante a lembrar é que aquilo que você quer fazer com abstração de dados tem limites.

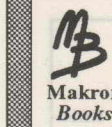
Questões de Revisão

1. Uma classe completa tem duas seções: uma _____ e uma _____.
2. A seção privada, ou interna, de uma classe pode conter membros _____ e _____.
3. A parte privada de uma classe permite que o programador oculte os _____ _____ da classe.
4. A parte pública (externa) da classe cria um _____ para o resto do programa.

Resumo

Este capítulo explorou muitos aspectos da classe C++. As classes permitem que o programador crie pequenas regiões autônomas dentro de um programa — regiões com suas próprias variáveis e funções locais. Estas regiões proporcionam o tipo de modularização necessária para o projeto moderno de software. Além disso, a classe C++ é um meio ideal para implementação da notação de um dado-objeto, ou uma representação estruturada de uma entidade complexa do mundo real. Você pode usar dados-objeto para aumentar ainda mais a modularidade e estrutura de um programa.

Uma classe representa uma espécie de dados de alto desempenho, definidos pelo usuário, que inclui tanto o armazenamento quanto funções de manipulação de dados. A parte interna contém variáveis e funções que podem ser acessadas somente por outros membros da classe. A seção externa (pública) representa a interface com o resto do programa.



Capítulo 4
Criando Classes Complexas

Criando objetos de classes: construtores e destrutores
A parte ativa de uma classe
Criando uma função amiga
Estruturas de dados unidas usando classes
Resumo

Este capítulo continua a discussão de classes em C++ e apresenta algumas outras características desta importante ferramenta da linguagem C++. Entre estas características, as principais são:

- construtores e destrutores
- função-membro *inline*
- a sobrecarga de funções-membro
- funções *amigas*
- membros estáticos

Estas características são ferramentas versáteis que permitem ao projetista criar estruturas de programação objeto-orientadas ainda mais poderosas. A discussão final deste capítulo refere-se ao uso das classes C++ para criar estruturas de dados articuladas, e culmina com uma definição de uma espécie de lista articulada de classe *container*.

Criando Objetos de Classe: Construtores e Destrutores

Para ajudá-lo a entender plenamente o conceito de classe, veremos como C++ reserva e administra o espaço de memória para uma classe. Primeiramente, lembre-se de como C reserva espaço de memória para variáveis em geral. Quando você declara um número inteiro, ou uma variável do tipo double, C reserva espaço para seu armazenamento em algum lugar da memória: às vezes numa pilha, ou em um segmento especial de memória. Onde e quando este espaço é reservado depende da interação da classe de armazenamento e do escopo da variável. Por exemplo, C pode colocar as variáveis *extern* e *static* em uma localização e variáveis *auto* (automáticas) em outra. Esta alocação é uma operação imediata porque tanto as variáveis quanto outros dados objetos não são dinâmicos em C e C++; o compilador os cria antes que seja executada qualquer ação sobre eles, independente de estarem no começo do programa (variáveis *static* e *extern*) ou na introdução a uma função (valores automáticos). C sempre conhece os parâmetros da alocação — tamanho etc.— antes que a memória seja realmente alocada. O mesmo é válido para ordens e disposição de classes que não tenham quaisquer funções-membro.

C também executa o mesmo tipo de alocação de memória para estruturas do tipo definidas pelo usuário. A definição *struct* especifica uma série de localizações variáveis numa certa ordem. Estas variáveis são contíguas na memória. C aloca a estrutura como um todo, com a exceção de variáveis-membro que sejam ponteiros. A única memória alocada para um ponteiro é para armazenamento de um endereço, e isso não inclui a localização de memória para a qual aponta a variável-membro. O exemplo a seguir:

```
struct info {
char *name,
     *address,
     * phone;
} x;
```

aloca três variáveis de ponteiro, mas não aloca o espaço para as cadeias de caracteres que elas indicam. Se você quer atribuir um nome ao membro *name*, primeiro você precisa solicitar a função de alocação *malloc()* ou encontrar alguma outra maneira de colocar em *name* o endereço de uma cadeia de caracteres.

No entanto, a criação de objetos de classes completos é mais complicada porque eles têm uma estrutura interna que inclui membros de armazenamento de dados e membros funcionais, além de ter a complicação adicional de possuírem seções pública e privada. Uma complicação ainda maior aparece quando uma classe contém variáveis de ponteiro que precisam ser inicializadas.

Construtores

Para manipular a complicação maior da alocação de classe e para melhorar seu poder, o mecanismo de declaração de classe inclui a propriedade de executar uma função especial classe-específica conhecida como *constructor* (construtor). A sintaxe da função construtor é simples — ela sempre tem o mesmo nome que a classe. Por exemplo, se a classe se chama *link_node*, a função construtor é *link_node()*. Essa função é sempre executada quando você cria um objeto do tipo classe. Para uma variável de armazenamento de classe *extern*, é chamada a função construtor uma vez no início do programa. Em contrapartida, se a variável for automática, a função construtor é chamada cada vez que é introduzido o escopo.

Nota sobre Terminologia

A terminologia de linguagens objeto-orientadas como C++ pode ser confusa, porque nem sempre é padronizada. Na discussão de classes C++ este livro usa os termos "objeto" e "variável" alternadamente para referir-se a um modelo completo de uma classe, incluindo todos os seus membros. O termo "variável-membro" ou simplesmente "membro" refere-se a um item de dado individual ou função-membro de uma classe.

Note que C++ solicita automaticamente a função construtor sem qualquer ação explícita pelo programa. Portanto, você não precisa se preocupar onde ou quando deveria usá-la.

Entretanto, o programador determina o conteúdo da função *constructor*. Não há restrições especiais e ela pode executar uma vasta gama de ações iniciais tanto na classe quanto nos seus membros constituintes. Aqui estão algumas das ações típicas de inicialização:

- alocar variáveis de ponteiro internas (por exemplo, criar variáveis cadeia de caracteres);

- atribuir valores específicos a variáveis-membro;
- executar programa de inicialização de máquina ou de dispositivo específico.

Esta lista, de modo algum esgota todas as possibilidades. Para entender melhor o uso de construtores, considere *message.h* (Listagem 4.1), a definição de uma classe *message*.

Listagem 4.1 Uma classe *message* simples: *message.h*

```
#include <stream.h>
#include <string.h>

class message {
  char *s;                  // um ponteiro para o conteudo da mensagem
  int len;                  // o tamanho da mensagem

public:
  message(char*);           // declara um construtor

  void send(char*);         // envia a mensagem para o seu destino
  void get(char*);          // obtem uma dele
  void clear() { *s = '\0'; }  // inicia com um novo buffer
  char* read_buf() { return s; }  // pega o conteudo da mensagem
};

message::message(char* msg)  // declara um objeto com uma mensagem
{
  len = strlen(msg);            // reserva memoria para a mensagem
  s= new char[len+1];           // ... e a copia la
  strcpy(s,msg);
}

void message::send(char* p)    // p aponta para o buffer de destino
{
  char* x=s;
  for (int i=0; i<=len; i++)  // copia caratere a caractere
    *p++ = *x++;
}

void message::get(char* p)       // aqui p e quem envia
{
  char* x=s;
  while ( (*x=*p) != '\0' ) {  // copia do remetente para o
```

```
                                    // buffer local
    x++;
    p++;
    }
}
```

Neste exemplo, a classe *message* cria um objeto que tem como conteúdo uma cadeia de caracteres. Geralmente, os caracteres são utilizados como dados de uso geral (embora você poderia substituir o ponteiro *s* pelo ponteiro *void* para torná-lo compatível com todos os tipos de dados). Um objeto message contém tanto os caracteres quanto um valor, indicando sua extensão. O exemplo define apenas quatro funções-membro:

- *send*() — copia o conteúdo, caractere por caractere, no local de destino
- *get*( ) — copia uma réplica do lugar de destino
- *read_buf*( ) — acessa o buffer interno contendo a mensagem
- *clear*( ) — reseta o buffer

A idealização de um modelo de mensagem transitória é simples: ele é baseado numa localização de memória que é comum tanto para o emissor quanto para o receptor. Para passar uma mensagem, você necessita apenas preencher esta localização. Você pode também facilmente expandir este modelo incluindo qualquer sistema que seja capaz de transmitir caracteres um a um. Por exemplo, o caractere na função *send*() poderia facilmente ser copiado numa porta de I/O.

Assim, você poderia usar esta classe mensagem como parte de um programa de comunicação de dados. Para isso, você precisa substituir a localização de memória pelo endereço da porta serial e depois acrescentar a instrução apropriada para mandar a mensagem através da porta. A comunicação de dados é o mais óbvio — mas de modo algum o único — lugar onde este tipo de mensagem transitória ocorre. Este tipo de organização é útil sempre que você necessita coordenar a atividade de dois ou mais subsistemas — hardware e software — em um único computador. Mesmo as aplicações que não são de comunicação podem se beneficiar deste recurso de mensagem transitória. Por exemplo, embora você possa considerar uma impressora como sendo um dispositivo apenas de saída, ela geralmente manda de volta informações sobre o estado em que ela se encontra (status) para o computador que está ordenando a impressão. Uma classe *message* seria uma maneira estruturada ideal de manipular esta informação em baixo nível

(geralmente estes valores estão em forma de "bit map" ou apenas uma palavra de computador).

Agora, vamos examinar a função construtor da classe *message( )*. Após você criar o objeto da mensagem, ela deve ser inicializada com o conteúdo da mensagem. Esta é a tarefa primária da função construtor, que aloca memória, liga-a à variável de ponteiro *s*, e copia os conteúdos. Este exemplo demonstra os usos típicos da função construtor.

O exemplo também ressalta uma importante característica das funções construtor. Se uma função construtor necessita de um argumento, você deve fornecê-lo quando declara a classe-objeto. No exemplo, o objeto da mensagem pode ser definido fornecendo-o com algum valor, assim:

```
message msg("isto é apenas um teste");
```

Naturalmente, você deve também fornecer quaisquer parâmetros adicionais que a função construtor requeira.

Destrutores

A função-membro conhecida como *destrutor* é o complemento da função construtor. Quando um variável não é mais necessária, um sistema que funcione bem devolve ao sistema operacional as posições de memória que estavam ocupadas pela variável para serem realocadas posteriormente. Naturalmente, esta operação depende do escopo da variável e da classe de armazenamento. Por exemplo, uma variável automática pode ser criada e destruída muitas vezes num programa, enquanto uma variável *static* ou *extern* é eliminada somente quando o programa termina. Assim como há o construtor, C++ também fornece o destrutor com o tipo de dado *class*. Quando um programa passa além do escopo de um objeto *class*, ele solicita um destrutor, se houver algum definido. Assim, a função destrutor é a ferramenta ideal para eliminar os desarranjos criados pela operação da própria classe.

Geralmente, funções destrutor são complementares das funções construtor. Se você tivesse que alocar uma variável-membro com o operador *new*, deveria desalojá-la com a função destrutor correspondente. Você pode também usar a

seqüência criada pela execução desta função para desligar quaisquer dispositivos ou subsistema do computador que foram abertos como representantes da classe. Esta operação importante pode ser efetuada enviando-se uma cadeia que anula a inicialização para uma impressora ou para uma porta, por exemplo. Mesmo coisas tão básicas, como, por exemplo, desalojar um usuário, poderiam muito bem ser executadas pelo destrutor.

O exemplo *message2.h* (Listagem 4.2) acrescenta um destrutor à classe *message* definida anteriormente. Devido à função destrutor ser o complemento da função construtor, ela é rotulada com um til (~), que usualmente é reservado para a operação complemento bit a bit. Assim, a função destrutor para a classe *message* é chamada de *message::~message( )*. Fora o notável simbolismo do seu nome, esse destrutor é uma simples função que limpa a memória interna da classe. Este é um uso bastante típico — e necessário — desta função. Lembre-se de que a memória alocada através do operador *new* existe até a subseqüente chamada da função *delete*. Esta memória poderia facilmente ser perdida se uma variável de classe passar além de seu escopo sem liberá-la. A memória estaria ainda alocada, mas você não poderia acessá-la porque a variável da classe teria se perdido. A outra variável-membro desta classe, *len*, é um simples número inteiro que é retomado pelo mecanismo simples que cuida dos tipos de dados internos.

Listagem 4.2 Uma classe *message* com um destrutor: *message2.h*

```
class message {
  char *s;                  // um ponteiro para o conteudo da mensagem
  int len;                  // o tamanho da mensagem

public:
  message(char*);           // declara um construtor
  ~message();               // retorna a memoria alocada pelo Sistema
                            // Operacional

  void send(char*);         // envia a mensagem para o seu destino
  void get(char*);          // obtem uma dele
  void clear();             // inicia com um novo buffer
  char* read_buf();         // pega o conteudo da mensagem
};

message::~message()
{
  delete s; // retorna a memoria alocada ao banco de memoria livre
}
```

Construtores e Destrutores Globais

A função destrutor é outra ferramenta que facilita atingir o objetivo da linguagem C++, que é permitir-lhe criar modelos definidos pelo programador que são tão convenientes e poderosos quanto os modelos internos. Mas não permita que este objetivo obscureça o fato de que esta facilidade — juntamente com a outra função construtor — pode ser usada para muitas coisas mais do que apenas recuperar recursos de memória. Assim como a função construtor amplia a noção de inicialização, também a função destrutor permite a redefinição da inicialização.

Você pode usar construtores e destrutores para criar um módulo que será a última instrução executada num programa. Lembre-se de que as variáveis declaradas como estáticas ou globais para qualquer função existem durante todo o tempo em que se roda o programa. Elas passam a existir quando são declaradas e retornam ao sistema operacional após a função *main( )* terminar sua execução — em outras palavras, no fim do programa. Esse procedimento é o mesmo tanto para variáveis de classe como para variáveis internas. Se uma variável de classe tem uma função construtor, a função construtor é solicitada quando a variável é declarada, e no fim do programa é chamada uma função destrutor. Por exemplo, se você faz uma declaração antes da função *main( )*, então a função construtor é executada antes do início do programa. A função destrutor é executada após o término de *main()* e é a última instrução executada no programa.

Esta possibilidade de executar funções antes do início e após o fim do programa dá a você um controle melhor sobre o ciclo de execução de um programa do que seria possível em C. E mais ainda, você exerce esse tipo de controle não com um acesso em baixo nível à máquina ou ao sistema operacional, mas através de uma programação fluente. Isso torna portáteis as instruções que você produz.

O programa *phone2.c* (Listagem 4.3) ilustra a discussão anterior. O programa inicia definindo uma classe simples que usa uma função construtor e uma função destrutor. A função construtor introduz na classe-objeto os valores fornecidos pelo usuário. A função destrutor simplesmente exibe os valores correntes antes de abandonar as variáveis. A parte interessante do exemplo focaliza a declaração da classe-objeto — ela é declarada diretamente antes da função *main( )*. Da mesma forma, é solicitada a função destrutor após o término da execução de *main( )*.

Listagem 4.3 Um programa que usa funções globais construtor e destrutor: *phone2.c*

```
#include <stream.h>
#include <string.h>

// define um tipo de classe interessante com um construtor
// e um destrutor

struct name {
  char first[40],
       mid[40],
       last[80];
};

struct phone {
  char area[4],
       exch[4],
       number[5];
};

class p_rec {
  name name;
  phone phone;

public:
  p_rec();                  // declara um construtor para a classe
  ~p_rec();                 // e um destrutor
  void display_rec();       // declara uma funcao de exibicao simples
};

p_rec::p_rec()
{
  cout << "construtor chamado:\n";  // para indicar que a funcao
                                    // foi solicitada
  cout << "primeiro nome:";         // reune os valores do usuario
  cin >> name.first;
  cout << "nome do meio:";
  cin >> name.mid;
  cout << "ultimo nome:";
  cin >> name.last;
  cout << "codigo de area:";
  cin >> phone.area;
  cout << "prefixo:";
  cin >> phone.exch;
```

```
  cout << "numero:";
  cin >> phone.number;
  cout << "\n\n\n";
}

p_rec::~p_rec()
{
  cout << "destrutor chamado:\n";
  cout << form("%s %s %s\n",name.first,name.mid,name.last);
  cout << form("(%s) %s-%s\n",phone.area,phone.exch,phone.number);
  cout << "\n\n\n";
}

void p_rec::display_rec()        // exibe os valores correntes
{
  cout << form("%s %s %s\n",name.first,name.mid,name.last);
  cout << form("(%s) %s-%s\n",phone.area,phone.exch,phone.number);
  cout << "\n\n\n";
}

p_rec x;                  // declara uma variavel class global para main

main()
{
  cout << "aqui ele e exibido na funcao main():\n";
  x.display_rec();        // faz alguma coisa com a classe em main
}
```

Esse simples programa tem pouca importância; ele apenas ilustra a interação entre a declaração de uma variável — seu escopo e classe de armazenamento — e a execução das funções construtor e destrutor. No entanto, você pode substituir as rotinas aqui definidas por outras mais úteis. Por exemplo, a função destrutor pode também introduzir a classe-objeto num arquivo como parte de sua instrução de limpeza do arquivo. O fato importante a ser lembrado é que você pode usar estas duas funções especiais para muitas finalidades, além de inicializar variáveis.

A Parte Ativa de uma Classe

Até aqui, esta discussão de classes não considerou realmente a implementação dos membros funcionais. Vamos nos dedicar agora às funções definidas como sendo parte destes dados complexos, determinados pelo programador. Tudo o que foi afirmado no Capítulo 2 sobre funções é igualmente verdadeiro para funções que sejam membros de uma classe. Examinaremos agora as características das definições *inline* e da sobrecarga de nome de função. Ambos os recursos são usados geralmente com funções-membro em uma classe, caso em que a sua sintaxe é diferente das funções não-membro comuns.

Funções inline em Classes

Lembre-se de que você pode especificar uma definição de função como *inline*, de maneira que qualquer chamada da função dentro de um arquivo é substituída não com um salto para a localização de memória que contenha as instruções da função, mas, sim, substituída pelas próprias instruções. Assim, se um arquivo chama três vezes a função *inline*, serão colocadas no arquivo três cópias de suas instruções. Estas funções *inline* proporcionam os benefícios de uma definição macro — que em C é simplesmente uma substituição textual — com a garantia total de uma chamada de função contendo variáveis locais e parâmetros totalmente articulados. Na realidade, o uso mais comum para as funções *inline* é como funções de membros de classes.

O exemplo *message3.h* (Listagem 4.4) ilustra dois tipos de definição *inline*. Você usa a definição *inline* padrão, ou explícita, com funções-membro, da mesma maneira como usaria com funções ordinárias. Aqui, a função destrutor *~message( )* é explicitamente *inline*. Observe que o membro é declarado na definição de classe, mas não é incluída nenhuma referência sobre como a função é definida. A função-membro *inline* é especificada como *inline* somente na definição real das próprias instruções da função.

Você pode também definir implicitamente uma função-membro *inline*. Se você inclui uma definição de função juntamente com a declaração do membro em

uma classe, então aquela função é implicitamente *inline*. No exemplo, as funções *clear( )* e *read_buf( )* são *inline* implícitas. A forma implícita de *inline* é mais usada por ser mais compacta.

Listagem 4.4 Uma classe *message* com funções-membro *inline*: *message3.h*

```
class message {
  char *s;              // um ponteiro para o conteudo da mensagem
  int len;              // o tamanho da mensagem

public:
  message(char*);       // declara um construtor
  ~message();           // retorna a memoria alocada pelo Sistema
                        // Operacional

  void send(char*);     // envia a mensagem para o seu destino
  void get(char*);                  // obtem uma dele
  void clear() { *s = '\0' };       // definicao inline implicita
  char* read_buf() { return s; }    // esta, tambem, e
                                    // implicitamente inline
};

inline message::~message()          // definicao inline explicita
{
  delete s; // retorna a memoria alocada ao banco de memoria livre
}
```

Observe que o uso aqui está de acordo com a boa prática de projeto, aplicado a definições *inline*. Somente funções muito simples — e, acima de tudo, pequenas — podem ser declaradas desta maneira. Devido ao fato de que as instruções que definem uma função *inline* tornam-se parte de cada solicitação da classe, funções *inline* muito grandes desperdiçam memória desnecessariamente e podem até tornar lento o acesso à classe. Funções longas e mais complicadas deverão ser tratadas como funções regulares. Este axioma é tão verdadeiro para funções-membro como o é para funções ordinárias.

Sobrecarregando Nomes de Funções-Membro

Os nomes de funções-membro têm a mesma propriedade da sobrecarga das funções ordinárias, tópico que foi discutido no Capítulo 2. Na verdade, os

programadores usam extensivamente a sobrecarga nas classes porque ela produz membros interface muito flexíveis. Você sobrecarrega uma função-membro implicitamente — não é necessária nem permitida uma declaração especial. Para criar uma função-membro sobrecarregada, simplesmente declare mais de uma definição com o mesmo nome.

O exemplo *date3.h* (Listagem 4.5) mostra uma função-membro sobrecarregada. Essa listagem é uma variação da classe *date* que foi definida no capítulo anterior (Listagem 3.8). Agora, no entanto, a função-membro que permite ao usuário resetar a data, *new_date( )*, está sobrecarregada. O usuário fornece a data para sua função, colocando três números inteiros: um para o mês, um para o dia, e um para o ano.O usuário pode também chamar *new_date( )* fornecendo uma cadeia de caracteres com o nome do mês em primeiro lugar, depois o dia, separados do ano por uma vírgula.

Este exemplo mostra a flexibilidade de uma função construtor sobrecarregada. Pelo fato dela fornecer dois métodos diferentes para inicializar esta classe, ela aumenta a generalidade da classe de maneira que pode ser usada em novos contextos de programas. Esta função construtor flexível torna também mais fácil para o projetista escrever o resto do programa e reutilizar módulos criados anteriormente. A reutilização de módulos é uma ferramenta importante de projeto e a sobrecarga de funções-membro aumenta a probabilidade de que duas classes que não foram idealizadas juntas nunca possam ser utilizadas juntas.

Este exemplo também usa uma função construtor com valores assumidos. Quando você declara uma variável do tipo *date*, devem ser fornecidos os mesmos três números inteiros necessários em *new_date( )* — na verdade, a função construtor solicita esta função para executar isso — pois, do contrário, os valores da classe serão definidos como zero.

Listagem 4.5 Uma nova classe *date* que utiliza uma função-membro sobrecarregada: *date3.h*

```
#include <stream.h>
#include <string.h>

#define COMMA ","
#define SPACE " "

const char* mname = {"mes incorreto","Janeiro","Fevereiro",
                     "Marco","Abril","Maio","Junho","Julho",
                     "Agosto","Setembro","Outubro","Novembro",
```

```
                    "Dezembro"};
// uma versao melhorada da classe date

class date {
  int day,      // esquema interno de armazenamento
      month,    // isto e agora privativo da classe
      year;     // e desconhecido externamente

public:
  date(int =0, int =0, int =0);// um construtor para a classe date
  void new_date(int,int,int);  // uma funcao membro para mudar a
                               // data
   void new_date(char *); // a mesma funcao com diferente formato
  char* give_date();           // ... e uma para relata-la
};

void date::date(int d, int m, int y)  // formato inteiro: igual
                                      // a 12, 28, 1987
{
  new_date(d, m, y);
}
void date::new_date(int d, int m, int y)  // toma os novos valores
{
  if((d>=1 && d<=31) && (m>=1 && m<=12)) { // verificar valores
  day = d;   // atribui os valores dos parametros para as
  month = m; // variaveis de armazenamento interno
  year = y;
}
void date::new_date(char* dat) // formato texto: igual a Dezembro
                               // 28, 1987
{
  char *mn,*dy,*yr;           // necessitamos alguns ponteiros
                              // para as subcadeias
  mn = strtok(dat,SPACE);     // usa a funcao de biblioteca
                              // strtok() para
  dy = strtok(0,COMMA);       // obter dia, mes e ano
  yr = strtok(0,COMMA);

  for (int i=1; i<=12; i++)
    if ( !strcmp(mn,mname[i])
      break;        // salta fora se tivermos o mesmo nome do mes
```

```
  if ( i<=12 ) {
    month = i;              // define os valores internos da classe
    day = atoi(dy);
    year = atoi(yr);
    }
  else   // coloca valor fantasma, no caso de nao haver igualdade
    month = day = year = 0;
}

char* date::give_date()
{
  char *buf = new char[80];

  if ( day==0 || month==0 || year==0 )    // verifica se a data
                                          // nao foi acertada
    sprintf(buf,"%s",mname[0]);
  else
    sprintf(buf,"%s %d,%d",mname[month],day,year);
  return buf;
}
```

Embora esta classe *date* revisada contenha um bom exemplo de funções-membro sobrecarregadas, você pode usar a sobrecarga com mais eficiência em outra situação. As funções construtor podem ser — e geralmente o são — sobrecarregadas para acomodar diferentes formatos para o mesmo tipo de dados. Para demonstrar esse uso, vamos expandir o exemplo *date*. Algumas vezes é conveniente armazenar uma data na forma de um número simples e positivo. Isso simplifica as comparações e os cálculos de duração de tempo. Você pode armazenar uma data como sendo o número de dias a partir de um evento conveniente (mas nem sempre significativo). Datas deste tipo são freqüentemente chamadas de datas julianas. O problema com este tipo arbitrário de datas é que às vezes elas contrariam a maneira como as pessoas vêem as datas. É um problema de conversão e de como se poderia movimentar convenientemente para frente e para trás entre dois tipos de representação de tempo. A solução é criar uma classe *new date* que acomoda diferentes métodos de definir a data. O exemplo *date4.h* (Listagem 4.6) implementa esta solução.

Listagem 4.6 Uma classe *date* que usa construtores sobrecarregados: *date4.h*

```
#include <stream.h>
#include <string.h>

// define alguns valores constantes uteis
```

```
const int months[]={0,31,59,90,120,151,181,212,243,273,304,334,365};

const char* mname = {"","Janeiro","Fevereiro","Marco",
                     "Abril","Maio","Junho","Julho",
                     "Agosto","Setembro","Outubro",
                     "Novembro","Dezembro"};

class julian {
  int days;          // numero de dias desde Janeiro 1

public:
  julian(int =0, int =0);      // primeiro construtor: igual a 12,3
  julian(char*, char*);        // alternativo: igual a Dezembro, 3

  char *current_date(); // exibe a data corrente no formato mes-dia
};
julian::julian(int mon, int day)
{
  days = months[mon-1] + day;
}
julian::julian(char* mon,char* dy)
{
  for (int i=1; i<=13; i++)        // encontra o mes correto
    if (!strcmp(mnames[i],mon)
      break;

  if (i > 12)                 // condicao de erro
    days = 0;
  else if (i == 1)            // e Janeiro, days e dy sao iguais
    days = atoi(dy);
  else
    days = months[i-1] + atoi(dy);     // calcula o numero de dias
                                       // desde Janeiro 1
}
char* julian::current_date()
{
  int mn,dy;

  if (days <= 31) {           // verifica se o mes e Janeiro
    mn = 1;
    dy = days;                // nao precisa calcular a data juliana
```

```
    }
  else
    for (int i=2; i<=12; i++)
    if (days <= months[i]) {
      mn = i;                    // define o numero de dias
      dy = days - months[i-1];   // calcula o dia
      break;
    }
  char *buffer = new char[20];

  sprintf(buffer,"%s %d,mnames[mn],dy);    // converte para
                                           // formato texto
  return buffer;                  // retorna a data como texto
}
```

Este exemplo define uma nova classe chamada de *julian*. As datas são armazenadas internamente no membro inteiro *days*. O programa interpreta uma data como sendo o número de dias a partir de 1 de janeiro do ano corrente (não são previstos os anos bissextos). Duas funções construtor oferecem a você duas maneiras de criar um objeto do tipo *julian*. Você pode introduzir dois números inteiros representando o mês e o dia, ou pode fornecer à função construtor uma cadeia de caracteres contendo o nome do mês e outra especificando o dia. A outra única função-membro definida é *current_date( )*, que converte os dias a partir de 1 de janeiro numa cadeia de caracteres formada pelo mês e o dia.

A codificação da função construtor na classe *julian* é bastante obscura e requer esclarecimento. A função construtor inicial:

```
julian(int =0, int =0);
```

converte seu argumento inicial no número de dias a partir de 1 de janeiro. O primeiro parâmetro representa o número do mês (janeiro é 1, fevereiro é 2, e assim por diante). Esse parâmetro identificador do mês deve então ser convertido no número de dias desde o começo do ano. A série *months* contém esta informação. Cada célula desta série contém o dia juliano (a partir de 1 de janeiro) do último dia de cada mês. Por exemplo, o dia juliano para 31 de janeiro é 31, mas o valor para 30 de junho é 181. A data atual é calculada tomando-se a data juliana do mês anterior e somando-se o dia do mês corrente. O dia juliano para 3 de julho, por exemplo, é 181+4, ou 185. A função construtor usa o parâmetro *mon* para encontrar na série o mês anterior, e depois acrescenta o outro parâmetro, *day*.

A outra forma da função construtor usa uma série similar, mas esta contém os nomes dos meses na devida ordem. Examine a série da cadeia de caracteres *mnames*. A cadeia "janeiro" ocupa a célula 1, "fevereiro" a célula 2, e o resto dos nomes segue na ordem apropriada. Essa função construtor converte o nome do mês em um número de mês e depois usa este valor para extrair a data juliana da mesma maneira que fez a primeira função construtor, mais simples. Esta correspondência é feita através do seguinte loop:

```
for(int i=1; i<13; i++)
   if(!strcmp(mnames [I], mon))
   break;
```

A função de biblioteca *strcmp( )* dá resultado 0 se as cadeias forem iguais, de maneira que este teste, para "not *strcmp()*" é na realidade um teste de igualdade. Observe que o loop vai além do número 12 para permitir a distinção entre "dezembro" e um nome de mês inválido. Mais adiante, no programa, a expressão *if-then-else-if* testa esta condição de erro e faz a variável *days* igual a zero se o usuário introduzir um nome de mês inválido. Caso contrário, o programa calcula o dia juliano de maneira semelhante à outra função construtor. É óbvio que poder criar tantas funções construtor quantos formatos de conversão existirem é uma ferramenta muito conveniente para o programador. No entanto, a propriedade de sobrecarga oferece outro recurso que vai além dessa conveniência. Por exemplo, você poderá usar a classe *julian* para servir de ponte entre tantos formatos diferentes de data quantos você queira. Uma vez que você acrescentou as funções-membro adequadas e depurou o programa, você não terá mais que se preocupar com a conversão explícita; o objeto faz isso para você. Conversões de software são sempre enganosas, geralmente confusas, e muitas vezes resultam num programa ilegível. Este procedimento dá a você os meios de mantê-la sob controle.

Questões de Revisão

1. É solicitada uma função construtor para uma classe sempre que um objeto da classe é _____.
2. Uma função destrutor executa qualquer tarefa necessária _____ como substituto de uma classe-objeto.

Projeto de Programação

1. Projete um sistema simples de mensagem transitória usando a classe em message.h.

Criando uma Função Amiga

Até agora, nossa discussão enfatizou o isolamento interno da classe. Porém, a sintaxe C++ tem também um mecanismo que permite que uma função não-membro tenha acesso à parte interna da classe. Declarando uma função como *amiga*, ela tem os mesmos privilégios de um membro da classe. A integridade da classe é mantida porque ela tem ainda uma interface restrita e definida. Porém, você pode agora permitir a entrada de estranhos selecionados. Naturalmente, você deve sempre ter o cuidado de não declarar tantas funções *amigas* que venham a comprometer a privacidade da classe. Porém, usada na medida certa, esta propriedade proporciona uma ferramenta de programação importante para criar software de fácil acesso.

O exemplo *timedat.c* (Listagem 4.7) mostra a sintaxe de uma definição *amiga*. Esse exemplo define duas classes simples — *time* e *date* — e inclui uma função motora simples que lhe permite explorar seu uso. Cada classe tem uma função construtor e uma função *amiga* — *gtime( )* para *time* e *gdate( )* para *date*. Cada função *amiga* formata e exibe os valores contidos na classe: *time* usa o formato *horas:minutos*; *date* usa o formato *dia/data/ano*. Observe que ambas as funções *amigas* também necessitam de um parâmetro do tipo apropriado. Note também que, mesmo quando se dá a uma função *amiga* um status especial quando ela acessa uma classe, ela não é parte daquela classe, e não está dentro do escopo da classe. Portanto, você deve passar a ela a classe-objeto sobre a qual ela agirá.

Listagem 4.7 Um programa que ilustra o uso de uma função *amiga*: *timedat.c*

```
#include <stream.h>
#include <string.h>

#define SEMICOLON ":"

////////////////////////////////////////////////////////////////
//   Define uma classe time simples com uma funcao "amiga"     //
////////////////////////////////////////////////////////////////

class time {
  long secs;          // o denominador comum para todos os formatos
                      // privativo da classe objeto
  friend char* gtime(time); // declara uma funcao de acesso externo
```

```
public:
  time(char *);       // acertando a hora do dia
};

time::time(char* tm)
{
  char *hr,*mn;
  hr = strtok(tm,SEMICOLON);     // separa a hora
  mn = strtok(0,SEMICOLON);      // ... e os minutos

  secs = atol(hr) * 3600 + atol(mn) * 60;     // calcula o total
                                              // de segundos
}

////////////////////////////////////////////////////////////////
//   Define uma classe date simples com uma funcao "amiga"    //
////////////////////////////////////////////////////////////////

class date {
  int month,          // armazena os valores da data
      day,
      year;
  friend char* gdate(date); // declara uma funcao de acesso externo

public:
   // um construtor simples
  date(int m, int d, int y) { month=m; day=d; year=y; }
};

////////////////////////////////////////////////////////////////
//              uma funcao driver simples                     //
////////////////////////////////////////////////////////////////

main()
{
  time x("10:30");          // cria um objeto time
  date d(12,29,1987);       // ... e um date

  char *gtime(time),        // declara funcoes "amigas"
       *gdate(date);

  cout << gtime(x) << "\n";    // acessa a classe time
  cout << gdate(d) << "\n";    // acessa a classe date
}
```

```
////////////////////////////////////////////////////////////////
//              define as funcoes "amigas"                    //
////////////////////////////////////////////////////////////////

char* gtime(time x)
{
  char *buffer;                  // cria um buffer
  buffer = new char[10];
  int h = x.secs / 3600,
     m = (x.secs % 3600) / 60;
  sprintf(buffer,"%02d:%02d",h,m);     // preenche-o com os
                                       // valores de tempo
  return buffer;
}

char* gdate(date x)
{
  char *buffer;              // cria um buffer e preenche-o com o
                             // formato desejado
  buffer = new char[15];
  sprintf(buffer,"%2d-%2d-%4d",x.month,x.day,x.year);
  return buffer;
}
```

Este exemplo mostra outras diferenças entre funções-membro e *amigas*. Declarando-se uma classe, declaram-se todos os membros daquela classe. No entanto, devido às funções *amigas* serem independentes, você deve declará-las da mesma maneira que declara qualquer outra função. No exemplo, ambas as funções dão como resultado uma cadeia de caracteres e são declaradas juntas. Naturalmente, a notação pontual usada pelas funções-membro não está disponível às funções *amigas* e você tem que fornecer os objetos explicitamente a elas. Este exemplo simples demonstra a sintaxe de uma declaração e definição *amiga*. No entanto, em ambas as classes, esta função de formatação e exibição poderia ter sido melhor manipulada por uma função-membro. Vamos ver alguns usos práticos para funções *amigas*.

Funções Amigas como Pontes entre Classes

Um papel importante que uma função *amiga* pode executar é agir como ponte entre duas classes diferentes. Neste caso, ela serve como um elo de ligação entre classes que não estão aninhadas. Embora se disponha de outros métodos para ligar classes, em muitos casos, as funções *amigas* oferecem uma ligação mais clara e segura. (Mais tarde, você verá que classes derivadas permitem a criação de hierarquia de classes.) Mostramos um exemplo simples desse tipo de "ponte amiga" em *timedat2.c* (Listagem 4.8).

Listagem 4.8 Um programa que ilustra uma função *amiga* usada como ponte entre duas classes: *timedat2.c*

```
#include <stream.h>
#include <string.h>

#define SEMICOLON ":"

class date;    // declara-a antes para que possa ser reconhecida
               // como um tipo

////////////////////////////////////////////////////////////////
//   Define uma classe time simples                           //
////////////////////////////////////////////////////////////////

class time {
  long secs;          // o denominador comum para todos os formatos
                      // privativo da classe objeto
  friend char *time_date(time,date); // declara uma funcao ponte

public:
  time(char *);       // acertando a hora do dia
};

time::time(char* tm)
{
  char *hr,*mn;
  hr = strtok(tm,SEMICOLON);      // separa a hora
  mn = strtok(0,SEMICOLON);       // ... e os minutos

  secs = atol(hr) * 3600 + atol(mn) * 60;      // calcula o total
                                               // de segundos
}
```

```
////////////////////////////////////////////////////////////////
//   Definindo uma classe date simples                        //
////////////////////////////////////////////////////////////////

class date {
  int month,          // armazena os valores da data
      day,
      year;
  friend char *time_date(time,date);   // declara a ponte

public:          // acertando a data - um construtor simples

  date(int m, int d, int y) { month=m; day=d; year=y; }
};


////////////////////////////////////////////////////////////////
//              uma funcao driver simples                     //
////////////////////////////////////////////////////////////////

main()
{
  time x("10:30");          // cria um objeto time
  date d(12,29,1987);       // ... e um date

  char *time_date(time,date);

  cout << time_date(x,d) << "\n";    // converte e exibe os valores
}
////////////////////////////////////////////////////////////////
//              define a funcao "amiga"                       //
////////////////////////////////////////////////////////////////

char *time_date(time t, date d)
{
  int h = t.secs / 3600,          // calcula o tempo
  m = (t.secs % 3600) / 60;
  char *buf;          // constroi um buffer com a saida desejada
  buf = new char[50];
  sprintf(buf,"hora: %02d:%02d\ndata : %2d-%2d-%4d",
          h,m,d.month,d.day,d.year);
  return buf;
}
```

Este exemplo declara a função *time_date( )* como *amiga* tanto para a classe *time* como para a classe *date*. Esta função "superamiga" formata a exibição de uma cadeia que contém informação da hora do dia e data e retorna para exibição. A única maneira pela qual ela pode fazer isso é acessando as partes internas de ambas as classes. Embora cada classe opere independentemente uma da outra e use sua própria estratégia interna de construtor e armazenamento, essa função *amiga* liga ambas as classes para uma finalidade comum. Você poderia também executar esta tarefa criando uma função regular com um parâmetro de hora e data. No entanto, neste caso, você precisaria fornecer duas classes-objeto à função e o acesso a seus valores internos requereria a chamada a uma função-membro. Claramente, o modelo da função *amiga* é mais eficiente.

O exemplo tem outro aspecto interessante. Note que você deve declarar *date* como uma classe antes de defini-la realmente. Se não fizer isso, a declaração de *time_date( )* na classe *time* será falha porque um dos seus parâmetros — *date* — está indefinido. Esta espécie de predefinição de tipos de classes é um conceito relativamente obscuro e que facilmente é passado por cima. No entanto, há outra maneira de evitar este problema — pelo uso de *void.* Lembre-se de que este tipo é compatível com um ponteiro para qualquer tipo de dado. O exemplo *timedat3.h* (Listagem 4.9) ilustra este modelo. Ele não tem problema de referência indefinida porque o parâmetro é declarado como um ponteiro para uma variável *void.* Posteriormente, na definição da função *time_date( )*, o parâmetro é jogado de volta para *date*; porém agora a classe *date* está definida e não é mais uma quantidade desconhecida.

Listagem 4.9 Uma função-ponte sem uma declaração prévia: *timedat3.h*

```
#include <stream.h>
#include <string.h>

#define SEMICOLON ":"

////////////////////////////////////////////////////////////////
//   Define uma classe time simples                           //
////////////////////////////////////////////////////////////////

class time {
  long secs;          // o denominador comum para todos os formatos
                      // privativo da classe objeto
  friend char *time_date(time*,void*); // declara uma funcao ponte
```

```
public:
  time(char *);      // acertando a hora do dia
};
time::time(char* tm)
{
  char *hr,*mn;
  hr = strtok(tm,SEMICOLON);      // separa a hora
  mn = strtok(0,SEMICOLON);       // ... e os minutos

  secs = atol(hr) * 3600 + atol(mn) * 60;        // calcula o total
                                                 // de segundos
}

////////////////////////////////////////////////////////////////
//    ... e uma classe date simples                           //
////////////////////////////////////////////////////////////////

class date {
  int month,          // armazena os valores da data
      day,
      year;
  friend char *time_date(time*,void*);    // declara a ponte

public:        // acertando a data - um construtor simples

  date(int m, int d, int y) { month=m; day=d; year=y; }
};


////////////////////////////////////////////////////////////////
//              define a funcao "amiga"                       //
////////////////////////////////////////////////////////////////

char *time_date(time* t, void* v)
{
  date *d = (date *)v;          // converte para um ponteiro data
  int h = t.secs / 3600,         // calcula o tempo
      m = (t.secs % 3600) / 60;
  char *buf;          // constroi um buffer com a saida desejada
  buf = new char[50];
  sprintf(buf,"hora: %02d:%02d\ndata : %2d-%2d-%4d",
          h,m,d->month,d->day,d->year);
  return buf;
}
```

Funções-Membro como Funções Amigas de Outras Funções

Todos os exemplos anteriores declaravam funções independentes como funções *amigas* a uma ou mais classes, oferecendo assim ligações externas com a estrutura interna destas classes. No entanto, as funções *amigas* não estão restritas a estas funções ordinárias. As funções-membro de uma classe podem também se tornar funções *amigas* de outras. Assim, como acontece no caso mais simples de função definida independentemente, é garantido acesso especial às funções-membro. Naturalmente, a diferença é que estas funções têm também acesso à parte privada de sua própria classe.

O programa *timedat4.c* (Listagem 4.10) demonstra a utilidade deste procedimento. Aqui, tanto *gdate( )*, que dá a data atual, quanto *gtime( )*, que dá a hora e o minuto, são membros da classe *date*, apesar de haver uma classe separada *time*. Definindo a classe *date* inteira como *amiga* para a classe *time*, você garante que *gtime( )* tenha acesso adequado à parte interna da classe *time*. Naturalmente, você poderia também incluir uma série inteira de funções-membro em uma classe para permanecer numa relação *amiga* com outra classe; você não está restrito a apenas uma função.

Listagem 4.10 Um programa que ilustra o uso das classes *amigas*: *timedat4.c*

```
#include <stream.h>
#include <string.h>

#define SEMICOLON ":"

class date;          // nao esqueca de declarar a classe date
                     // antecipadamente

////////////////////////////////////////////////////////////////
//   Define uma classe time simples com uma classe "amiga"    //
////////////////////////////////////////////////////////////////

class time {
  long secs;         // o denominador comum para todos os formatos
                     // privativo da classe objeto
  friend class date;    // todas as funcoes em date sao "amigas"
                        // em time

public:
```

```
  time(char *);          // acertando a hora do dia
};

time::time(char* tm)
{
  char *hr,*mn;
  hr = strtok(tm,SEMICOLON);        // separa a hora
  mn = strtok(0,SEMICOLON);         // ... e os minutos

  secs = atol(hr) * 3600 + atol(mn) * 60;      // calcula o total
                                               // de segundos
}

////////////////////////////////////////////////////////////////
//   define uma classe date simples com uma funcao "amiga"    //
////////////////////////////////////////////////////////////////

class date {
  int month,          // armazena os valores da data
      day,
      year;
public:
    // um construtor simples
  date(int m, int d, int y) { month=m; day=d; year=y; }
  char* gdate();              // um membro de acesso a classe date
  char* gtime(time);          // um membro que e uma funcao "amiga"
                              // para a classe time
};

////////////////////////////////////////////////////////////////
//   define as funcoes membro para a classe date              //
////////////////////////////////////////////////////////////////

char* date::gtime(time x)
{
  char *buffer;             // cria um buffer
  buffer = new char[10];
  int h = x.secs / 3600,          // calcula o tempo
      m = (x.secs % 3600) / 60;
  sprintf(buffer,"%02d:%02d",h,m);  // preenche-o com os valores
                                    // de hora
  return buffer;
}
```

```
char* date::gdate()
{
  char *buffer;      // cria um buffer e preenche-o com o formato
                     // desejado
  buffer = new char[15];
  sprintf(buffer,"%2d-%2d-%4d",month,day,year);
  return buffer;
}

////////////////////////////////////////////////////////////////
//              uma funcao driver simples                     //
////////////////////////////////////////////////////////////////

main()
{
  time x("10:30");          // cria um objeto time
  date d(12,29,1987);       // ... e um date

  cout << d.gtime(x) << "\n";    // acesso a classe time
  cout << d.gdate() << "\n";     // acesso a classe date
}
```

Embora as funções *amigas* raramente sejam essenciais à programação C++, elas servem para produzir uma codificação mais eficiente do que aquela que seria produzida dependendo exclusivamente de funções de membros de classes. Esta eficiência é particularmente óbvia quando você considera o tempo de processamento. Através de sua habilidade em acessar a parte interna de outra classe, as funções *amigas* permitem evitar as dificuldades em que incorreria outra função-membro. Este aumento na eficiência é particularmente importante se o seu programa tiver que alterar ou atualizar continuamente a parte interna da classe. Mesmo com esta vantagem, você deverá usar funções *amigas* com moderação. Do ponto de vista da filosofia C++ são preferidas ainda as funções-membro que definem a interface para uma classe.

Questões de Revisão

1. O acesso à parte interna de uma classe é restrito às funções-membro e às funções _____.
2. Uma função *amiga* pode servir como uma _____entre duas classes distintas.
3. Uma _____inteira pode ser declarada como *amiga* a uma classe.

Projeto de Programação

1. Acrescente uma função-membro à classe em date4.h para medir duração.

Classes e Ponteiros

A relação entre ponteiros e classes (e os membros das classes) é direta e deriva diretamente da relação que existe em C entre ponteiros e objetos de estrutura. Você pode declarar uma variável de ponteiro que contenha o endereço de uma classe-objeto. Você inicializa esta variável de duas maneiras diferentes: ou usando um operador *address-of* (&) ou solicitando o operador de alocação de memória, *new*. Este último método é usado mais comumente porque é mais flexível.

Após inicializar uma variável de ponteiro para uma classe-objeto, você pode empregar a notação da flecha para acessar os membros da classe. Por exemplo, considere a classe *message* definida anteriormente na Listagem 4.1. Você pode declarar uma classe *message* e inicializá-la com a seguinte instrução:

```
message msg ("isto é um teste");
```

Agora que você tem um objeto, pode declarar e inicializar um ponteiro para ele, assim:

```
message *ptr = &msg;
```

Agora, você pode acessar as funções-membro da classe através do ponteiro, Para exibir o conteúdo corrente do buffer *message*, por exemplo, coloque a linha seguinte:

```
cout << ptr->read_buf( );
```

Você pode usar o mesmo método para acessar qualquer outro membro na parte externa da classe.

Estrutura de Dados Concatenados Usando Classes

A estrutura de dados concatenados é um importante desafio para o programador porque ela experimenta os recursos da linguagem de implementação. Essa espécie de dados varia desde construções comuns, como árvores binárias, até redes multicadeia mais exóticas. A classe C++ tem características especiais que simplificam a criação da codificação que manipula estes tipos de dados concatenados.

A estrutura básica de dados concatenados, a lista concatenada, está ilustrada na Figura 4.1. A lista consiste em nós que podem ser armazenados em qualquer lugar na memória do sistema. Cada nó contém os dados associados a ele e o endereço do próximo nó. A implementação é igualmente direta. Cada nó pode ser implementado como uma classe-objeto, que é, naturalmente, a mesma implementação usada comumente em C. No entanto, duas características de C++ ajudam a simplificar a criação de tais nós — membros de classe estáticos e o ponteiro "this".

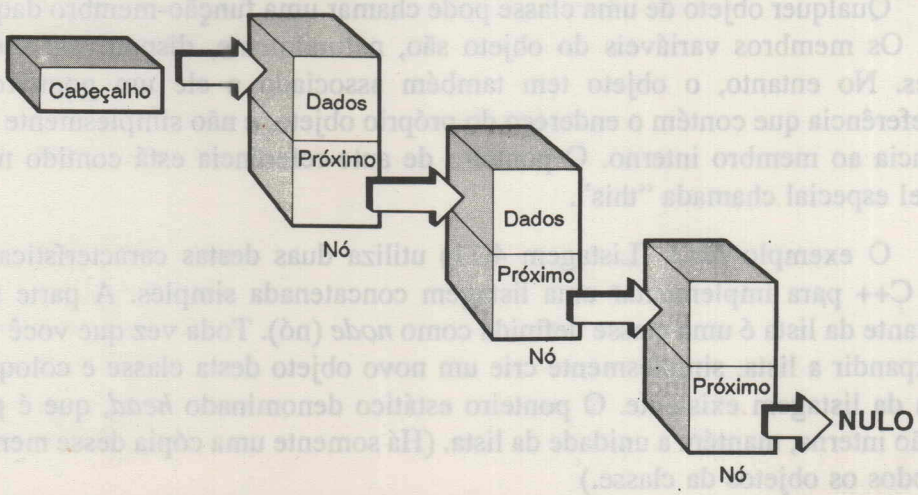


Figura 4.1 Uma estrutura de listagem concatenada.

Os membros variáveis de uma classe têm um escopo que está restrito aos objetos daquela classe. Os membros internos somente podem ser acessados pelas funções-membro ou *amigas*. No entanto, você pode também declarar estes membros variáveis como *estáticos*, o que os torna comuns a todos os objetos da classe. Um membro não estático, pelo contrário, é exclusivo para o objeto particular que é declarado. Esta relação está ilustrada na Figura 4.2.

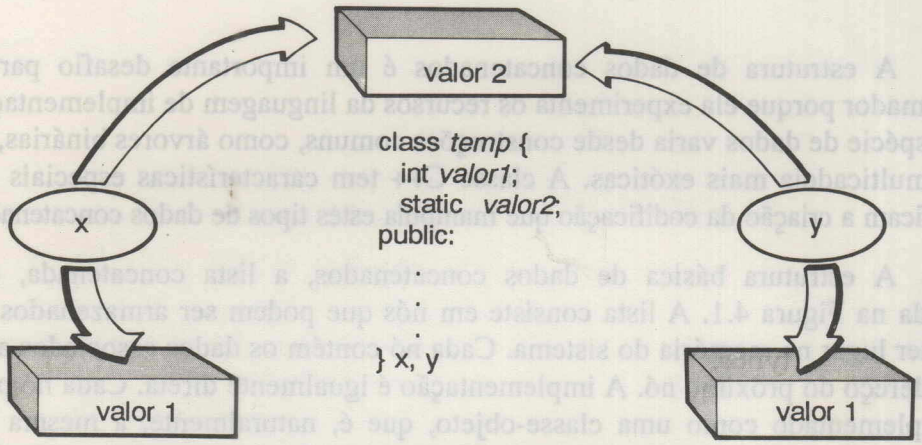


Figura 4.2 Membros de classe estática e não-estática.

Qualquer objeto de uma classe pode chamar uma função-membro daquela classe. Os membros variáveis do objeto são, naturalmente, disponíveis a estas funções. No entanto, o objeto tem também associado a ele um ponteiro de auto-referência que contém o endereço do próprio objeto, e não simplesmente uma referência ao membro interno. O ponteiro de auto-referência está contido numa variável especial chamada "this".

O exemplo *list.h* (Listagem 4.11) utiliza duas destas características da classe C++ para implementar uma listagem concatenada simples. A parte mais importante da lista é uma classe definida como *node* (nó). Toda vez que você tiver que expandir a lista, simplesmente crie um novo objeto desta classe e coloque-o no fim da listagem existente. O ponteiro estático denominado *head*, que é parte da seção interna, mantém a unidade da lista. (Há somente uma cópia desse membro para todos os objetos da classe.)

Listagem 4.11 Uma classe de uma listagem concatenada simples: *list.h*

```
#include <string.h>

class node {
  static node *head;  // um ponteiro para a cadeia de todos os
                      // membros da classe
  node *next;         // a uniao com o proximo no
  char *info;         // dados do no

public:
  node(char* =0);     // declara um construtor
  void display_all(); // mostra cada no da cadeia
};

node::node(char* ptr)     // o construtor
{
  if (ptr != 0) {                  // nao e o primeiro no da lista
    info = new char[strlen(ptr)+1]; // reserva area para o valor
    strcpy(info,ptr);              // ... e o copia
    next = 0;                      // estabelece-o como o ultimo elo
    node *cursor = head;           // cria um cursor
    while (cursor->next != 0)      // busca o final da lista
      cursor = cursor->next;

    cursor->next = this;           // atribui a lista o no corrente
    }
  else {                                // e o primeiro no da lista
    info = new char[strlen("root")+1];   // estabelece os dados
    strcpy(info,"root");           // assinala-o como o primeiro no
    next = 0;                       // estabelece-o como ultimo elo
    head = this;                   // atribui o no a head
    }
}

void node::display_all()      // uma funcao de exibicao simples
{
  node *cursor = head; // define uma variavel ponteiro temporaria
  while (cursor != 0) {                    // percorre a lista
    cout << form("*%s\n",cursor->info);    // ... exibindo cada no
    cursor = cursor->next;
    }
}
```

Cada vez que você cria um novo nó, a função construtor da classe inicia no membro *head* (ponteiro). Devido a este membro ser estático, seu escopo é comum à classe inteira de objetos; portanto ele aponta para o primeiro nó, e não para o nó que está sendo criado. Toda vez que se percorre esta lista, move-se do primeiro para o último membro. Se você declarou o membro *head* como um membro qualquer, sua referência estará restrita ao nó corrente, tornando assim impossível a criação da lista.

A Figura 4.3 ilustra outra característica importante, ainda que sutil, da classe C++. No presente contexto, você precisa também de uma maneira de se referir ao nó recentemente criado — ao objeto da classe como um todo, não simplesmente a seus membros — com o objetivo de colocá-lo no fim da lista. Para fazer isso, você precisa usar o ponteiro *this*. Lembre-se de que o ponteiro *this* sempre aponta para o objeto atual da classe. Portanto, ele contém o endereço do nó-objeto criado por último.

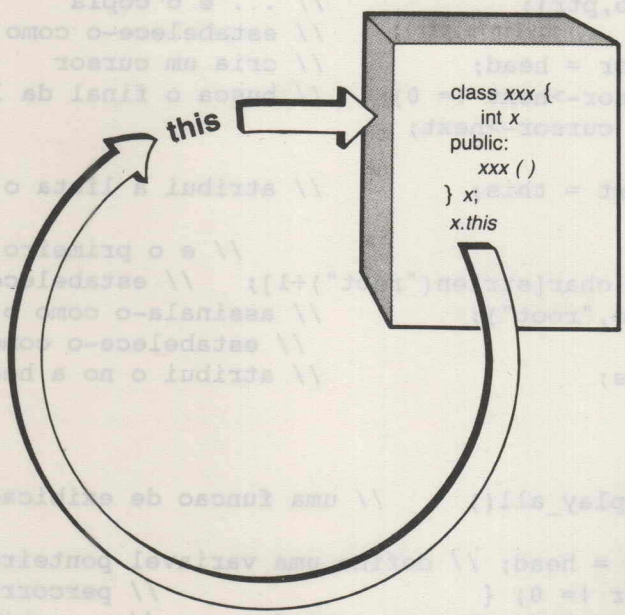


Figura 4.3 O ponteiro *this*.

Classes Container

Uma variação importante da lista concatenada descrita anteriormente é a classe *container*. Basicamente, um container é uma espécie de dados que consiste em uma coleção dinâmica de valores — a tradicional tabela de dados, por exemplo. Embora a criação e o uso destas espécies sejam partes importantes de toda a programação, a espécie *classe* C++ torna sua implementação particularmente simples.

O exemplo *contain.c* (Listagem 4.12) define uma classe *token* (ficha) ou, mais corretamente, uma classe "tokenizing" (fichário). Na criação de um objeto dessa espécie, a função construtor aceita uma cadeia de caracteres como parâmetro. A classe *token* contém uma seqüência de "fichas" incluindo as inicializações das cadeias de caracteres que as compõem. Neste caso, uma ficha é definida como sendo um ou mais caracteres contíguos separados por espaços em branco. A interface primária é através da função-membro *nth_token( )*. Essa função toma um número ordinal e dá como resultado a ficha apropriada na lista.

Listagem 4.12 Um exemplo de classe container: *contain.c*

```
// inclusao dos arquivos header necessarios

#include <stream.h>
#include <string.h>

#define SPA " "

////////////////////////////////////////////////////////////////
//  cria uma classe value para armazenar uma unica ficha
////////////////////////////////////////////////////////////////

struct tvalue {    // inicializa os valores da classe token
  char *value;    // valor cadeia de caracteres
  tvalue *next;   // proxima ficha da sequencia
  };

////////////////////////////////////////////////////////////////
// define uma classe token para conter uma sequencia de fichas
////////////////////////////////////////////////////////////////

class token {
  static tvalue *toke; // ponteiro de ligacao da sequencia de fichas
```

```
public:
  token(char *);    // construtor
  ~token();         // destrutor

  char *nth_token(int); // acesso a fichas individuais da sequencia
};
////////////////////////////////////////////////////////////////////
//  define o construtor e destrutor
////////////////////////////////////////////////////////////////////
token::token(char *x)
{
  char* temp;

  if ((temp=strtok(x,SPA)) == 0) {    // testa condicao de erro:
                                      // sem fichas
    cout << "condicao de erro\n";
    exit(1);
  }
  toke = new tvalue;      // cria e inicializa a primeira ficha
  tvalue *cursor = toke;
  cursor->value = new char[strlen(temp)+1];
  strcpy(cursor->value,temp);
  cursor->next = 0;

  while ((temp=strtok(0,SPA)) != 0) {    // usa strtok() para
                                         // separar as fichas
    cursor->next = new tvalue;  // cria um novo no e inicializa-o
    cursor = cursor->next;
    cursor->value = new char[strlen(temp)+1];
    strcpy(cursor->value,temp);
    cursor->next = 0;
    }
}

token::~token()
{
  tvalue *prev, *cursor=toke;

  while (cursor != 0) {
    prev = cursor;            // percorre a sequencia de fichas
    cursor = cursor->next;  // elimina cada no antes de prosseguir
    delete prev->value;
```

```
    delete prev;
    }
}
////////////////////////////////////////////////////////////////
// define a unica funcao membro
////////////////////////////////////////////////////////////////

char* token::nth_token(int num)
{
  tvalue *cursor = toke;  // cria uma lista de cursor concatenada

  for (int i=1; i<num; i++)       // passo a passo, "num" vezes
    if (cursor->next != 0)        // se nao for final da lista
      cursor = cursor->next;      // va para o proximo
    else
      return '\0';              // opa, "num" muito alto
    return cursor->value;         // retorna a "num"esima ficha
}

////////////////////////////////////////////////////////////////
// uma funcao driver simples para demonstrar a classe
////////////////////////////////////////////////////////////////

main()
{
  token x("isto e apenas um teste");      // cria um objeto token

  for (int i=4; i>=1; i--)          // mostra a sequencia de fichas
                                    // em ordem inversa
    cout << x.nth_token(i) << "\n";
}
```

Faz sentido definir a classe *tvalue* como uma estrutura. Na verdade ela não necessita ter uma parte interna e externa porque não tem membros de funções e está totalmente contida na classe *token*. Numa listagem encadeada simples, isso seria a definição de nó. Uma variável tipo cadeia de caracteres contém o valor e o elo para o próximo nó na lista. Esta é a estrutura que será usada para construir a lista concatenada.

O único membro interno da classe *token* é o ponteiro para a lista de *tvalues*. A maior parte do trabalho desta classe é feita pela função construtor simples, que toma uma cadeia de caracteres como variável inicializadora *token x ("isto é apenas*

um teste"). Se você não fornecer esta cadeia de caracteres na declaração, resultará uma condição de erro, e não será criado nenhum dado-objeto. Se, no entanto, for colocada em *token( )* uma cadeia de caracteres corretamente formatada, então a função-membro cria uma lista encadeada, na qual cada nó é uma única *palavra* na cadeia. A função biblioteca-padrão *strtok( )* faz a análise gramatical (embora você poderia facilmente escrever esta função *strtok( )* em C++).

A chamada inicial da função *strtok( )* é feita na construção condicional que testa a presença de uma cadeia:

```
if((temp=strtok(x,SPA)) == 0) {
```

Note que o primeiro nó da lista é criado e preenchido independentemente do loop que separa as outras palavras da lista. Isso é feito principalmente porque, fazendo o teste, o programa já agarrou a primeira ficha. As *palavras* subseqüentes são colocadas nos nós por chamadas repetidas a *strtok( )*:

```
while((temp=strtok(0,SPA)) ! = 0)
   cursor->next=new tvalue;
   cursor=cursor->next;
   cursor->value=new char [strlen(temp)+1];
   strcpy(cursor->value,temp);
   cursor->next=0;
}
```

Estas chamadas no loop continuam até que a função retorne um zero para indicar o fim da cadeia. A execução da função construtor deixa o dado-objeto do tipo *token* que consiste em uma lista encadeada de "fichas".

A função-membro *nth_token( )* retorna o valor do nó especificado na lista encadeada. Assim, uma chamada à função *nth_token(3)* retorna o valor da terceira palavra na lista. Um simples loop *for* percorre a lista até encontrar o nó desejado. Se *nth_token( )* é chamada com um número muito alto, ela retorna um sinal *fim da cadeia ('\0')* para indicar a falha.

A função destrutor para a classe *token* é imediata, apesar de um pouco difícil de manejar. Ela percorre a lista encadeada e desaloca cada nó. Se você examiná-la mais atentamente, notará que ela remove nós atrás de si própria — um tanto estranho, mas razoavelmente eficiente.

Em contraste com o exemplo anterior na Listagem 4.11, essa classe consiste em duas classes aninhadas. Uma classe, *tvalor,* define uma lista encadeada

simples. As diferenças entre esta classe e a classe anterior definem a noção de uma classe container. Numa lista encadeada tradicional, cada nó é uma instância da classe. No entanto, numa classe container a lista inteira está contida num objeto-classe. A Figura 4.4 ilustra a diferença.

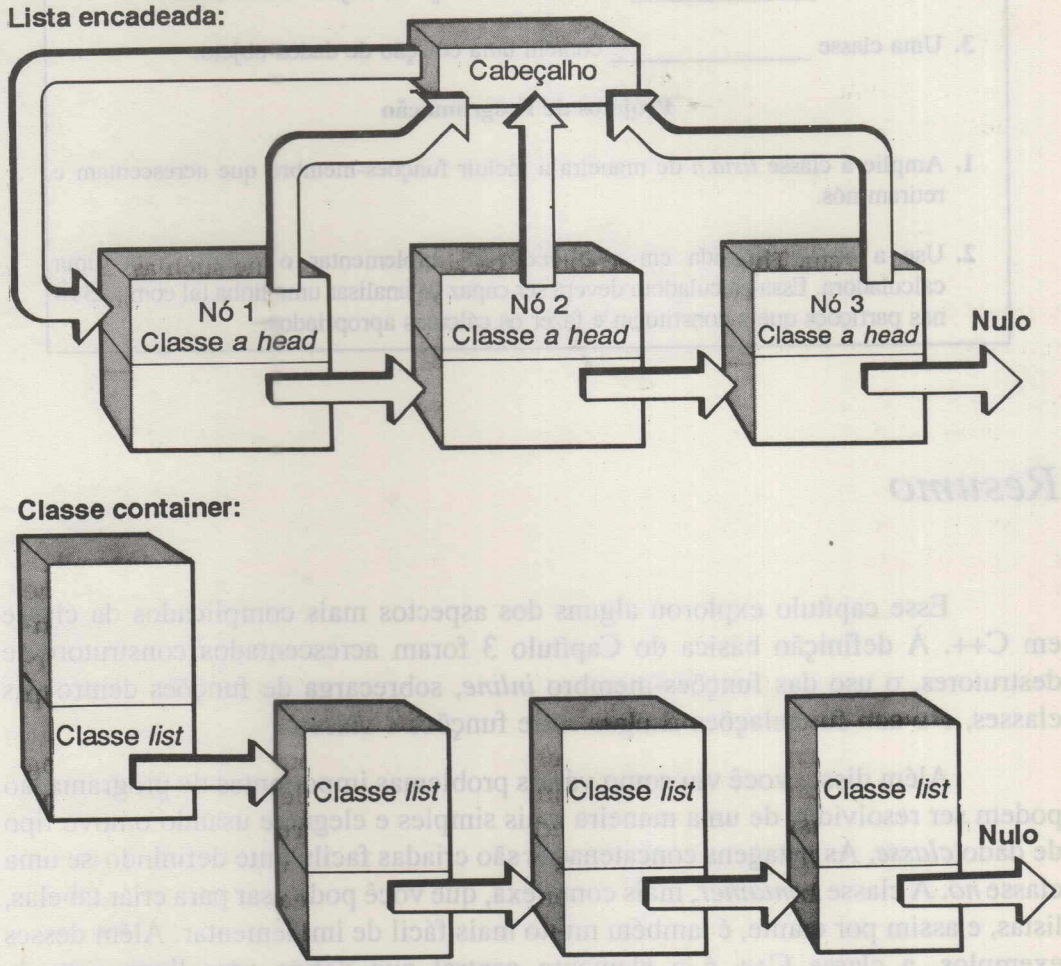


Figura 4.4 Listagens encadeadas *versus* classes container.

Questões de Revisão

1. Uma variável-membro _____ é comum à classe inteira.
2. O ponteiro _____ contém o endereço do objeto-classe atual.
3. Uma classe _____ contém uma coleção de dados-objeto.

Projetos de Programação

1. Amplie a classe *lista.h* de maneira a incluir funções-membro que acrescentam e retiram nós.
2. Use a classe definida em *contain.c* para implementar o programa de uma calculadora. Essa calculadora deverá ser capaz de analisar uma linha tal como 23+4 nas partições que a constituem e fazer os cálculos apropriados.

Resumo

Esse capítulo explorou alguns dos aspectos mais complicados da classe em C++. À definição básica do Capítulo 3 foram acrescentados construtores e destrutores, o uso das funções-membro *inline*, sobrecarga de funções dentro das classes, e o uso das relações amigas entre funções e classes.

Além disso, você viu como vários problemas importantes de programação podem ser resolvidos de uma maneira mais simples e elegante usando o novo tipo de dado *classe*. As listagens concatenadas são criadas facilmente definindo-se uma classe *nó*. A classe *container*, mais complexa, que você pode usar para criar tabelas, listas, e assim por diante, é também muito mais fácil de implementar. Além desses exemplos, a classe C++ é o elemento central que define essa linguagem de programação melhorada.

Makron
Books

Capítulo 5
Sobrecarregando Operadores

Uma revisão da sobrecarga
A mecânica da sobrecarga de operador
Sobrecarregando o símbolo ( )
Uma classe string bidimensional
Resumo

O Capítulo 3 tratou da modelagem de entidades e situações complexas do mundo real. A solução C++ vai em direção a uma espécie de dados complexa chamada classe. O Capítulo 4 detalhou a maneira como as classes interagem com o resto de C++ para produzir programas úteis e confiáveis. Até agora, no entanto, estas discussões levaram somente a uma solução parcial do problema de modelagem do mundo real.

Agora, vamos nos dirigir ao restante das proposições levantadas pela definição de classes. Por exemplo, uma das fraquezas das linguagens de programação tradicional ocorre quando elas precisam lidar com construções de nível superior, tais como as estruturas. Devido a estas novas espécies de dados serem criadas pelo programador, elas não podem ser manipuladas com os mesmos operadores que funcionam com os tipos internos. Em C, por exemplo, você não pode somar duas estruturas, mesmo que a operação seja legítima, como seria no caso de estruturas que definem números complexos. Você pode definir uma função equivalente que executa a adição em estruturas particulares, mas não pode juntar magicamente aquela função ao operador "+". Assim, para executar a operação, você precisa usar o mecanismo de chamada de função em lugar da simples sintaxe

de operadores. Por exemplo, suponha que você tenha definido o seguinte tipo de número complexo em C:

```
struct complex {
    double real,
           imag;
} x,y,z;
```

Para tornar o programa apto a somar dois números complexos, por exemplo, você precisa definir uma função do tipo *addcplx( )*. A verdadeira soma, então, fica parecida com isso:

```
z = addcplx ( x, y);
```

A maioria dos programadores está acostumada com este tipo de circunlóquio. No entanto, pense como ficaria muito mais clara a codificação desta tarefa, se ela fosse escrita simplesmente assim:

```
z = x + y;
```

Como você pode esperar, C++ pretende suprir esta deficiência de C. Ela acrescenta flexibilidade aos seus programas, permitindo-lhe redefinir os operadores existentes de maneira que eles funcionem também com objetos de classes. Você pode não somente dar novas características aos símbolos-padrão, tais como +, * e +=, mas ainda redefinir os símbolos de subscrição e o operador de chamada de função ( ). Por ser também possível sobrecarregar os operadores *new* e *delete* para gerenciamento de memória, você ganha a habilidade de criar rotinas de alocação e limpeza de memória.

Para que a sobrecarga seja eficiente, as ações associadas ao objeto devem ter as mesmas relações íntimas com o valor como aquelas dos dados internos. Por exemplo, quando você soma dois números inteiros usando o operador +, o programa que implementa aquela operação de soma sabe como um número inteiro é armazenado e como manipulá-lo. Você não precisa escrever todo o código que realiza esta tarefa, ponto por ponto. C++ dá a mesma flexibilidade aos tipos de dados criados pelo programador. O objetivo do mecanismo de sobrecarga de operador é curto-circuitar a conexão externa entre dado e ação que se encontram nas linguagens tradicionais de programação, formando assim uma conexão mais íntima entre estes dois aspectos do objeto.

Este capítulo explora as propriedades de sobrecarga de operador de C++, oferecendo exemplos práticos e ilustrativos das técnicas úteis. Por exemplo, os

programas neste capítulo mostram-lhe como usar operadores sobrecarregados para simplificar a manipulação de:

- números complexos
- cadeias de caracteres
- uma classe círculo
- dados do tipo cruzeiros e centavos

Cada exemplo demonstra como substituir a sintaxe de chamada de função por um simples e evidente operador.

Revisão de Sobrecarga

O Capítulo 2 discutiu a base da sobrecarga de função, e o Capítulo 4 explicou como usá-la para obter funções flexíveis para classes definidas pelo usuário. Por exemplo, um programa (Listagem 4.6) mostrou como usar uma função construtor sobrecarregada para criar objetos de classe com uma variedade de formatos de dados.

Sobrecarga de Função

Antes de discutir a sobrecarga de operador, vamos rever rapidamente o mecanismo da sobrecarga. Várias funções com o mesmo nome podem representar diferentes trechos de programa. Por exemplo, uma função *mean( )* pode manipular valores *double*, enquanto outra, também *mean( )*, pode calcular a média de números inteiros. O programa decide qual destas funções é especificada por uma particular chamada de função, confrontando o tipo de argumento verdadeiro com as duas definições da função. Por exemplo, uma solução muito simples para um problema permanente: você pode especificar duas funções *mean( )* — uma para valores *long* e outra para valores *double*:

```
overload mean;
long mean(long);
double mean(double);
```

Cada versão de *mean( )* deve ter uma definição diferente, de maneira que o programa possa escolher a codificação correta. Neste caso, a seleção é baseada nos tipos de dados dos parâmetros para a função. No seguinte trecho de programa:

```
int x,m;
double y, rm;
   .
   .
m = mean(x);
rm = mean(y);
```

A versão *long* de *mean( )* serve à variável *m* porque o parâmetro *x* é *long*. Da mesma forma, *rm* acessa a função apropriada para *double* porque *y* é *double*.

Você tem muitas vantagens usando funções sobrecarregadas. Primeiramente, você pode dispensar a codificação inconveniente que é necessária para escolher a implementação de uma função sobre outra. Em segundo lugar, você não precisa mais dar nomes esquisitos às funções, como *fmean( )* e *imean( )* — e colocá-las em alguma instrução *if* que usa uma expressão condicional complicada para escolher uma ou outra. Em terceiro lugar, as listagens do seu programa-fonte tornam-se claras e fáceis de ser lidas porque você pode substituir os meandros tortuosos das expressões tradicionais complexas por codificações mais diretas.

Estendendo a Sobrecarga aos Operadores

A sobrecarga de operador permite que o programador dê novos significados aos símbolos de operador já existentes em C++. Por exemplo, você pode construir sua própria implementação de + ou —, o incremento e decremento de operadores (++ e —) ou mesmo a operação de alocação de memória *new* e *delete*. Lembre-se, também, de que quando você executa a sobrecarga a funcionalidade existente dos operadores C++ é sempre preservada; por exemplo, ++ continua sendo o incremento de um número inteiro, como esperávamos.

A sobrecarga de operador — dar significados múltiplos ao mesmo símbolo — não é tão incomum como pode parecer à primeira vista. Toda linguagem de

programação utiliza este procedimento, até certo ponto, mas, devido à sobrecarga ser um processo interno, você talvez não se dê conta de que ele está sendo usado. Por exemplo, o + que soma dois números inteiros não é na realidade a mesma coisa que o + que soma dois valores *double*. Tanto em C como em C++, muitos operadores básicos são também redefinidos para executar completamente funções que não são cálculos aritméticos. Por exemplo, o operador *, dependendo do contexto, pode ser um operador de multiplicação, ou representar um referenciador indireto de uma variável de ponteiro, permitindo-lhe obter os dados do endereço apontado. A novidade em C++ é que esta propriedade de sobrecarga se estende aos dados definidos pelo usuário — as classes.

No entanto, há algumas restrições ao uso da sobrecarga de operador. A redefinição é confinada aos símbolos do operador existente — você não pode construir uma sintaxe de operador totalmente nova. Os operadores podem ser sobrecarregados somente no contexto das definições de classe. Você não pode, por exemplo, redefinir para inteiros ou qualquer outra espécie de dados internos. Você não pode também alterar a posição que o operador ocupa na ordem de precedência. Para finalizar, a capacidade para definir novos operadores para ponteiros é limitada. No entanto, mesmo com estas restrições, a sobrecarga de operador é uma ferramenta poderosa para a estruturação de programa.

Vantagens da Sobrecarga de Operador

A principal vantagem de se poder definir novos operadores para trabalhar com classes é que isso ajuda a estabelecer tipos de classe como partes totalmente funcionais da linguagem de programação C++. A sobrecarga de operador permite que C++ se torne uma linguagem extensível num sentido ainda mais profundo: o programador pode acrescentar à linguagem espécies de dados especializados que promovem uma melhor representação dos dados, além da modularidade, e são ainda tão fáceis de usar quanto os operadores básicos embutidos.

Eliminando a distinção entre os tipos definidos pelo usuário e os tipos tradicionais, C++ também abre as portas a uma codificação mais eficiente. Agora você pode ampliar as operações naturais para manipular situações similares, porém não idênticas. Por exemplo, você pode redefinir o operador + — usado tradicional-

mente para indicar a adição de números — para juntar cadeias de caracteres. Em C, você somente pode executar esta junção usando uma função da biblioteca-padrão:

```
char s1[10] = "ABC",
     s2[] = "DEF",
     s3[20];
     s3 = strcat(s1,s2);
```

Aqui, a junção é executada através da chamada da função, de modo que você deve ter cuidado para que a variável da cadeia receptora tenha espaço suficiente para armazenar ambas as cadeias. Em C++, por outro lado, você pode redefinir o operador + para representar esta mesma operação:

```
char s1[10] = "ABC",
     s2[] = "DEF",
     s3[20];
s3 = s1 + s2;
```

Esta codificação pode executar a mesma tarefa, mas a sintaxe do operador é mais clara e não é ambígua. E, o que é mais importante, a função da codificação é mais evidente; a concatenação, ou junção, num sentido mais amplo, é semelhante ao conceito aritmético da adição. Muitos programadores que virem o operador usado neste contexto entenderão, quase instintivamente, o significado da operação.

A sobrecarga de operador então permite-lhe criar "metáforas" de programação, como é o caso da junção de operador. A Figura 5.1 ilustra este conceito. Uma metáfora é simplesmente uma extensão do conceito operacional de uma espécie de dados para outra espécie diferente, porém análoga. No caso da junção, a noção de adição numérica ganha um significado semelhante no contexto das cadeias de caracteres.

Note que esta extensão do significado da soma não é simplesmente um procedimento mecânico — a operação de somar dois números é diferente da operação de juntar duas cadeias. Portanto, a metáfora de programação permite que o programador mude de perspectiva e veja os dados, particularmente aqueles mais complexos, por um novo ângulo.

Muitos operadores C++ estão desiludidos com a sobrecarga de operador. Eles têm a forte impressão de que esta é uma técnica que pode ser facilmente mal usada. Alguns até acham que a especificação e implementação desta parte da linguagem precisa ser melhorada. Sem tomar partido nesta discussão, é certamente verdade que o programador C++ principiante deve tratar a sobrecarga de operador

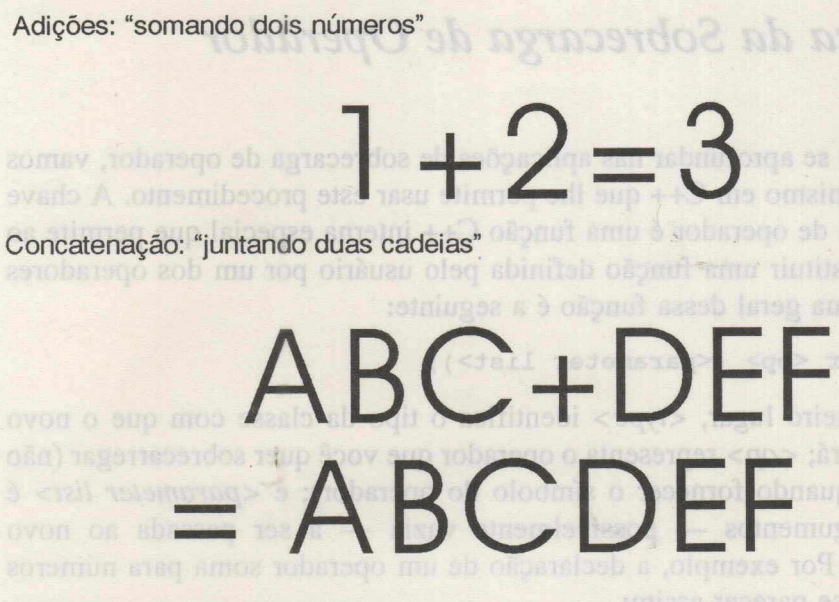


Figura 5.1 Metáforas de programação.

com muito cuidado. Ela mostra ser poderosa, mas não tão bem comportada ou controlável como parece — uma combinação perigosa. Ainda em relação à sobrecarga de função, o principiante pode facilmente produzir uma codificação que é obscura, chegando mesmo a ser opaca — não somente ao leitor, mas até mesmo para o autor.

Até mesmo os programadores experimentados correm risco. O uso e significado desses operadores pode tornar-se tão arraigado que, se o significado de um deles é mudado numa parte do programa, pode causar confusão sobre o significado de um outro. Isso é um problema quando você usa com frequência o significado original de um operador dentro da função que o redefine.

O uso mais seguro da sobrecarga de operador é com números. A definição de números complexos como classe e a sobrecarga dos operadores aritméticos usuais é um exemplo comum e bem-comportado de um uso claro e moderado desse recurso. No entanto, nem todas as possíveis situações de sobrecarga serão úteis. Tenha isso em mente quanto estiver esboçando seus programas C++.

A Mecânica da Sobrecarga de Operador

Antes de se aprofundar nas aplicações de sobrecarga de operador, vamos examinar o mecanismo em C++ que lhe permite usar este procedimento. A chave para a sobrecarga de operador é uma função C++ interna especial que permite ao programador substituir uma função definida pelo usuário por um dos operadores existentes. A forma geral dessa função é a seguinte:

```
<type> operator <op> (<parameter list>);
```

Em primeiro lugar, *<type>* identifica o tipo da classe com que o novo operador trabalhará; *<op>* representa o operador que você quer sobrecarregar (não coloque espaço quando fornecer o símbolo do operador); e *<parameter list>* é uma lista de argumentos — possivelmente vazia — a ser passada ao novo operador-função. Por exemplo, a declaração de um operador soma para números complexos pode se parecer assim:

```
complex operator + (complex);
```

Embora este exemplo use apenas um argumento, a lista de parâmetros não está restrita a uma só variável. Lembre-se, a declaração de um novo operador deve ser seguida por sua definição. A sintaxe de definição de uma função operador é a mesma que para qualquer outra função-membro de classe: um cabeçalho identifica a classe e o membro em particular, como no exemplo:

```
complex::complex operator + (complex x)
```

e é seguido do código apropriado.

Após esta definição ser compilada, sempre que o programa se refere ao operador especificado, uma chamada a esta função operador executa estas instruções. Por exemplo, a codificação a seguir:

```
main( )
{
complex x(2,3),
y(4,6);
x + y;
}
```

deixa a variável *x* com um valor real igual a 6 e um valor imaginário igual a 9.

O exemplo *cmplx.h* (Listagem 5.1) ilustra como se redefine um operador. Esta definição de um número complexo inclui uma função construtor, um operador de adição e uma função de exibição. (Devido à atenção se concentrar sobre a definição do operador, neste caso, o exemplo omite a função destrutor e outras funções-membro potencialmente úteis.) Lembre-se de que cada número complexo consiste em duas partes — uma parte real e uma parte imaginária. Cada uma delas é representada por um valor *double* armazenado na seção interna da classe *complex*. A soma de números complexos envolve a soma das partes real e imaginária independentemente, para produzir um novo valor composto por duas partes. No exemplo, você tem que inicializar um dado-objeto com um valor na forma de *mm+nni* (você pode colocar quaisquer números nesta expressão). A Figura 5.2 ilustra a forma de um número complexo.

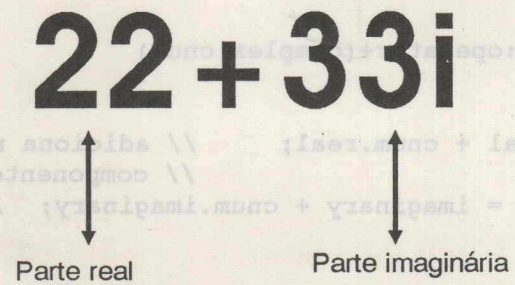


Figura 5.2 Um número complexo.

Observe que, para conveniência, a expressão de inicialização é uma cadeia de caracteres que o programa converte nos valores numéricos apropriados usando funções de cadeia de caracteres da biblioteca-padrão. Especificamente, *strtok( )* divide a expressão da cadeia em duas partes através do caractere +.

O programa assume que a parte real está do lado esquerdo desse operador e que a parte imaginária está do lado direito. Como verificação posterior, o programa chama novamente a função *strtok( )* para procurar o caractere *i* que tradicionalmente indica a parte imaginária de um número complexo. O programa então usa a função-biblioteca *atof( )* para converter ambas as cadeias em números.

Listagem 5.1 Uma classe de números complexos que usa um operador + sobrecarregado: *cmplx.h*

```
#include <string.h>
```

```
class complex {
  double real,          // armazena a parte real do numero complexo
         imaginary;     // ... e a parte imaginaria
public:
  complex(char*);  // declara um construtor
  complex operator+(complex&);      // declara o operador "+" para
                                    // numeros complexos
  char* display(); // nao tem graca se voce nao pode ver o valor
};
complex::complex(char* cnum)
{
  real = atof(strtok(cnum,"+")); // separa e converte a parte real
  imaginary = atof(strtok(0,"i")); // converte a parte imaginaria
}

complex complex::operator+(complex cnum)
{
  complex temp;
  temp.real = real + cnum.real;      // adiciona no novo
                                     // componente real
  temp.imaginary = imaginary + cnum.imaginary;  // e a imaginaria
  return temp;
}

char* complex::display()
{
  char* temp;      // cria um espaco de buffer temporario
  temp = new char[10];
  sprintf(temp,"%f + %fi",real,imaginary); // preenche e formata-o
  return temp;                             // retorna o valor
}
```

Esse exemplo focaliza a função do operador, que é declarado para redefinir o símbolo + para executar a adição de dois números complexos. Esta redefinição é muito simples. O número complexo que é fornecido como parâmetro é somado, componente por componente, à classe *complex* que o solicitou.

A ação desta função operador é implementada pelas seguintes instruções de sua definição:

```
temp.real = real + cnum.real;
temp.imaginary = imaginary + cnum.imaginary;
```

Um dos números complexos é fornecido à função como parâmetro *cnum*. A parte interna da primeira classe fornece o outro valor. Esta classe é a classe solicitante, embora sua posição seja obscurecida pela sintaxe. O programa soma as duas partes reais e depois as duas partes imaginárias. Em ambos os casos, o resultado é armazenado na variável *temp*. No fim da operação, esse valor temporário é retornado à função que solicitou.

Note que o símbolo redefinido — + — é acrescentado à palavra-chave *operator* e está fora dos parênteses que delimitam a lista de argumentos da função. Isso acontece porque o símbolo é parte do nome da função e não um parâmetro dela. No exemplo, o parâmetro é outro número complexo que deve ser somado ao valor do objeto *complex* corrente, o objeto que contém *operator+( )* como membro. Devido à função operador fornecer o resultado, a função deve ser declarada como sendo do tipo *complex*.

Vamos esclarecer um assunto potencialmente confuso: recorde que a redefinição do operador — + neste caso — tem lugar apenas no contexto da classe na qual ocorre a sobrecarga. É por isso que você pode usar o significado-padrão do símbolo + dentro da própria definição da função. Os valores que estão sendo somados aqui são valores *double* comuns. Você não pode sobrecarregar operadores quando eles se aplicam a dados internos porque os novos significados do operador podem ser criados somente para tipos definidos pelo usuário (classes). O programa *cmplx.c* (Listagem 5.2) é um simples driver que ilustra como você poderia usar esta classe. Esse programa usa a função-membro *display( )* para produzir um resultado legível e formatado.

Listagem 5.2 um programa driver que testa a classe *complex*: *cmplx.c*

```
#include <stream.h>

main()
{
  complex x("22+2i"),          // inicializa dois numeros complexos
          y("11+3i"),
          z;

  z = x + y;                          // adiciona os dois numeros
  cout << form("z = %s\n",z.display());      // exibe o resultado
}
```

Questões de Revisão

1. A redefinição de operadores embutidos em C++ é chamada de _____
2. Uma metáfora de programação ocorre quando uma operação executada em um tipo de dados é _____ de maneira que funcione em outro tipo.
3. Somente operadores _____ podem ser sobrecarregados.

Projeto de Programação

1. Amplie a classe de números complexos acrescentando definições para as outras operações aritméticas: —, * e /.

Sobrecarregando Operadores Binários

Além das restrições gerais mencionadas anteriormente, há também restrições da maneira como um operador sobrecarregado em particular pode funcionar. Algumas dessas restrições provêm da definição original do símbolo do operador. Primeiramente, você precisa respeitar o gabarito funcional original do operador. Por exemplo, você não pode mudar um operador binário — um que trabalhe com dois objetos, como o operador da divisão / — para criar um operador unário que trabalha com um único objeto. Da mesma maneira, você não pode converter um operador unário para executar operações binárias. Além disso, você deve manter a forma geral da sintaxe do operador mesmo que você possa mudar o que o operador faz e os objetos sobre os quais ele age. Naturalmente, quando você sobrecarrega operadores que podem executar tanto operações binárias como unárias (tais como + e -) você pode usá-los em qualquer contexto e C++ reconhecerá a diferença.

Outra restrição relacionada ao gabarito do operador diz respeito à precedência. Lembre-se de que a precedência controla a ordem na qual as operações são executadas quando há dois ou mais operadores na expressão. Por exemplo, na expressão inteira x+y/2, a divisão é executada antes da adição porque ela tem prioridade ou precedência mais alta. C++ tem uma série explícita e complexa de regras de precedência (veja o quadro a seguir). Você pode mudar a definição de um operador, mas não pode mudar a sua precedência. Um operador sobrecarregado sempre retém sua posição nesta lista.

Precedência dos Operadores

++ — sizeof (<type>) new delete *avalia da direita para a esquerda*

* (indirection) & (address of) + (unary) - (unary)
* / %

+ -

<< >>

< <= > >=

++ !=

& (bitwise and) *a ordem de avaliação não é garantida*

| bitwise or *a ordem de avaliação não é garantida*

&&

||

? (conditional operator) *avalia da direita para a esquerda*

= += -= *= /= %= >>= <<= &= ^= |=

, (comma)

A função construtor no exemplo *string.c* (Listagem 5.3) inicializa o valor da cadeia. O operador + foi definido como um operador de concatenação de cadeia que acrescenta os caracteres ao fim de uma cadeia já existente.

Listagem 5.3 Uma classe com um operador de concatenação: *string.c*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;          // cria um ponteiro para os caracteres
  int len;          // armazena o numero de caracteres corrente
```

```
public:
  string(const char* =0);    // declara um construtor inicializador
  ~string() { delete v; }    // define um destrutor
  void operator+(char*);     // declara um operador de concatenacao
  char* dump() { return v; }// define um membro de I/O simples
};

string::string(const char* s)
{
  if (s != 0) {              // o construtor possui um valor inicial
    v = new char[(len = strlen(s))];    // aloca memoria
                                        // suficiente - estabelece
    strcpy(v,s);                        // o tamanho parametro
                                        // para a copia da string
    }
  else {                  // o construtor e chamado sem inicializacao
    v = new char[MINLINE];     // aloca um espaco minimo de memoria
    len = 0;                   // estabelece o tamanho
    }
}

void string::operator+(char* p)
{
  char *buf;              // aloca memoria suficiente para o buffer

  buf = new char[len = strlen(p) + len];// estabelece o membro len
  strcpy(buf,v);          // copia o valor original no buffer
  strcat(buf,p);          // concatena o novo valor
  delete v;               // desfaz a alocacao de memoria anterior
  v=buf;                  // estabelece o valor da cadeia para o
                          // novo buffer
}

main()
{
  string s("isto e somente um teste");   // declara um tipo string

  cout << form("%s\n",s.dump());         // exibe o valor corrente
  s + " nada pode dar errado";  // "adiciona" os novos caracteres
  cout << form("%s\n",s.dump());       // exibe o novo valor
}
```

A sobrecarga do operador + neste exemplo demonstra alguns dos princípios e técnicas que aumentam a eficácia de uma série de dados definida pelo

usuário. O significado desse símbolo de adição foi expandido para que pudesse abranger a operação de acrescentar novos caracteres a uma cadeia de caracteres, sendo esta uma operação significativamente diferente. Esta classe de cadeias contém uma simples operação de concatenação, a qual designa cadeias de caracteres C++, que são um conjunto de caracteres terminados por '\0' (caractere que significa *fim-de-cadeia*) a uma variável da classe *string*. Por exemplo, o seguinte trecho de programa:

```
string p("isto é um teste");
p+"nada pode sair errado.";
```

acrescenta uma cadeia de caracteres literal à variável *p*. (Note que o operador + sempre acrescenta a cadeia representada por seu segundo operando à cadeia representada por seu primeiro operando.) A variável *p* contém agora a cadeia original mais os caracteres adicionais e é exibida na tela através da seguinte instrução:

```
cout << form ("%s\n", s.dump () )
```

A função *form( )* formata a saída como uma cadeia de caracteres tradicional.

A definição desse novo operador segue as regras de sintaxe que você usou para criar a classe de números complexos. Você declara a função *operator+( )* com um tipo e com uma lista de parâmetros. Neste caso, o único argumento para a função é uma cadeia de caracteres (um vetor de caracteres terminado pelo símbolo '\0'). Após a variável *v* ser ampliada, os valores velho e novo são combinados no buffer apontado por *v*. O novo operador é do tipo *void* porque ele não dá um valor de retorno; em vez disso, ele simplesmente altera o operador da esquerda. (Esse procedimento parece ser estranho, considerando a função normal do operador +; no entanto, a implementação da operação é correta porque o operando da esquerda é a única cadeia da classe no programa.)

O exemplo *string.c* também contém uma pequena função *main( )* que serve como propulsora para demonstrar a operação da classe. Após usar um valor de inicialização para declarar a cadeia-objeto *s*, o programa "soma" uma cadeia de caracteres literal. Ambas as formas da cadeia são então mostradas na tela.

Embora possa não ser muito aparente no programa, a função *operator+( )* é implementada com um operador binário. O valor que é passado como parâmetro é o segundo operando, ou operando da direita; o primeiro operando é um objeto

da classe *string* da qual esta função é um membro. Isso permite que a função operador tenha um acesso especial ao objeto. Embora a classe não esteja explicitamente na lista de parâmetros, C++ não obstante trata-a implicitamente como parte da função; isso torna a função um operador binário, como condição assumida.

A associação da função *operador+( )* na classe *string* satisfaz também outro requisito importante para a redefinição de operador — um dos objetos envolvidos tem que ser uma classe-objeto. Este exemplo contém uma classe-objeto e um tipo de dados muito básico — a tradicional cadeia de caracteres.

Operadores com Duas Classes de Objetos

O exemplo *string2.c* (Listagem 5.4) ilustra um operador de concatenação levemente diferente. Neste exemplo, a definição de classe utiliza o operador < para representar a concatenação de uma cadeia em outra. O exemplo anterior (Listagem 5.3) também usou um vetor de caracteres (um tipo de dado embutido) para ampliar a cadeia. Aqui, a concatenação envolve duas classes, uma situação mais complicada.

Listagem 5.4 Um operador que concatena dois objetos do tipo *string*: *string2.c*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;          // ponteiro para os caracteres
  int len;          // tamanho corrente da cadeia

public:
  string(const char* =0);      // construtor
  ~string() { delete v; }      // destrutor
  void operator<(string&);     // concatena dois objetos de
                               // classe string
  char* dump() { return v; }  // define um membro de I/O simples
};
```

```
string::string(const char* s)
{
  if (s != 0) {                    // fornecido valor inicial
    v = new char[(len = strlen(s)) + 1];    // aloca memoria
    strcpy(v,s);                   // copia o valor para a string
    }
  else {                           // usando o valor assumido
    v = new char[MINLINE];         // aloca um espaco minimo de memoria
    len = 0;                       // exibe tamanho zero
    }
}

void string::operator<(string& s)
{
  char *buf;   // aloca buffer de tamanho igual a ambos os strings
  buf = new char[len = len + s.len + 1];
  strcpy(buf,v);          // copia a string original para o buffer
  strcat(buf,s.v);        // adiciona o conteudo da segunda string
  delete v;               // troca a anterior pela nova
  v = buf;                // estabelece o valor da cadeia para o
                          // novo buffer
}

main()
{
  string s("isto e somente um teste"),   // declara e inicializa
         t( " nada pode dar errado");    // as duas strings

  cout << form("%s\n%s\n",s.dump(),t.dump()); // exibe os
                                              // correntes ...
                                              // ... valores
   s < &t;                          // "adiciona" os objetos string
  cout << form("%s\n",s.dump());    // exibe o novo valor
}
```

A função *operator<( )*, neste exemplo, acrescenta o conteúdo de uma cadeia (tomada como parâmetro) a outra. Devido a esta função ser um operador binário, ela requer um operando à direita e outro à esquerda. No entanto, somente o operando da esquerda muda. O operando da direita — neste caso, o parâmetro — mantém seu valor. A função *operator<( )* tira vantagem do fato de que uma função-membro tem acesso privilegiado à parte privada da classe. Devido a este privilégio derivar da classe e não do objeto, ele se estende igualmente tanto à parte

implícita quanto à explícita do valor do argumento. Assim, a função abre a parte privada da classe sem violar a capacidade de ocultamento de dados de quaisquer das classes.

A definição de um operador de concatenação de cadeia neste exemplo é significativamente diferente da definição do exemplo anterior (Listagem 5.3). A diferença principal é que este operador concatena os objetos de duas classes *string*. O operador anterior simplesmente executou uma operação do tipo preenchimento da cadeia que alterou o valor que a classe *string* tinha, não com uma classe, mas sim com uma cadeia de caracteres normais C. Durante o desenvolvimento de um programa, ambas as situações ocorrem freqüentemente: o caso anterior é uma espécie de *conversão de cadeia* porque um dos parâmetros não é uma classe. O último exemplo apresenta uma concatenação muito mais direta.

Usando Funções Operador Amigas para Maior Flexibilidade

Os dois últimos exemplos mostram a função que define o operador como um membro da mesma classe que o novo operador. Neste caso, o argumento à esquerda é sempre implícito, e a classe "possui" a função. No entanto, esta não é a única maneira de se produzir um novo operador. Você pode também redefinir um operador usando funções *amigas*. O programa *string3.c* (Listagem 5.5) é equivalente ao exemplo anterior, exceto que ele usa uma função do tipo *amiga*.

Listagem 5.5 Um operador definido como *amigo* a uma classe que concatena duas cadeias: *string3.c*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;          // ponteiro para os atuais caracteres
  int len;          // tamanho corrente da cadeia

public:
  string(const char* =0);          // declara o construtor
```

```
  ~string() { delete v; }          // ... e destrutor
  friend void operator<(string&,string&);
  char* dump() { return v; }       // exibe tudo
};

string::string(const char* s)
{
  if (s != 0) {                  // fornecido valor inicial
    v = new char[(len = strlen(s))];     // aloca memoria
    strcpy(v,s);                 // copia o valor para a string
    }
  else {                          // nao e fornecido valor inicial
    v = new char[MINLINE];       // alocacao de memoria minima
    len = 0;                     // estabelece tamanho zero
    }
}

void operator<(string& a, string& b)          // define o
                                              // operador usando
{                                             // referencias da
                                              // classe string
  char *buf;                                  // cria buffer de
                                              // tamanho suficiente
  buf = new char[a.len = a.len + b.len];      // para armazenar o
                                              // novo valor
  strcpy(buf,a.v);          // copia o antigo
  strcat(buf,b.v);          // ... e entao o novo
  delete a.v;               // troca o valor
  a.v = buf;
}

main()
{
  string s("isto e somente um teste"),     // inicializa um
                                           // objeto string
         t( " nada pode dar errado.");     // ... e o outro

  cout << form("%s\n",s.dump());       // exibe o primeiro
   s < t;                              // concatena-os
  cout << form("%s\n",s.dump());       // exibe o novo valor
}
```

Aqui, a função *operator<( )* se parece com uma que foi definida no exemplo *string2.c* (Listagem 5.4). No entanto, em vez de usar uma cadeia

subentendida como operando do "lado direito", ela usa como argumento dois objetos explícitos. A primeira cadeia é alterada; a segunda não. Você deve declarar o primeiro parâmetro como referência porque o operador modifica-o. Embora não seja necessário declarar o segundo parâmetro como referência, fazer isso economiza um pouco de memória ao rodar o programa, porque C++ não precisa criar uma nova cadeia dentro da função *operator<( )*. (Lembre-se, como o tipo de referência substitui um parâmetro com o endereço do argumento real, não é necessário criar uma variável local ou preencher com o valor do parâmetro verdadeiro.) A função propulsora *main( )* mostra como usar o novo operador com esta classe.

Usar uma função *amiga* para redefinir um operador é uma maneira mais clara e natural do que fazer o mesmo usando uma função-membro. O modelo de função *amiga* é mais detalhadamente ilustrado pelo exemplo *string4.c* (Listagem5.6). Neste exemplo, o operador + combina duas cadeias, da mesma maneira que o programa da Listagem 5.5. Porém, em vez de alterar a primeira cadeia, ele deixa ambos os operandos inalterados e armazena o resultado em nova variável. Este procedimento aproxima-se mais do tipo de operação usualmente associado com o símbolo +, mesmo que não haja exigências da sintaxe para esta associação. Você poderia implementar este exemplo usando funções-membro sem perda de funcionalidade, mas a legibilidade do seu programa ficaria prejudicada.

Listagem 5.6 Uma função operador *amiga* que resulta num ponteiro para uma cadeia: *string4.c*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;          // ponteiro para os atuais caracteres
  int len;          // tamanho corrente da cadeia

public:
  string(const char* =0);          // declara o construtor
  ~string() { delete v; }          // ... e destrutor
  friend void operator+(string&,string&); // declara um novo
                                          // operador "+"
  char* display() { return v; }    // exibe tudo
};
```

```
string::string(const char* s)
{
  if (s != 0) {                                  // fornecido valor
                                                 // inicial
    v = new char[(len = strlen(s))];              // aloca memoria
    strcpy(v,s);                                 // copia o valor para a
                                                 // string
    }
  else {                                 // nao e fornecido valor inicial
    v = new char[MINLINE];               // alocacao de memoria minima
    len = 0;                             // estabelece tamanho zero
    }
}
string* operator+(string& a, string& b)    // define o operador
                                           // usando string
{
  string* temp;
  temp = new string;                 // aloca a nova string
  temp->v = new char[temp->len = a.len + b.len]; // cria um
                                                 // buffer
  strcpy(temp->v,a.v);               // copia a primeira string
  strcat(temp->v,b.v);               // ... e entao a outra
   return temp;                      // retorna o novo objeto
}

main()
{
  string s("isto e somente um teste"),     // inicializa um objeto
                                           // string
        t( " nada pode dar errado."),      // ... e o outro
       *result;                            // declara um ponteiro
                                           // para uma matriz
  result = s + t;                          // concatena as duas
                                           // strings

  cout << result->display();               // exibe o conteudo
}
```

Aparece uma restrição na sobrecarga de operador quando você usa funções *amigas* para criar novos operadores; uma função operador precisa ter pelo menos uma classe como parâmetro. Antes do exemplo que acabamos de ver, esse requisito havia sido satisfeito pelo argumento *this* que estava implícito na classe na qual a função de redefinição de operador era um membro. A Listagem 5.3, que concatena

um vetor de caracteres a uma cadeia, demonstrou que uma função operador pode ter um argumento que não pertence a nenhuma classe. No entanto, você não pode usar uma função de redefinição de operador que tenha somente argumentos não-classe, mesmo que você a declare *amiga* à classe, de maneira que ela tenha privilégios para acessar a área privada da classe. Pelos menos um parâmetro deve ser sempre uma classe.

Há uma situação de programação na qual você *deve* usar a função operador *amiga* em vez de usar uma função-membro. Se o primeiro argumento de uma redefinição de operador não é uma variável de classe, então você deve usar uma função operador *amiga*. Considere o exemplo *string5.c* (Listagem 5.7), que concatena uma antiga cadeia de caracteres — um vetor de caracteres terminado por '\0' — com uma nova cadeia, para criar uma nova cadeia de caracteres. Se você tenta fazer isso com uma função-membro, o primeiro argumento é o implícito *this*, e a cadeia de caracteres deve vir primeiro. Neste caso, você deve usar a seguinte forma:

```
<string object> + <character string>
```

No entanto, isso não é o que você quer fazer. Somente as funções *amigas* permitem que o programador controle a ordem dos argumentos para um operador, como no caso do exemplo a seguir:

Listagem 5.7 Uma função operador *amiga* que resulta num ponteiro para uma cadeia: *string5.c*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;                                  // ponteiro para os
                                            // atuais caracteres
  int len;                                  // tamanho corrente
                                            // da cadeia
public:
  string(const char* =0);           // declara o construtor
  ~string() { delete v; }           // ... e destrutor
  friend string* operator+(char*,string&);  // declara um novo
                                            // operador "+"
  char* display() { return v; }             // exibe tudo
```

```
};

string::string(const char* s)
{                                          :
  if (s != 0) {                             // fornecido valor inicial
    v = new char[(len = strlen(s))];        // aloca memoria
    strcpy(v,s);                           // copia o valor para a string
    }
  else {                                   // nao e fornecido valor inicial
    v = new char[MINLINE];                 // alocacao de memoria minima
    len = 0;                               // estabelece tamanho zero
    }
}
string* operator+(char* a, string& b)      // define o operador
                                           // usando string
{
  string* temp;
  temp = new string;                       // aloca a nova string
  temp->v = new char[temp->len = strlen(a) + b.len]; // cria
                                                     // um buffer
  strcpy(temp->v,a);                       // copia a primeira string
  strcat(temp->v,b.v);                     // ... e entao a outra
   return temp;                            // retorna o novo objeto
}

main()
{
  string s(" nada pode dar errado."), // inicializa um objeto
                                      // string
       *result;                       // declara um ponteiro
                                      // para um objeto string
  result = "isto e somente um teste" + s;  // combina vetor de
                                           // caracteres e string

  cout << result->display();                // exibe o conteudo
}
```

Operadores com Múltiplas Sobrecargas

Tome cuidado para que não passe despercebida a questão da ordem dos operandos numa função operador. Afinal, a maioria das operações executadas sobre vetores de dados internos são comutativas, quer dizer, funcionam em ambas as direções. Com operações mistas — aquelas com um número inteiro e uma variável *double*, por exemplo, você pode seguramente ignorar a diferença, porque você sabe que o compilador converterá o número inteiro num valor *double* antes de executar quaisquer operações. Quando você cria suas próprias classes e suas operações associadas, deve escolher explicitamente a ordem dos operandos. Se for permitida mais de uma ordem, então você deve fornecer mais do que uma função operador. Felizmente, da mesma forma que acontece com funções sobrecarregadas, você pode usar múltiplos operadores sobrecarregados em uma classe enquanto suas listas de argumentos forem diferentes.

O exemplo *string6.c* (Listagem 5.8) ilustra um operador que aceita mais de uma ordem de operando. Novamente, o símbolo + é o operador de concatenação redefinido. Como no exemplo anterior, os operandos permanecem inalterados, e o operador produz como resultado uma nova cadeia. O operador suporta ambas as seqüências de chamada:

```
<character string> + <string>
```

e

```
<string> + <character string>
```

porque o programa contém duas definições diferentes da função *operator+( )*. Assim, a própria função *operator+( )* é sobrecarregada neste exemplo.

Listagem 5.8 Uma função operador *amiga* sobrecarregada que resulta num ponteiro para uma cadeia: *string6.c*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;          // ponteiro para os atuais caracteres
```

```
  int len;          // tamanho corrente da cadeia

public:
  string(const char* =0);          // declara o construtor
  ~string() { delete v; }          // ... e destrutor
  friend string* operator+(char*,string&);   // declara um novo
                                             // operador "+"
  friend string* operator+(string&,char*);   // ... e um
                                             // alternativo "+"
  char* display() { return v; }      // exibe tudo
};

string::string(const char* s)
{
  if (s != 0) {              // fornecido valor inicial
    v = new char[(len = strlen(s))];    // aloca memoria
    strcpy(v,s);                 // copia o valor para a string
    }
  else {                         // nao e fornecido valor inicial
    v = new char[MINLINE];       // alocacao de memoria minima
    len = 0;                     // estabelece tamanho zero
    }
}

string* operator+(char* a, string& b)    // define o operador
                                         // usando string
{
  string* temp;
  temp = new string;           // aloca a nova string
  temp->v = new char[temp->len = strlen(a) + b.len]; // cria
                                                     // um buffer
  strcpy(temp->v,a);           // copia a primeira string
  strcat(temp->v,b.v);         // ... e entao a outra
   return temp;                // retorna o novo objeto
}

string* operator+(string& a, char* b)    // define uma funcao
                                         // operador+
{                                        // alternativa
  string* temp;
  temp = new string;                     // aloca a nova string
  temp->v = new char[temp->len = a.len + strlen(b)]; // cria
                                                     // um buffer
  strcpy(temp->v,a.v);                   // copia a primeira string
  strcat(temp->v,b);                     // ... e entao a outra
```

```
   return temp;                          // retorna o novo objeto
}

main()
{
  string s(" nada pode dar errado."),   // inicializa um objeto
                                        // string
       *result;                         // declara um ponteiro
                                        // para um objeto string

  result = "isto e somente um teste" + s;  // combina vetor de
                                           // caracteres e string

  cout << result->display() << "\n";        // exibe o conteudo

  string x("alo ");              // inicializa um novo objeto string

  result = x + "mundo";          // adiciona a um vetor de caracteres

  cout << result->display();  // exibe o resultado

}
```

Criando Operadores Unários

Pelo fato de você ter de respeitar o "gabarito de chamada" do símbolo de um operador, você pode usar alguns símbolos somente para criar operadores unários, que têm apenas um operando. Alguns poucos operadores admitem seqüências de chamada tanto com um só operando quanto com dois. O quadro a seguir lista todos os operadores C++ disponíveis para a sobrecarga.

Operadores Unário, Binário e Binário-Unário

Unário:	+	—	!	~
Binário:	/	%	()	[]
	new	delete	+=	-=
	*=	/=	|	^
	||	&&	<	<=
	>	>=	<<	>>
	|=	^=	&=	<<=
	>>=	==	!=	
Unário ou Binário:	+	-	*	&

Redefinir um operador unário é semelhante a redefinir um binário. Você usa um símbolo com a função do operador para criar um operador que é único a uma classe em particular. Para isso, convém uma função *amiga* ou uma função-membro. No entanto, devido ao operador unário usar apenas um parâmetro, você deverá usar como argumento uma classe.

O exemplo *circincl.c* (Listagem 5.9) ilustra um operador unário definido como uma função-membro. Esse programa define uma classe *circle* que consta de um valor para o raio de um círculo e um incremento que aumenta seu tamanho. São definidas também funções-membro para a área e para a circunferência. A parte importante do exemplo, porém, é o operador ++. Esse operador foi redefinido para funcionar com esta nova figura, enquanto conserva ainda a essência do operador incremento da maneira como se aplica a tipos de dados internos. Devido à função-membro *operator++( )* ser um operador unário, ela não pode ter outros argumentos além do parâmetro implícito *this*. O valor privado *incr* aumenta o comprimento do raio.

Listagem 5.9 Redefinição de um operador unário: *circincl.c*

```
#include <stream.h>

const double pi = 3.1415;

class circle {
  double radius;     // valor chave para o tamanho
  int incr;          // quanto incrementa-lo a cada vez
```

```
public:
  circle(double r =0, int i =1) { radius = r; incr = i; } // cria
                                                          // um circulo
  double area() { return pi * radius * radius; } // retorna a area
  double cir() { return pi * 2 * radius; } // e a circunferencia
  void operator++();          // declara um operador de incremento
};

void circle::operator++()
{
  radius += incr;
}

main()
{
  circle x(35);

  for (int i=0; i<10; i++) {
    cout << form("area=%10.3f\ncircunferencia=%10.3f\n",
                 x.area(),x.cir());
    x++;
    }
}
```

A função driver *main( )* ilustra como funciona o novo operador ++. Note que, no contexto de um operador sobrecarregado, ++x e x++ executam operações idênticas. As diferenças entre estas duas formas ocorrem quando o objeto é do tipo inteiro e não é traduzido para esta nova definição.

O exemplo *circinc2.c* (Listagem 5.10) usa o mesmo código da listagem anterior, exceto que ele define *operator++( )* como *amigo* à classe *circle*. Neste caso, você deve passar a classe argumento explicitamente como parâmetro para a nova função. Fora esta diferença, os dois programas se comportam identicamente, e até compartilham da mesma função driver.

Listagem 5.10 Redefinição de um operador unário como função *amiga*: *circinc2.c*

```
#include <stream.h>

const double pi = 3.1415;

class circle {
  double radius;      // armazena o valor de definicao
  int incr;           // isto nos diz de quanto deve ser o
```

```
                    // incremento do raio

public:
  circle(double r =0, int i =1)  { radius = r; incr = i; } // cria
                                                    // um circulo
  double area() { return pi * radius * radius; } // retorna a area
  double cir() { return pi * 2 * radius; }    // ... e a
                                              // circunferencia
  friend void operator++(circle&);          // declara um
                                            // operador de incremento
};

void operator++(circle& c)                  // c e uma referencia a
                                            // uma classe objeto
{
  c.radius += c.incr;           // incrementa o raio de acordo com
                                // os valores correntes
}

main()
{
  circle x(35);

  for (int i=0; i<10; i++) {
    cout << form("area=%10.3f\ncircunferencia=%10.3f\n",
                 x.area(),x.cir());
    x++;
    }
}
```

Operadores Combinados

Um exemplo anterior (Listagem 5.8) mostrou que você poderia criar mais do que uma definição e assim sobrecarregar a função operador. Essa técnica é útil tanto com os operadores como também com funções sobrecarregadas comuns. Aplicam-se as mesmas vantagens — elimina o código de seleção redundante e permite um estilo de programação mais simples, fácil de acompanhar. No entanto, com os operadores, você pode avançar um passo a mais na sobrecarga. É possível definir uma função como operador unário e como operador binário — desde que

o "gabarito" do símbolo admita ambos os tipos de chamada de operador. (Veja no quadro anterior os operadores apropriados.)

Alguns poucos símbolos de operadores naturalmente caem dentro dessa categoria de uso dual. O símbolo mais usado comumente é o sinal - que representa tanto a subtração quanto um número negativo. Ambos os usos ocorrem nos tipos de dados numéricos em C++. Da mesma maneira, o símbolo + representa a soma ou o número positivo. Os símbolos & e * podem também ser sobrecarregados desta maneira. Estes dois símbolos, no entanto, são ainda mais importantes no contexto corrente porque são os únicos operadores que podem ser definidos como unário e binário simultaneamente.

O programa *dollars.c* (Listagem 5.11) contém a definição de uma classe que usa operadores + e -. Estes operadores, por sua vez, têm cada um uma definição binária e uma definição unária. A própria classe aceita um valor de inicialização no formato tradicional de cruzeiros-e-centavos, e executa adição e subtração. Estas operações manipulam números com um ponto decimal fixo, tais como 11.47 e 12.00. A adição e subtração de números com estes não são de imediato possibilitadas pelos tipos de dados internos em C++. Você não pode usar diretamente variáveis inteiras dos tipos *short*, *int* e *long* porque elas representam números inteiros, e portanto qualquer ponto decimal implícito deve ser fornecido pelo usuário. Números reais — *float* e *double* — são também inúteis porque introduzem um erro de arredondamento no cálculo. No entanto, a classe *cruzeiro* é construída para manipular expressões da seguinte forma:

```
1.23 + 12.00
```

e receber o resultado apropriado sem quaisquer erros de arrendondamento. O formato de exibição é o mesmo formato da entrada. O projeto da classe usa a estratégia básica que consta em armazenar o valor atual como um número inteiro *long*, o número de centavos.

Listagem 5.11 Um operador de sobrecarga definido como unário e binário: *dollars.c*

```
#include <stream.h>
#include <string.h>

const char decpt = ".",    // uma referencia para o ponto decimal
           eos = '\0';     // ... e uma para o final da cadeia

class dollar {
  long pennies;            // numero total deles
```

```
public:
  dollar(char *);          // inicializa o construtor
  friend dollar* operator+(dollar&,dollar&);      // um novo
                                                  // operador "+"
  friend dollar* operator-(dollar&,dollar&);      // ... e um "-"
  void operator+()                          // define um unario "+"
    { pennies = ( pennies<0 ) ? -pennies : pennies; }
  void operator-()                          // e um unario "-"
    { pennies = ( pennies<0 ) ? pennies : -pennies; }
  char* display();                          // mostra os conteudos
};

dollar::dollar(char* bucks)
{
  char* temp;
  temp = strtok(bucks,".");          // remove o ponto decimal
  strcat(temp,strtok(0,0);
  pennies = atol(temp);              // converte para long
}

dollar* operator+(dollar& a, dollar& b)
{
  dollar* temp;
  temp->pennies = a.pennies + b.pennies;     // soma os valores
                                             // apropriados
  return temp;
}
dollar* operator-(dollar& a, dollar& b)
{
  dollar* temp;
  temp->pennies = a.pennies - b.pennies;     // subtrai os valores
                                             // apropriados
  return temp;
}
char* dollar::display()
{
  char* temp;                // declara um ponteiro como buffer
  temp = new char[20];       // aloca memoria para o buffer -
                             // provavelmente area muito grande!
  sprintf(temp,"%ld",pennies);     // converte para numero
  int p = strlen(temp); // onde esta a marca de final de cadeia?
  temp[p+2] = eos;           // adiciona o ponto decimal a direita
```

```
  temp[p+1] = temp[p];
  temp[p] = temp[p-1];
  temp[p-1] = temp[p-2];
  temp[p-2] = decpt;
  return temp;                          // retorna o novo valor
}

main()
{
  dollar x("12.50"),          // declara e inicializa um objeto dollar
         y("4.25"),           // ... e um outro
         *result;             // declara um ponteiro como objeto dollar

  result = x + y;                                // adicao simples
  cout << result->display() << "\n";             // ... e exibicao

  result = x - y;                                // subtracao simples
  cout << result->display() << "\n";             // ... e exibicao

  -x;                                            // cria um valor negativo
  cout << form("%s\n",x.display());              // exibe-o

  result = x + y;                    // adiciona agora o valor negativo
  cout << form("%s\n",result.display());     // exibe o resultado

}
```

O ponto-chave a destacar neste exemplo é o uso da sobrecarga de operadores. Para cada redefinição binária, é declarada uma função *amiga*; isso resulta num ponteiro para um novo objeto da classe, cujo valor é deduzido executando-se a operação apropriada nos dois operandos que são tomados como argumentos. O símbolo + soma os valores correntes dos argumentos, enquanto o símbolo - subtrai o segundo operando do primeiro.

A versão unária de + é uma função-membro cujo único argumento é o ponteiro *this* implícito. Qualquer que seja o valor corrente em cruzeiros, esta função converte-o em uma quantidade positiva, usando a simples chamada do operador condicional. Note que a função é definida como *inline*. O símbolo - converte o valor numa quantidade negativa e age como um exato complemento da definição +.

Tenha cuidado quando sobrecarregar operadores. Neste exemplo, você não tem dificuldades em distinguir os dois usos dos operadores porque suas funções — e usualmente suas posições físicas no programa — indicam seu uso. No entanto,

o compilador não reconhecerá uma situação ambígüa e, às vezes, isso pode levar a uma perda no poder de expressão na sintaxe sobrecarregada. Por exemplo, a combinação a seguir:

```
result = -x+y;
```

não funciona, apesar de ser uma expressão perfeitamente legítima se você estiver usando valores inteiros ou *double*. A habilidade de poder criar seus próprios operadores traz a responsabilidade de considerar todas as situações nas quais ele possa ser usado.

Questões de Revisão

1. Qual função é usada para sobrecarregar operadores dentro do contexto de uma classe?

2. Uma função operador pode ser _____ da classe ou função_____.

3. Um operador binário tem _____ operandos. Um operador unário está restrito a _____.

4. O _____ básico de um operador deve ser fiel.

5. Os símbolos seguintes podem ser usados somente com operações unárias _____ _____.

6. Por outro lado, estes operadores _____ podem ser unários ou binários.

Projetos de Programação

1. Usando a classe *circle* no programa *circinc2.c* como um modelo, defina uma classe para o cilindro.

2. Crie uma classe *string* que tenha um conjunto completo de operadores e funções.

Sobrecarregando o Símbolo ( )

Na maioria dos casos de sobrecarga de operador, você usará um símbolo para o operador de ação comum. Símbolos como +, - e * geralmente sugerem operações análogas para uma classe que reflita as operações dos dados internos. A definição da soma de números complexos foi um exemplo, como também o foi o uso do operador incremento ++ para a classe *circle*. Mesmo para combinações mais sutis — a concatenação de cadeias de caracteres e adição, por exemplo — há usualmente uma simetria que sugere uma relação de bom senso. O uso de << e >> com a biblioteca *stream*, por exemplo, simboliza obviamente o redirecionamento a muitos elementos. No entanto, não há regra de sintaxe que lhe restrinja a esses operadores comuns.

Você pode usar toda a gama de operadores C++ para redefinição. Esta gama de operadores inclui algumas opções não usuais que podem levar tanto a expressões sem sentido quanto a expressões surpreendentemente poderosas. Por exemplo, a sintaxe de chamada de função está implantada no operador ( ). Este operador contém duas coisas que podem passar desapercebidas ao programador principiante — primeiro, que ele é um operador, afinal, e, segundo, que ele pode ser usado separado do nome da função. Este último ponto é ilustrado pela forma de um ponteiro a uma função. Recorde que uma declaração destas assume a seguinte forma:

```
int (*f) ( );
```

Está claro que os parênteses são mais do que uma operação de agrupamento. Neste contexto, os parênteses separam a variável **f* da chamada da função criada por ( ). Sem esta forma, C++ interpretaria a sentença como uma função chamada de *f* que resulta num ponteiro para um valor inteiro, em vez do uso correto de *f* como um ponteiro de função.

Com uma classe C++, você pode tornar-se um pouco mais independente do operador ( ) e redefini-lo na realidade como anteriormente redefiniu o operador +. O mecanismo de redefinição permite-lhe captar uma chamada de função e interpretá-la de uma maneira especial para uma aplicação. Esta é uma extensão muito poderosa das ferramentas usualmente disponíveis ao programador. Na verdade, um possível uso dessa sintaxe é a criação de conversão personalizada de operadores.

Primeiramente, uma palavra de precaução. O número de casos nos quais tem sentido uma redefinição da seqüência de chamada na função é ainda mais restrito do que para os operadores mais comuns. A finalidade da redefinição de operador é simplificar a estrutura de um programa. No entanto, os mesmos mecanismos que permitem isso, se forem levados ao extremo, podem causar o efeito oposto, obscurecendo assim o seu programa e tornando impossível entendê-lo.

Uma redefinição útil e interessante do operador ( ) permite-lhe criar um operador de conversão para uma classe definida pelo usuário. Você pode então usar este modelo para converter a classe-objeto — ou, mais usualmente, parte dela — em algum outro tipo de dado. A aptidão para fazer uma conversão explícita a partir de uma classe é ainda mais importante em C++, onde você freqüentemente executa conversões de tipos e pode facilmente movimentar valores entre duas estratégias de armazenamento relacionadas.

O exemplo *strcast.h* (Listagem 5.12) inclui a definição de uma classe *string* ampliada. O objeto *string* inclui não apenas os caracteres que formam a cadeia atual, mas também o número de caracteres contidos atualmente no objeto. (Aqui, você não precisa usar o costumeiro sinal *fim-de-cadeia* — o caractere '\0' — que caracteriza uma cadeia de caracteres C++ ortodoxa.) Para que este objeto seja o mais válido possível, você talvez precise convertê-lo no formato tradicional de cadeia embutida. Você poderia realizar esta tarefa com uma função de conversão definida como membro da classe *string* ou mesmo *amiga* a esta classe. Porém, como você está convertendo duas espécies de dados similares, um veículo mais apropriado para isso seria um operador de conversão. Em vez de uma forma como esta abaixo:

```
string x("isto é apenas um teste");
char* y;
y = x.convert( );
```

você pode usar uma expressão mais clara:

```
string x("isto é apenas um teste");
char* y;
y = (char*) x;
```

Para quem está lendo pela primeira vez o programa, a segunda forma é mais clara. É sempre uma boa prática usar operadores que sejam familiares ao programador com novas variáveis definidas pelo usuário. Neste caso, você pode

usar o mesmo tipo de operação de conversão para qualquer variável de classe que você defina.

Listagem 5.12 Um operador de conversão explícita definido para uma classe *string*: *strcast.h*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;          // ponteiro para os atuais caracteres
  int len;          // tamanho corrente da cadeia

public:
  string(const char* =0);          // construtor
  ~string() { delete v; }          // destrutor
  void operator<(string*);         // operador de concatenacao
  operator char*();                // novo tipo de modelo de
                                   // operador
};
string::string(const char* s)
{
  if (s != 0) {                // fornecido valor inicial
    v = new char[(len = strlen(s))];    // aloca memoria
    strcpy(v,s);               // copia o valor para a string
    }
  else {
    v = new char[MINLINE];     // alocacao de memoria minima
    len = 0;                   // estabelece tamanho zero
    }
}

void string::operator<(string* s)
{
  char *buf;                         // cria um novo e largo buffer
  buf = new char[len = len + s->len];
  strcpy(buf,v);                     // copia ambas as string para ele
  strcat(buf,s->v);
  delete v;                    // troca o conteudo da string original
   v = buf;
}
```

```
string::operator char*()
{
  *(v + len + 1) = '\0';     // coloca a marca de final de cadeia
  return v;
}
```

A sintaxe de conversão é simples. O programa verifica a última posição na variável *v* que tem no momento um valor. Isso é indicado, naturalmente, pelo membro *len*. Ele acrescenta um a esse valor e coloca uma marca de *fim-de-cadeia* na posição indicada por este número. Isso converte *v* numa cadeia de caracteres normal tal que seu valor pode ser utilizado de volta na função como uma cadeia de caracteres. (Se você é contrário ao uso de um modelo para alterar o valor do objeto, simplesmente copie o valor numa variável *temp* do tipo *char** e execute as mesmas operações.)

Não há nenhuma mágica no uso desse tipo de conversão. A sintaxe do operador fornece a seqüência de chamada: ela se compromete a tomar os operandos fornecidos e convertê-los em argumentos para a função operador especificada — neste caso os argumentos são implícitos, mas muitos dos exemplos deste capítulo usaram argumentos explícitos. Após você fornecer os argumentos, a função torna-se igual a qualquer outra. As sentenças e definições mantêm-se verdadeiras aqui como nos subprogramas ordinários. A única diferença está na interface.

O programa *strcast.c* (Listagem 5.13) contém um driver que lhe permite testar esta nova classe. Ele cria uma classe-objeto *temp* e inicializa-a com um valor. A função de saída formatada *form( )* imprime o valor. (Note, no entanto, que o formato que ela especifica é uma cadeia de caracteres tradicional, ou seja, um vetor de caracteres do tipo "terminado com \0".) O programa amolda a classe-objeto a esse tipo embutido e a função *cout* mostra-a na tela. Sem o novo operador, somente uma função-membro definida poderia mostrar esse valor.

Listagem 5.13 Um programa driver para testar o exemplo anterior (Listagem 5.12): *strcast.c*

```
main()
{
  string temp("Alo Mundo"); // cria e inicializa uma classe objeto

  cout << form("%s\n",(char *)temp);    // molda-a como uma cadeia
                                        // ordinaria
}
```

Vamos ver mais um exemplo nesta seção. A classe definida em *strops1.h* (Listagem 5.14) é embasada na funcionalidade da *string* redefinindo o operador [ ], usado como índice para um vetor. O exemplo a seguir acrescenta esta propriedade de indexação à classe string.

Listagem 5.14 Uma redefinição do operador [ ]: *strops1.h*

```
#include <stream.h>
#include <string.h>

// cria uma classe string verdadeiramente dinamica

#define MINLINE 55

class string {
  char *v;          // ponteiro para os atuais caracteres
  int len;          // tamanho corrente da cadeia

public:
  string(const char* =0);          // construtor
  ~string() { delete v; }          // destrutor
  void operator<(string*);         // concatena duas cadeias
  operator char*();                // converte em cadeia ortodoxa
  char* operator[](int);           // declara um operador substring
};

string::string(const char* s)      // definicao do construtor
{
  if (s != 0) {      // se for fornecido o valor inicial entao ...
    v = new char[(len = strlen(s)) + 1];     // aloca memoria E
                                             // define o tamanho
    strcpy(v,s);                 // copia o valor para a string
    }
  else {
    v = new char[MINLINE];  // caso contrario, aloca quantidade
                            // de memoria padrao
    len = 0;                // indica caracteres nao ativos ainda
    }
}

void string::operator<(string* s) // define concatenacao
{
  char *buf;                        // declara um buffer temporario
  buf = new char[len = len + s->len + 1]; // aloca um espaco
                                          // suficientemente grande
```

```
  strcpy(buf,v);              // copia o primeiro valor
  strcat(buf,s->v);           // ... entao insere o restante ate o fim
  delete v;                   // remove o valor antigo
   v = buf;                   // atribui o novo
}
string::operator char*()              // define tipo modelo
{
  *(v + len + 1) = '\0';     // insere a marca de final de cadeia
  return v;                // retorna o valor corrente agora alterado
}

char* string::operator[](int pos)     // redefine o operador "[]"
{
  return v+pos-1;             // calcula a posicao especificada em v
}
```

Atribuir uma operação de indexação para o objeto string definido nesta classe é uma tarefa fácil. A indexação numa série regular de caracteres meramente resulta no caractere que está na localização desejada. Por exemplo, a sentença seguinte:

```
cout << form("ch=%c\n",ch[5]);
```

retorna o valor que está armazenado na quinta posição em *ch*. O programa não se afasta muito dessa noção, mas, por tratar-se de cadeias de caracteres, ele estabelece a redefinição do operador para resultar numa cadeia de caracteres em vez de um único caractere. Quando você executa o seguinte trecho de programa:

```
string x("ABCDEF");
cout << form("%s\n",x[3]);
```

você pode esperar o seguinte resultado

```
CDEF
```

É mostrado o conteúdo atual de *x* começando pelo terceiro caractere e continuando até o fim da cadeia.

A função-membro *operator[ ]( )* toma como parâmetro a posição dentro da cadeia, para iniciar. Ela usa este número para calcular um deslocamento do endereço do primeiro caractere na variável-membro *v*. Então, dá como resultado este endereço.

O programa *strops1.c* (Listagem 5.15) é um recurso a ser usado com classes *string* definidas em *strops1.c*. Ele declara e inicializa duas cadeias. Após elas serem concatenadas, o programa mostra a mais longa de duas maneiras — como uma cadeia de caracteres e como uma subcadeia — começando a partir do quinto caractere. Essa última operação é possível somente devido ao operador [ ] redefinido.

Listagem 5.15 Um programa driver para a classe definida no exemplo anterior (Listagem 5.14): *strops1.c*

```
main()
{
  string s("isto e apenas um teste"),    // inicializa duas cadeias
         t(" nada pode dar errado");

  s < &t;                                 // concatena-as

  cout << form("->%s\n",(char*)s); // imprime a nova cadeia, a maior

  for (int i=0; i<=s.len; i++) // imprime-a caractere a caractere
    cout << form("*%c",s[i]);

}
```

Uma Classe String Bidimensional

A aptidão para redefinir operadores rotineiros salienta o poder da sobrecarga de operador, mas também envolve algumas combinações de "truques" de velhos e novos procedimentos. Freqüentemente, no processo de redefinição de um operador, você precisa usar seu significado original com a função *operator( )*. Você tem visto que a maioria dos exemplos deste capítulo é assim. Por exemplo, para definir um operador = para uma classe, você precisa usar a força intrínseca do símbolo em operações internas no membro da classe. Essa mistura de níveis de significado pode ser confusa, mas geralmente é fácil distinguir. O mais difícil é combinar significados novos e velhos de operadores — não internamente numa função-membro — mas no próprio programa.

O programa *strops2.h* (Listagem 5.16) define uma classe simples. Essa classe armazena e manipula uma classe-cadeia de caracteres e, como aconteceu no exemplo anterior, *strops1.h* (Listagem 5.14), ela também redefine o operador endereçamento de vetor para permitir o acesso aos membros individuais na cadeia. Esse exemplo, entretanto, cria um vetor string bidimensional.

Listagem 5.16 Uma classe que ilustra a redefinição do operador [ ] e cria um vetor bidimensional: *strops2.h*

```
#include <stream.h>
#include <string.h>

class string {
  char* v;
public:
  string() { v = 0; }          // define um construtor simples
   void operator=(char*);      // declara um operador de atribuicao
  char* out() { return v; } // define um membro para saida
   char* operator[](int);      // declara um novo operador "[]"
};

void string::operator=(char* s)     // redefine a atribuicao
{
  char* buf;          // declara um buffer temporario
  buf=new char[strlen(s)+1];          // torna-o suficientemente
                                      // grande

  strcpy(buf,s);      // copia o valor de inicializacao
  delete v;           // remove qualquer valor antigo no tipo
  v=buf;              // atribui o novo valor
}

char* string::operator[](int pos) // define o novo operador "[]"
{
  if (pos > strlen(v))                // se o indice for muito grande,
                                      // retorna um nulo
    return 0;
  return v+pos-1;                     // caso contrario, retorna o
                                      // caractere especificado
}
```

Diferentemente dos exemplos de classes anteriores, a string definida neste exemplo está mais próxima do significado embutido de uma cadeia de caracteres em C++. Tanto essa classe quanto a noção mais tradicional usam um vetor de caracteres finalizado por um sinalizador especial, zero. A função construtor para esta classe, *string( )*, não lhe permite inicializar uma classe. Ela nem mesmo reserva memória; simplesmente garante a você que o caractere *v* tem inicialmente o valor zero. O programa não define uma função destrutor, embora alguém poderia normalmente estender esta classe.

Além da função construtor, a nova classe *string* contém uma função-membro de saída simples, *out( )*, que resulta no valor do membro *v* de armazenamento na forma de um ponteiro de caractere. Devido a *out( )* ser usada pelas funções em *stream.h*, onde são adequadas as cadeias de caracteres, não é necessário nenhum modelo explícito aqui. Esta função também é muito pequena para que você a declare como *inline*.

Um aperfeiçoamento desta cadeia de caracteres em relação à interna é a definição de um operador de atribuição (=). Esse membro permite-lhe usar a costumeira sintaxe de atribuição de outras espécies de dados em C++ com uma cadeia de caracteres. A codificação para operator=( ) é simples. Ela determina um buffer temporária que é suficientemente grande para conter o valor que está do lado direito do sinal =. Depois, ela copia aquele valor nesse buffer temporária (qualquer valor que estiver armazenado é removido por delete v). Finalmente, o programa copia o endereço do buffer em *v*. Esse operador de atribuição funciona de uma maneira consistente com a experiência diária de um programador a respeito de como funciona a atribuição.

A função *operator[ ]( )* é similar à definição em *strops1.h* (Listagem 5.14). Ela toma o valor do índice como parâmetro e soma o valor ao endereço armazenado em *v* para calcular a posição do caractere desejado. Essa classe oferece uma melhor verificação de erro. Se o valor do índice for maior do que o tamanho da cadeia de caracteres apontada por *v*, resulta um valor zero para indicar uma condição de erro.

O programa *strops2.c* (Listagem 5.17) é um simples driver que ilustra a operação da classe. Ele determina o único objeto e atribui a ele um valor literal. O resultado aparece na tela.

Listagem 5.17 Um programa driver que ilustra a classe na Listagem 5.16: *strops2.c*

```
main()
{
  string s;                          // declara um objeto string

  s = "isto e apenas um teste");     // atribui um valor a ele

  cout << form("%s\n",s[6]);         // imprime um elemento
                                     // arbitrario daquele valor
}
```

O programa *strops3.c* (Listagem 5.18) contém uma função *main( )* mais interessante. Esse programa define um vetor string bidimensional misturando dois níveis de abstração:

- um vetor de objetos de classe *string*, e
- um vetor redefinido para objetos *string*.

A codificação para se conseguir isso não está restrita somente à classe; ela também se relaciona com a forma pela qual a cadeia é endereçada na função *main( )*.

Listagem 5.18 Um programa driver que ilustra as classes na Listagem 5.16 e usa operadores embutidos redefinidos para criar um vetor bidimensional: *strops3.c*

```
main()
{
  string s[3];     // declara um vetor como objeto string

  s[0] = "isto e apenas um teste";  // atribui por sua vez um
                                    // valor a cada elemento
  s[1] = "nada pode sair errado";
  s[2] = "... sair errado ...";

  for (int i=0; i<3; i++)  // imprime somente o segundo caractere
    cout << form("%s\n",s[i][2]);   // de cada elemento do vetor
}
```

A primeira instrução da função *main( )* usa a sintaxe embutida da criação de vetores para determinar um vetor de variáveis string. Cada elemento desse vetor representa uma variável string e pode ter atribuído a si um valor. Por exemplo:

```
s[0] = "isto e apenas um teste";
```

coloca o valor dessa cadeia literal no primeiro elemento do vetor. A função *main()* executa uma atribuição similar para cada elemento.

Após as variáveis terem sido preenchidas com valores, um loop *for* mostra-as na tela. No entanto, neste caso, o programa mostra as cadeias somente a partir do segundo caractere, ficando assim:

```
isto e apenas um teste
ada pode sair errado
.. sair errado ...
```

A operação acima é executada no programa pela seguinte instrução:

```
cout << form("%s\n",s[i][2];
```

Você pode usar essa forma para uma variável desse tipo porque o operador [ ] redefinido permite a indexação de classes. A primeira série de colchetes extrai do vetor um elemento da classe. O par de colchetes final opera internamente neste objeto. Assim, o conjunto final é uma *string* bidimensional.

Resumo

Este capítulo apresentou a sobrecarga quando aplicada a operadores comuns em C++. Ele mostrou como redefinir símbolos com + e - para funcionar como tipos de classes definidas pelo usuário da mesma maneira conveniente que eles funcionam com tipos embutidos, tais como *int*, *double* e *char*. No universo C++, faz sentido tanto "somar" duas cadeias de caracteres quanto executar esta mesma operação com números de precisão dupla.

O segredo da sobrecarga reside na função

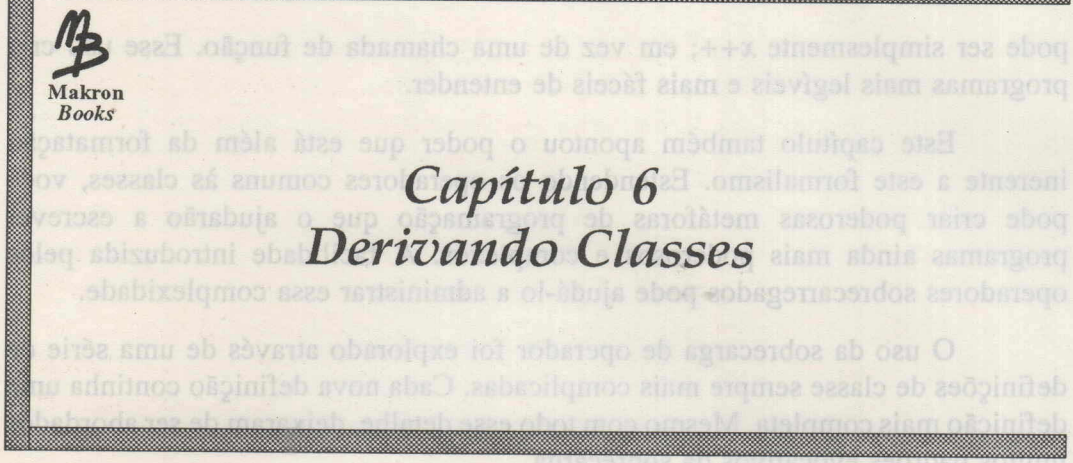
operator *ops( )*;

Nesta função, *ops* pode ser qualquer símbolo definido de operador, inclusive aqueles compostos como ++ e --. Além do mais, essa função pode ser um membro de qualquer classe. Ela cria uma seqüência de chamada especial que reflete o uso costumeiro do operador. Para a função *operator++( )*, a instrução para chamá-la

pode ser simplesmente *x*++; em vez de uma chamada de função. Esse uso cria programas mais legíveis e mais fáceis de entender.

Este capítulo também apontou o poder que está além da formatação inerente a este formalismo. Estendendo os operadores comuns às classes, você pode criar poderosas metáforas de programação que o ajudarão a escrever programas ainda mais poderosos e complexos. A facilidade introduzida pelos operadores sobrecarregados pode ajudá-lo a administrar essa complexidade.

O uso da sobrecarga de operador foi explorado através de uma série de definições de classe sempre mais complicadas. Cada nova definição continha uma definição mais completa. Mesmo com todo esse detalhe, deixaram de ser abordados muitos padrões evocativos da sobrecarga.



Capítulo 6
Derivando Classes

Classes derivadas como ferramenta de desenvolvimento
Criando uma classe derivada
Classes derivadas com construtores e destrutores
Classes de base pública e privada
Funções virtuais
Sistemas complexos de classes
Resumo

Antes de examinar relações avançadas de classes, vamos rever brevemente o tipo de dado *classe*. Essa estrutura contém todos os elementos que compõem uma programação para objeto os membros que armazenam itens de dados e as funções-membro que manipulam os dados e se comunicam com o resto do programa. A mais importante entre as funções de classe é a função construtor, que é chamada quando é criada uma classe. Estas funções-membro podem executar inicializações complexas e extensivas e outras ações de partida que às vezes são necessárias nas classes complicadas.

As classes são especialmente importantes porque ajudam a implementar modelos que possibilitam aos programas de computador simular as situações complexas da vida real. As classes administram eficientemente a complexidade de grandes programas através da modularização. Cada classe contém uma parte pública e uma parte privada. Os membros — tanto variáveis quanto funções — declarados na parte privada da classe são ocultos ao resto do programa. A seção pública é a interface que une os objetos ao restante do ambiente do software. Isso

poupa você dos detalhes da implementação de um programa peça por peça. À medida que você completa cada seção com sucesso, pode ocultar do resto do programa todos os detalhes da implementação, de maneira que somente a pequena interface pública pode afetar — ou pode ser afetada — por outros módulos.

No entanto, a classe somente não é suficiente para apoiar totalmente a programação e projeto objeto-orientada. Você precisa também uma maneira de relacionar classes desiguais umas com as outras. Os primeiros exemplos usaram classes aninhadas, mas estas constituem apenas uma solução parcial porque as definições de classes aninhadas não são ocultas — mesmo quando são definidas na parte privada de uma classe abrangente. Em nenhum caso uma classe aninhada tem os requisitos necessários para relacionar totalmente uma classe com outra. O que se necessita é um mecanismo para criar hierarquias de classes no qual uma classe A é *cognata* da classe B, mas com características adicionais. C++ tem um mecanismo para criar essa hierarquia — a classe derivada. O processo de derivação de classes é chamado *herança*, e é uma das características que distinguem as linguagens objeto-orientadas.

Classes Derivadas como Ferramenta de Desenvolvimento

Uma das características mais marcantes da linguagem C++ é a maneira como ela permite construir classes, uma sobre a outra. Após você definir uma classe, você pode usá-la como base para uma família inteira de classes relacionadas ou *derivadas*. Por exemplo, na biblioteca gráfica, você pode definir uma classe *retângulo*. Quando você precisa definir um novo tipo de retângulo, por exemplo, um retângulo com cantos arredondados, pode usar os recursos básicos da classe *retângulo* e depois acrescentar algumas propriedades únicas para derivar uma nova classe de retângulos (veja Figura 6.1).

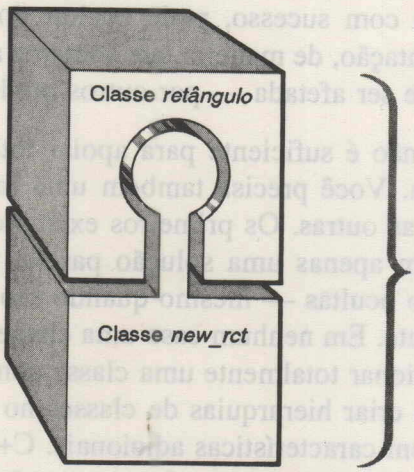


Figura 6.1 Derivando uma nova classe *retângulo*.

Classes Derivadas para Melhor Representação dos Dados

Como exemplo mais prático, considere um programa financeiro no qual uma classe de contabilidade contém a identificação de um cliente como classe básica, e acrescenta informações específicas às várias espécies de contas como classes derivadas. O assunto original é preservado; ele é simplesmente reproduzido de maneira que toda a sua informação fique disponível para cada classe derivada que segue. Esse mecanismo ajuda você a produzir programas mais eficientes e elegantes (veja Figura 6.2). Estes exemplos sugerem como a herança permite-lhe mais facilmente modelar os objetos da vida real e suas relações, sem ter que usar obscuras instruções condicionais e desvios encontrados na abordagem C tradicional.

Classes Derivadas Promovem Modularidade

Além do seu uso como meio de representar características comuns e aspectos especializados de diferentes estruturas de dados, a derivação de classe é

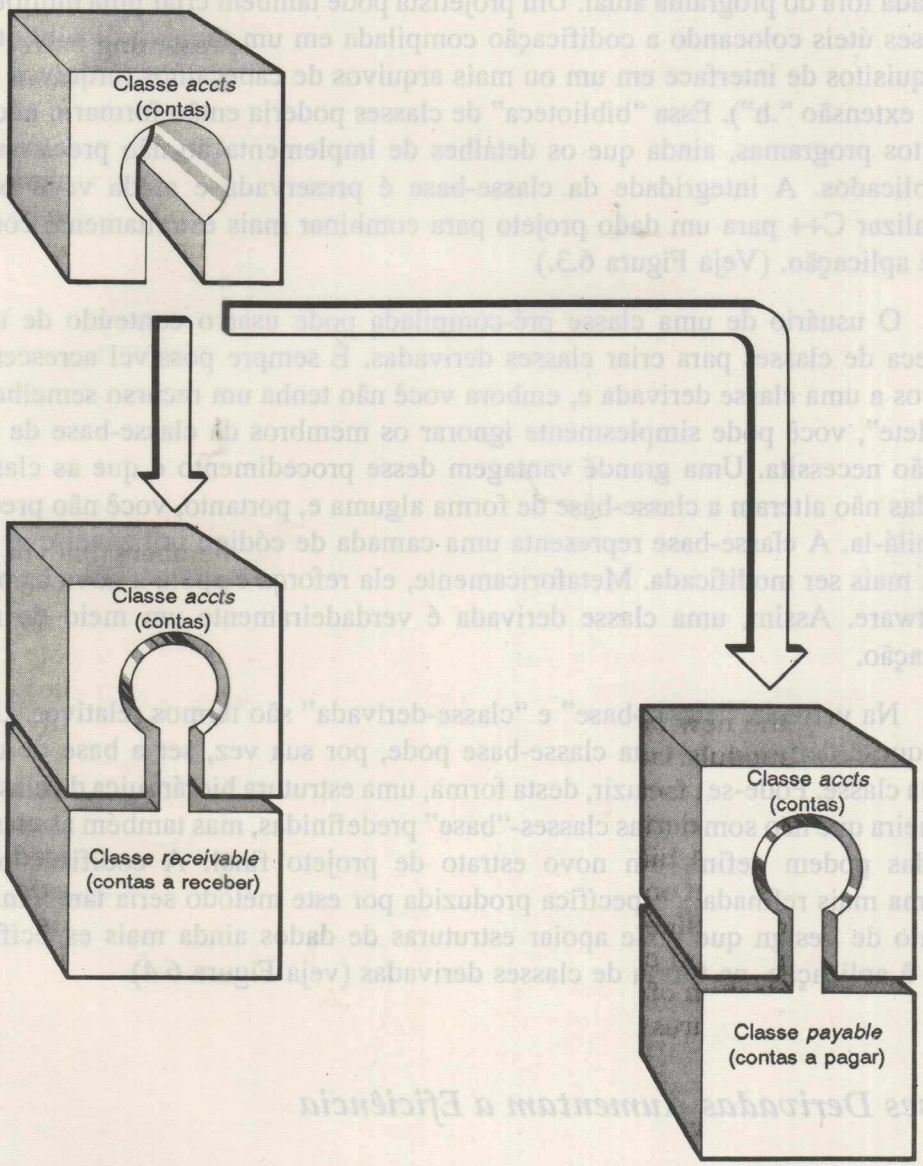


Figura 6.2 Derivando classes especializada para contabilidade.

também uma ferramenta para modularização. Por exemplo, um programador pode definir uma classe derivada baseada numa classe existente que foi produzida e

compilada fora do programa atual. Um projetista pode também criar uma biblioteca de classes úteis colocando a codificação compilada em um arquivo de biblioteca e os requisitos de interface em um ou mais arquivos de cabeçalhos (arquivos que usam a extensão ".h"). Essa "biblioteca" de classes poderia então formar o núcleo de muitos programas, ainda que os detalhes de implementação não precisassem ser publicados. A integridade da classe-base é preservada e ainda você pode personalizar C++ para um dado projeto para combinar mais estreitamente com a área de aplicação. (Veja Figura 6.3.)

O usuário de uma classe pré-compilada pode usar o conteúdo de uma biblioteca de classes para criar classes derivadas. É sempre possível acrescentar membros a uma classe derivada e, embora você não tenha um recurso semelhante ao "delete", você pode simplesmente ignorar os membros da classe-base de que você não necessita. Uma grande vantagem desse procedimento é que as classes derivadas não alteram a classe-base de forma alguma e, portanto, você não precisa recompilá-la. A classe-base representa uma camada de código utilizável que não precisa mais ser modificada. Metaforicamente, ela reforça e apóia a nova camada do software. Assim, uma classe derivada é verdadeiramente um meio de modularização.

Na verdade, "classe-base" e "classe-derivada" são termos relativos. Uma classe que é derivada de uma classe-base pode, por sua vez, ser a base de uma segunda classe. Pode-se produzir, desta forma, uma estrutura hierárquica de classes, de maneira que não somente as classes-"base" predefinidas, mas também as classes derivadas podem definir um novo estrato de projeto final. A codificação de programa mais refinada e específica produzida por este método seria também um substrato de design que pode apoiar estruturas de dados ainda mais específicas quanto à aplicação, na forma de classes derivadas (veja Figura 6.4).

Classes Derivadas Aumentam a Eficiência

O uso de classes derivadas aumenta também a eficiência. Você não precisa mais criar códigos repetitivos; em cada nível, os detalhes dos níveis inferiores podem ser agrupados com qualquer processamento necessário. Sem dúvida, você não precisa mais criar uma série de estruturas de dados independentes e redundantes para cuidar de qualquer variação possível na organização dos dados usados em um

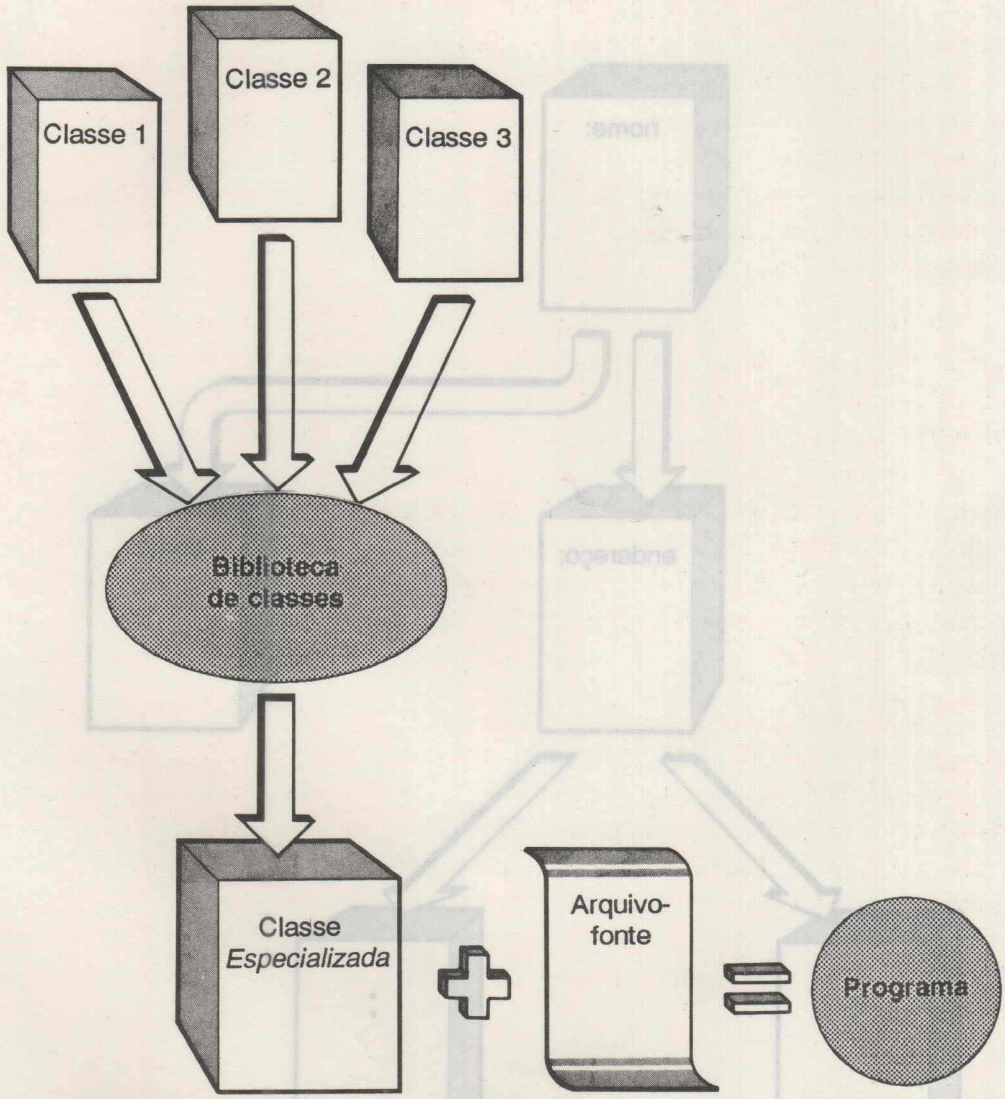


Figura 6.3 Usando uma biblioteca de classe.

programa. Por exemplo, se um programa de contabilidade deve manter registro de indivíduos com múltiplas contas, você não precisa mais acrescentar códigos para unir a informação de identificação comum com uma série de estruturas de dados únicas, uma para cada tipo de conta.

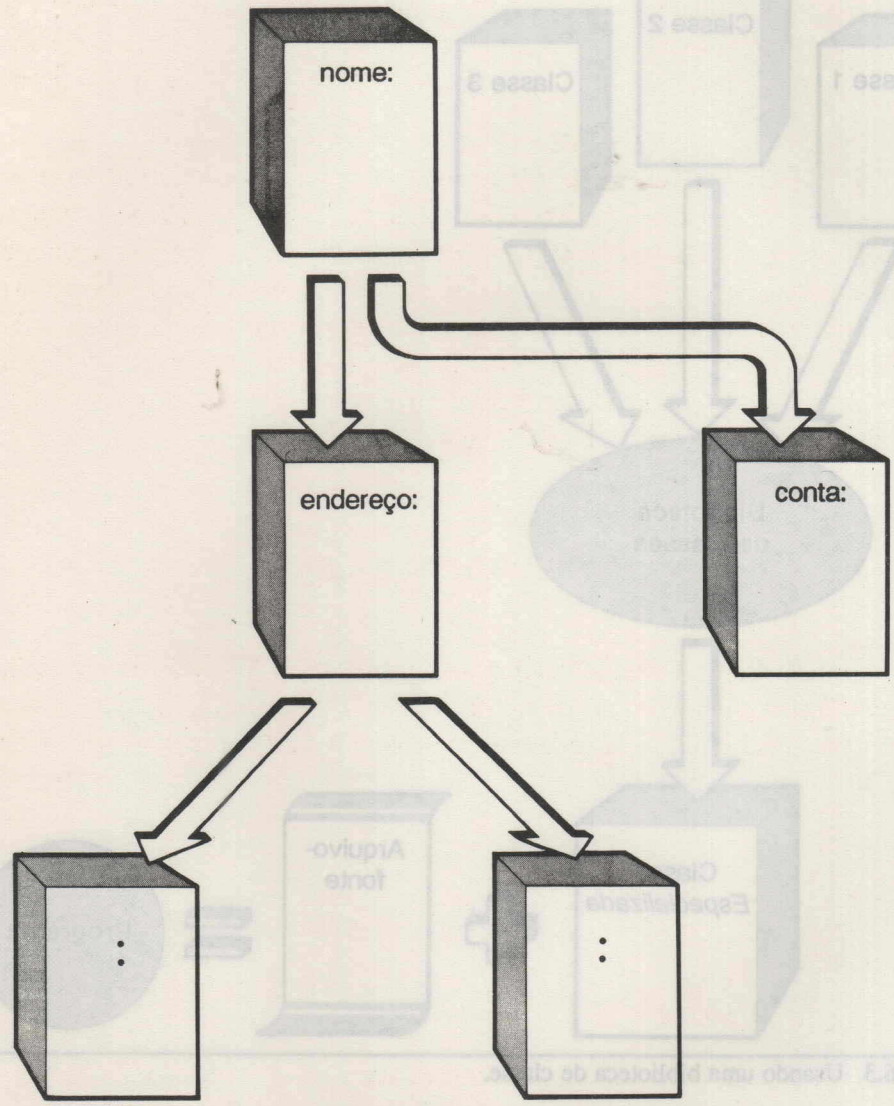


Figura 6.4 Uma estrutura hierárquica de classe.

Em C++ você pode criar a mesma estrutura usando uma série de classes derivadas — mantém a informação de identificação na classe-base e coloca os

detalhes do tipo de conta na classe derivada apropriada. O nome e outros dados de identificação do cliente estão ainda disponíveis, mas agora eles estão retidos dentro da própria *conta*. Isso lhe dá duas vantagens imediatas: você não precisa criar uma estrutura complexa de condicionais para ligar uma série de valores a outra, e nem precisa incluir o nome do cliente toda vez que achar que o programa precise.

As classes derivadas permitem que o programador economize muito trabalho, porque os blocos de informação redundante não precisam ser mantidos. Usar estruturas de dados tradicionais, a cada vez que um campo ou membro é repetido, aumenta o uso da memória e o acesso fica mais lento. Os programas também requerem mais tempo para avaliar as expressões booleanas e para trabalhar através de séries complexas de condicionais com o fim de acessar a série de dados desejada. Em C++, a derivação de classes dispõe a hierarquia dos dados de uma maneira altamente otimizada e transparente.

Derivação de Classe: Um Estudo de Caso

Vamos examinar uma situação que ocorre com freqüência na programação — administrar uma série de informações. Neste exemplo, imagine que você tenha que lidar com registros de estudantes, consistindo cada um nas seguintes informações:

nome

ano escolar (calouro, segundanista etc.)

alojamento

número do quarto

Você precisa manipular estes valores para gerar faturas dos quartos e das refeições, ou para criar um cadastro postal de todos os veteranos. Numa primeira consideração, você talvez decida fazer os itens de registro membros de uma só classe; você poderá até mesmo usar uma estrutura para tornar isso mais "semelhante a C". No entanto, antes de você se entregar a esta formulação, examine os dados mais cuidadosamente. Eles se dividem naturalmente em pelo menos duas partes: "alojamento" e "número do quarto", definindo assim o endereço do estudante.

"nome" e "ano escolar" identificam os indivíduos. (Na realidade, a situação é mais complexa, porque é necessário acrescentar informações adicionais como série e matérias, por exemplo.) Entretanto, a questão é que a informação tende a se dividir em grupos relacionados. Estes grupos, por sua vez, estão relacionados somente pelo fato de que pertencem ao registro de um indivíduo.

Usualmente manipulam-se situações semelhantes criando-se dados individuais que abrangem cada grupo de informação relacionada. Tipicamente, esses dados são ligados entre si por uma de duas maneiras diferentes — alojando objetos individuais numa espécie de "superobjeto", ou criando um valor comum em cada dado independente. (Veja Figura 6.5.)

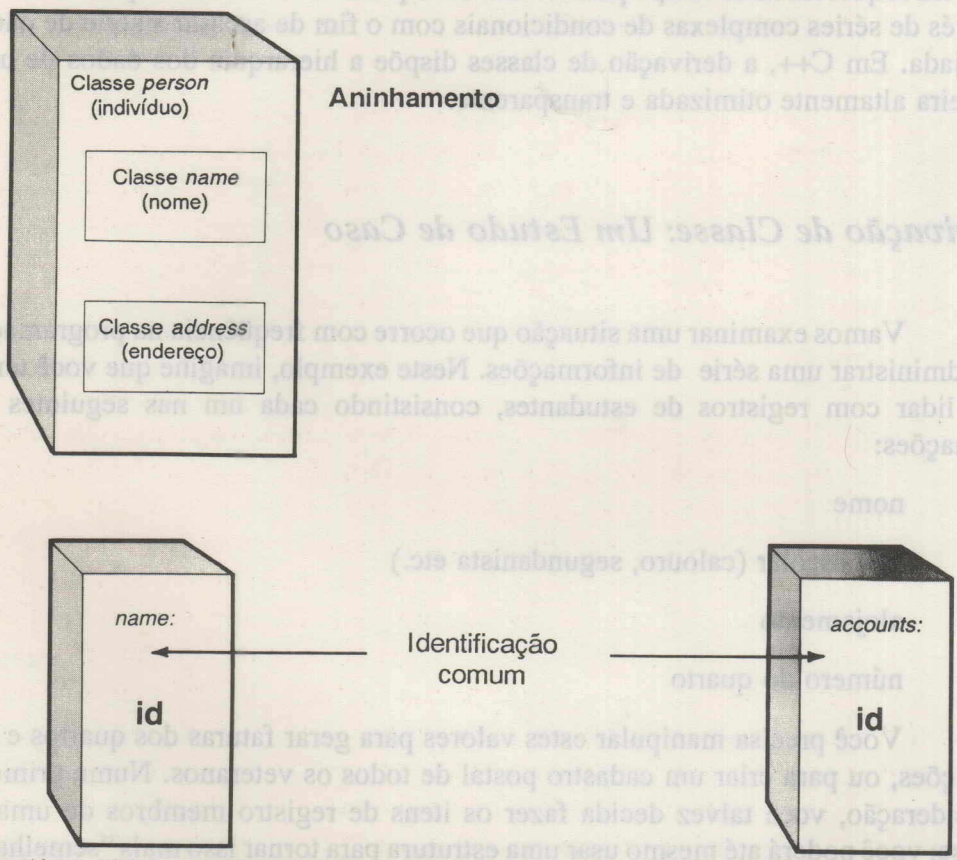


Figura 6.5 Estratégias para informação relacionada.

O primeiro caso pode ser representado pelo uso de membros de classe dentro de outra classe — uma declaração aninhada. No segundo caso você pode, por exemplo, definir cada grupo como uma classe e colocar um nome nela. Este membro da classe constituído pelo nome unirá os demais dados dispersos. Naturalmente, você deve implementar essa conexão explicitamente — e muito visivelmente — na codificação do programa.

A maior desvantagem neste tipo de solução é que a abstração de dados é implementada de maneira imperfeita. Mesmo se você define uma classe dentro da parte privada de outra classe, o escopo da classe privada não está realmente oculto do resto do programa. Em muitos casos isso pode não ser uma consideração importante. No entanto, o escopo da definição é diferente do escopo dos demais itens da classe, e essa disparidade pode ser causa de confusão num programa complicado.

A Figura 6.6 ilustra outra abordagem para o problema de se combinar dados relacionados. Aqui, a primeira informação sobre o estudante está dividida em duas partes relacionadas — uma contém a informação de identificação e a outra contém o endereço. Isso é exatamente o que a linguagem C++ cria quando deriva uma classe de outra. A área onde está escrito "on_campus" é derivada da outra área onde está escrito "student"; ela contém toda a informação sobre este último objeto e mais a informação referente ao endereço, que é local em relação a ele.

A derivação, como está na Figura 6.6, é muito diferente do simples procedimento de aninhamento mencionado anteriormente. O objeto derivado é, de certa forma, ainda do mesmo tipo que a base; eles podem até compartilhar dos mesmos ponteiros! Em contrapartida, o relacionamento entre duas classes aninhadas é o da justaposição; eles estão próximos um do outro na codificação do programa, mas não têm nenhuma conexão orgânica. De um modo particular, a classe derivada tem as mesmas aptidões de sua base, mais algumas adicionais. Você não precisa redeclarar os membros da base porque eles estão disponíveis na classe derivada. E, o que é mais importante, se a classe-base e todas as suas funções-membro estão definidas, compiladas e armazenadas em um arquivo, você pode usar uma classe derivada para mudar a definição da relação de dados sem perturbar qualquer aspecto da classe-base. Qualquer alteração numa classe aninhada — não importa quão pequena seja — requer a recompilação de todos os seus programas e definições associados.

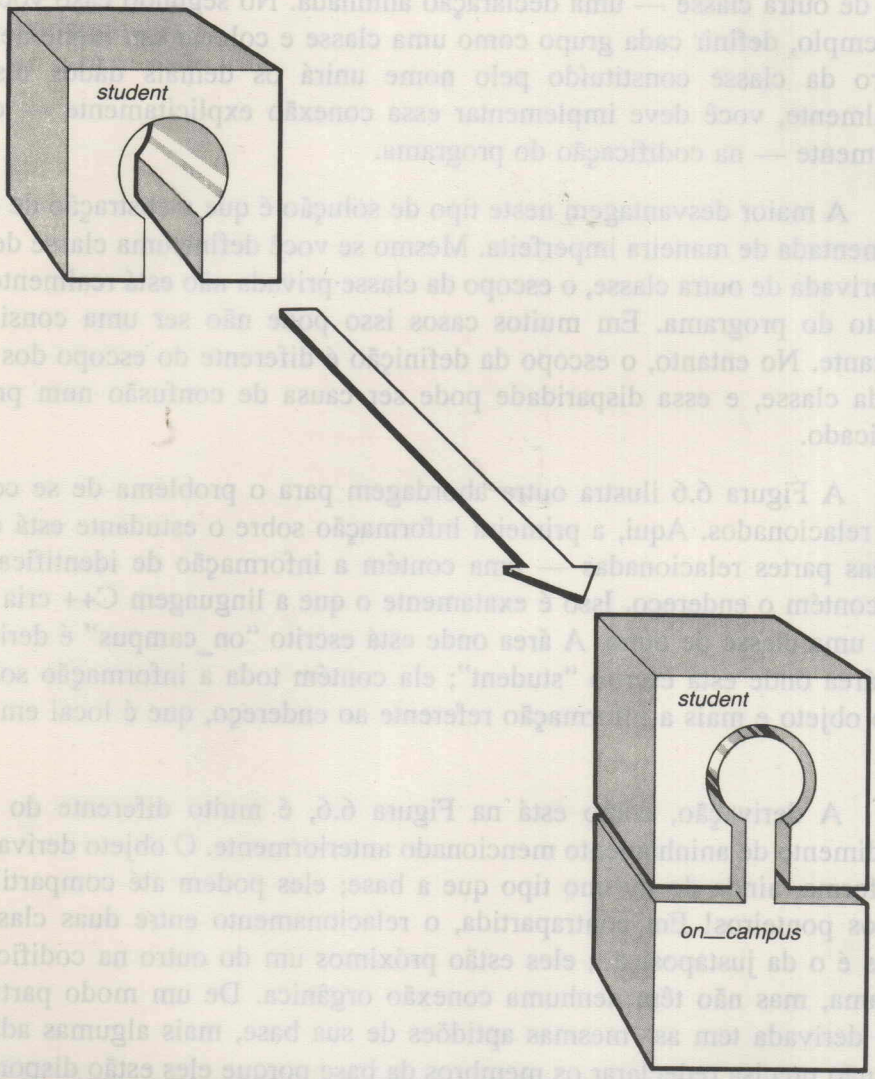


Figura 6.6 Uma solução melhor para relacionamento de dados.

Uma vantagem desta redefinibilidade ampliada é que ela permite que o programador facilmente defina variações num dado similar. Fazendo uma extensão mais real do exemplo anterior, você deve dar margem ao fato de que alguns estudantes não moram nos alojamentos, e, portanto, o programa precisará de

campos diferentes para seus endereços. Para resolver essa situação, você deriva uma classe que usa a classe-base, mas tem uma série diferente de elementos de endereço — rua, cidade, estado e CEP (em vez de alojamento e quarto). Agora, você tem uma relação entre uma classe-base e duas classes derivadas. Esta relação está esquematizada na Figura 6.7.

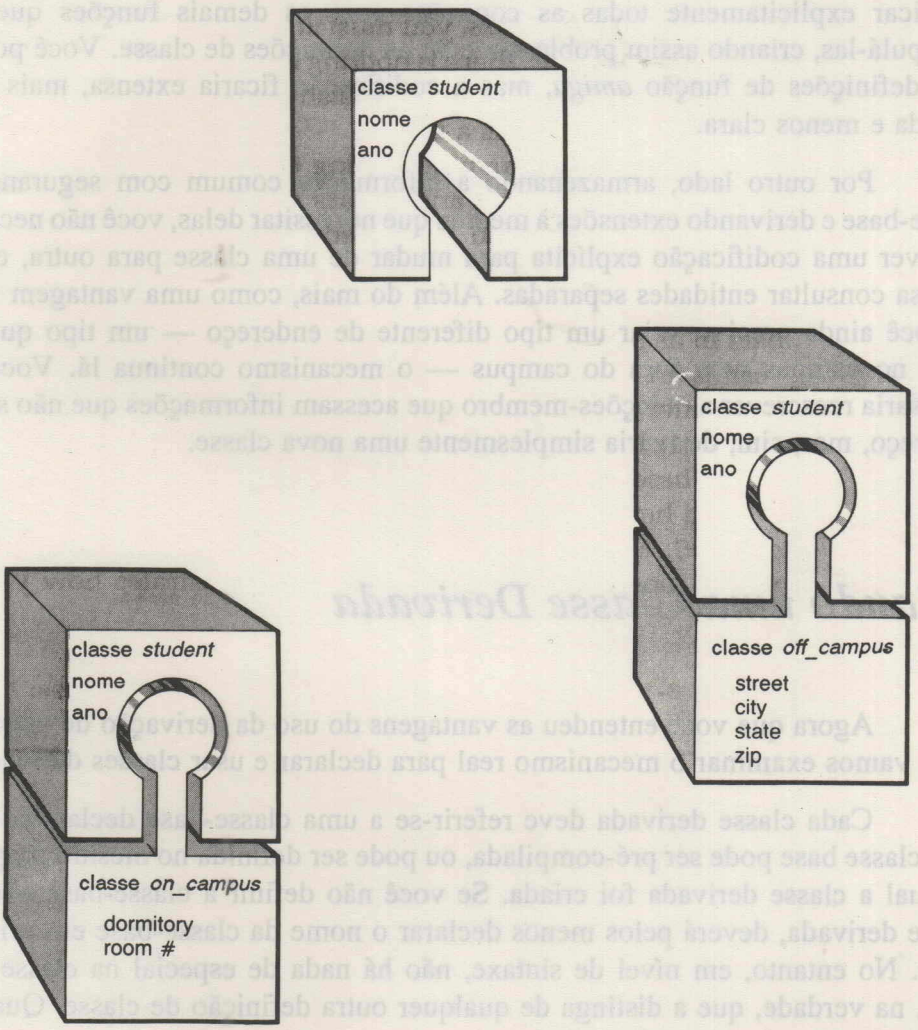


Figura 6.7 Uma classe-base e duas classes derivadas.

Sem o recurso de derivar classes, você precisaria criar duas entidades separadas e incompatíveis, uma para armazenar os endereços daqueles estudantes que moram no campus e outra para aqueles que moram fora do campus. As ligações entre as duas entidades teriam de envolver informação redundante — um nome, ou, mais provavelmente, um único campo de identificação. Depois, você precisaria codificar explicitamente todas as conexões com as demais funções que irão manipulá-las, criando assim problemas com as definições de classe. Você poderia usar definições de função *amiga*, mas a codificação ficaria extensa, mais complicada e menos clara.

Por outro lado, armazenando a informação comum com segurança na classe-base e derivando extensões à medida que necessitar delas, você não necessita escrever uma codificação explícita para mudar de uma classe para outra, e nem precisa consultar entidades separadas. Além do mais, como uma vantagem extra, se você ainda precisar criar um tipo diferente de endereço — um tipo que não fosse no campus nem fora do campus — o mecanismo continua lá. Você não precisaria reescrever as funções-membro que acessam informações que não são de endereço, mas, sim, derivaria simplesmente uma nova classe.

Criando uma Classe Derivada

Agora que você entendeu as vantagens do uso da derivação de classe em C++, vamos examinar o mecanismo real para declarar e usar classes derivadas.

Cada classe derivada deve referir-se a uma classe-base declarada antes. Esta classe base pode ser pré-compilada, ou pode ser definida no mesmo programa no qual a classe derivada foi criada. Se você não definir a classe-base antes da classe derivada, deverá pelos menos declarar o nome da classe-base em primeiro lugar. No entanto, em nível de sintaxe, não há nada de especial na classe-base, nada, na verdade, que a distinga de qualquer outra definição de classe. Qualquer classe pode servir como classe-base.

O exemplo *student.c* (Listagem 6.1) ilustra uma família simples que consiste em uma classe-base e uma classe dela derivada. Este exemplo implementa a situação descrita na seção anterior e esquematizada na Figura 6.7; ele demonstra,

também, a sintaxe-base do procedimento. A definição de *student*, a classe-base, contém dois membros em sua seção privada — *nome* é um ponteiro para o nome do estudante, e *ano* é uma variável inteira que indica o grau em que se encontra o estudante. Esta última variável contém um número que indica calouro (O), segundanista (1), terceiroanista (2), ou quartoanista (3). (A escolha de valores é arbitrária e é baseada na tradição de programação C e C++.) Não é definida aqui nenhuma função construtor, embora a classe inclua uma função de impressão, *display( )*, e uma função que cria novas entradas, *create_rec( )*. A função *main( )* é uma simples função atuadora que ilustra como você usa estas classes.

Listagem 6.1 Uma definição de classe-base e derivada: *student.c*

```
#include <stream.h>
#include <string.h>

////////////////////////////////////////////////
// Define uma classe base
////////////////////////////////////////////////

class student {
  char* name;        // nome do estudante
  int year;       // grau atual; 1=calouro, etc

public:
  void display();                 // mostra o nome e o ano
  void create_rec(char*, int);    // coloca dados num registro
};

void student::display()
{
  cout << form("\nnome: %s\nano: %d\n",name,year);  // mostra-os
                                                    // na tela
}

void student::create_rec(char* n, int y)
{
  name = new char[strlen(n) + 1];   // aloca memoria suficiente
  strcpy(name,n);                   // copia o nome para a classe
  year = y;                         // ... e o ano
}

////////////////////////////////////////////////
// Define uma classe derivada de student
////////////////////////////////////////////////
```

```
class on_campus : student {
  char *dorm,                    // informacao adicional
    *room;                       // ao assunto estudante
public:
  void a_disp();                 // declara um membro para exibicao
                                 // simples
  void new_student(char*,int,char*,char*);   // ... e um para
                                             // entrada de dados
};

void on_campus::a_disp()
{
  display();                     // mostra os valores da classe base

  cout << form("dormitorio: %s\nquarto: %s\n",dorm,room);
                                             // ... depois os novos
}

void on_campus::new_student(char* n, int y, char* d, char* r)
{
  create_rec(n,y);               // introduz os valores base
  dorm = new char[strlen(d) + 1];   // aloca memoria
  strcpy(dorm,d);                   // copia
  room = new char[strlen(r) + 1];   // ... e para este campo
  strcpy(room,r);
}

// uma funcao driver simples para testar a nova classe

main()
{
  cout << "\n==================================\n";

  student x;                     // inicia com a classe base
  x.create_rec("Joe Smith",1);   // preenche-a
  x.display();                   // ... e mostra-a

  cout << "\n==================================\n";

  on_campus y;                                // uma classe derivada
  y.new_student("Hank Jones",1,"Foll Hall","L304");  // coloca os
                                                     // valores e
```

```
  y.a_disp();                                    // exibe-os
}
```

A parte mais interessante deste exemplo é a classe derivada *on_campus*. A primeira linha:

```
class on_campus" : student {
```

declara que esta é uma classe derivada e também que a classe-base é *student*. O restante da definição desta nova classe é quase tão imediato quanto aquela da base. Embora não use também nenhuma função-membro construtor, ela tem uma função de exibição, *a_disp( )*, e uma função que acrescenta objetos à classe, *new_student( )*. Esta última função pega os valores que estão na nova classe *on_campus*.

Acesso à Classe-Base

Agora, vamos examinar a relação entre a classe-base e uma classe derivada. A classe *on_campus* contém toda a informação contida em sua base, *on_student*, mas — e isso é uma consideração importante — não tem nenhum acesso especial à parte privada da classe-base. Sem a função-membro *display( )* da classe *student*, *o membro a_disp( )* de *on_campus* poderia amostrar apenas o nome do alojamento e o número do quarto. Há uma maneira de fugir desse problema — a seção protegida (discutida mais adiante neste capítulo) — mas, por enquanto, isso representa uma grande limitação ao relacionamento. Observe, no entanto, que as funções-membro públicas da classe-base estão disponíveis à classe derivada, bem como estariam com qualquer outra classe. Assim, por exemplo, a função-membro *a_disp( )* de *on_campus* pode chamar livremente a função *display( )* sem qualquer outra referência a um elemento do tipo *student*, porque qualquer função-membro pública da classe-base é automaticamente uma função-membro da classe derivada. Semelhantemente, *new_student( )* depende da função-membro *create_rec( )* de *student* para inicializar os dados básicos na classe-nome e ano escolar. Da mesma maneira que em *a_disp( )*, este membro é também uma parte da classe *on_campus*.

Classes Derivadas Múltiplas

O exemplo *student2.c* (Listagem 6.2) contém uma nova classe derivada, *off_campus*, que usa a classe *student* como base. Ela contém também uma nova função *main( )* que demonstra a interação da hierarquia de classe agora em três partes. A classe *off_campus* contém um formato de endereço alternativo — o tradicional rua, número, cidade, estado e CEP. Ela usa também uma função-membro *a_disp( )* para mostrar os valores atuais, e uma função-membro *new_student( )* para inicializá-los.

Listagem 6.2 Acrescentando nova classe derivada à classe-base definida na Listagem 6.1: *student2.c*

```
////////////////////////////////////////////////
// uma outra classe derivada de student
////////////////////////////////////////////////

class off_campus : student {
  char *street,                 // nova informacao para adicionar
       *city,                   // aos dados base de student
       *state,
       *zip;

public:
  void a_disp();          // declara um membro para exibicao simples
  void new_student(char*,int,char*,char*,char*,char*);
                                // ... e um para entrada de dados
};

void off_campus::a_disp()
{
  display();              // coloca os valores da classe base na tela
                          // ... entao, a nova informacao
  cout << form("endereco: %s\n%s, %s %s\n",street,city,state,zip);
}

void off_campus::new_student(char* n,int y,char* s,char* cty,
                             char* st,char* z)
{
  create_rec(n,y);                      // coleta os valores comuns
  street = new char[strlen(s) + 1];     // aloca memoria e
  strcpy(street,s);                     // copia  a nova informacao
```

```
  city = new char[strlen(cty) + 1];
  strcpy(city,cty);
  state = new char[strlen(st) + 1];
  strcpy(state,st);
  zip = new char[strlen(z) + 1];
  strcpy(zip,z);
}

////////////////////////////////////////////////////////
// uma funcao driver simples para ilustrar a familia
// de classes baseadas na student
////////////////////////////////////////////////////////

main()
{
  cout << "\n===================================\n";

  student x;              // inicia com a classe base
  x.create_rec("Joe Smith",1);    // preenche-a
  x.display();                    // ... e mostra-a

  cout << "\n===================================\n";

  on_campus y;                                  // uma classe derivada
  y.new_student("Hank Jones",1,"Foll Hall","L304");  // coloca os
                                                     // valores e
  y.a_disp();                                        // exibe-os

  cout << "\n===================================\n";

  off_campus z;       // cria e exibe um objeto de uma outra classe
  z.new_student("Sally Green",2,"22 Main Str","San Francisco",
                "CA", "94113");
  z.a_disp();

}
```

Quando você compara as duas classes derivadas, *on_campus* e *off_campus*, observa que suas funções-membro têm os mesmos nomes. Fica claro também, que estas funções têm nomes diferentes de sua correspondentes na classe-base. A função *display( )* mostra os valores que são partes de *student*, mas a função *a_disp( )* mostra a informação referente ao endereço das classes *on_campus e off_campus*. Devido ao fato de todos os membros públicos da classe-base estarem disponíveis

à classe derivada, quando você usa o mesmo nome da função-membro na base, como faz na classe derivada, a função-membro da base torna-se não disponível à classe derivada. Em algumas circunstâncias, isso pode ser desejável. Se, por exemplo, você usa o nome "display" em alguma das classes derivadas, isso iria mascarar a função-membro *display( )* na classe-base *student*. Você pode também usar os mesmos nomes de membros nas duas classes derivadas. No entanto, eles são, naturalmente, mutuamente exclusivos.

Numa classe derivada, somente as funções-membro podem chamar os membros da classe-base. Os dois exemplos de classes derivadas são simétricos no estilo, de maneira que o que for verdadeiro para um será também verdadeiro para outro. Na função *a_disp( )* o membro da classe-base *display( )* é solicitado para mostrar a informação que se encontra na classe-base — especificamente, o nome e o ano escolar. O resto dessa função usa *cout* para imprimir a informação específica na classe derivada alojamento, quarto e número, no caso de *on_campus*, e um endereço comum para *off_campus*. Em *new_student( )* é usada a outra função-membro da classe-base; esta solicita a função-membro *create_rec( )* da classe-base na primeira instrução.

Referências Explícitas aos Membros

Uma construção alternativa que permite o acesso a qualquer função-membro independente do nome, usa o operador referência (::) de C++ para criar uma referência explícita a um membro em particular. A definição de classe que se encontra em *student3.c* (Listagem 6.3) usa este procedimento. Aqui, o nome-base *student* vem seguido de :: e do nome do membro, *display( )*. Nem sempre quando você está usando definições de classe predefinidas — e geralmente pré-compiladas — você tem completa liberdade para escolher nomes de variáveis, de maneira que é bom ter um mecanismo que lhe permita escolher um nome fora do presente escopo.

Listagem 6.3 Uma hierarquia da classe *student* redefinida usando operadores de referência explícita: *student3.c*

```
#include <stream.h>
#include <string.h>
```

```
////////////////////////////////////////////////
// Define uma classe base
////////////////////////////////////////////////

class student {
  char* name;      // nome do estudante
  int year;     // grau atual; 1=calouro etc.

public:
  void display();                  // mostra o nome e o ano
  void create_rec(char*, int);     // coloca dados num registro
};

void student::display()
{
  cout << form("\nnome: %s\nano: %d\n",name,year);  // mostra-os
                                                    // na tela
}

void student::create_rec(char* n, int y)
{
  name = new char[strlen(n) + 1];    // aloca memoria suficiente
  strcpy(name,n);                    // copia o nome para a classe
  year = y;                          // ... e o ano
}
////////////////////////////////////////////////
// Define uma classe derivada de student
////////////////////////////////////////////////

class on_campus : student {
  char  *dorm,                 // informacao adicional
        *room;                 // ao assunto estudante
public:
  void display();       // declara um membro para exibicao simples
  void create_rec(char*,int,char*,char*);     // ... e um para
                                              // entrada de dados
};

void on_campus::display()
{
  student::display();      // chama explicitamente o membro student
```

```
  cout << form("dormitorio: %s\nquarto: %s\n",dorm,room);
                                        // ... depois os novos
}

void on_campus::create_rec(char* n, int y, char* d, char* r)
{
  student::create_rec(n,y);             // o membro base
  dorm = new char[strlen(d) + 1];       // aloca memoria
  strcpy(dorm,d);                       // copia
  room = new char[strlen(r) + 1];       // ... e para este campo
  strcpy(room,r);
}


////////////////////////////////////////////////
// uma outra classe derivada de student
////////////////////////////////////////////////

class off_campus : student {
  char  *street,                // nova informacao para adicionar
        *city,                  // aos dados base de student
        *state,
        *zip;
public:
  void display();     // declara um membro para exibicao simples
  void create_rec(char*,int,char*,char*,char*,char*); // ... e um
                                        // para entrada de dados
};

void off_campus::display()
{
  student::display();           // membro da base, nao o membro atual
                                // ... entao, a nova informacao
  cout << form("endereco: %s\n%s, %s %s\n",street,city,state,zip);
}

void off_campus::create_rec(char* n,int y,char* s,char* cty,char*
st,char* z)
{
  student::create_rec(n,y);         // uma outra chamada explicita

  street = new char[strlen(s) + 1];   // aloca memoria e
  strcpy(street,s);                   // copia  a nova informacao
  city = new char[strlen(cty) + 1];
```

```
  strcpy(city,cty);
  state = new char[strlen(st) + 1];
  strcpy(state,st);
  zip = new char[strlen(z) + 1];
  strcpy(zip,z);
}

////////////////////////////////////////////////////////

main()
{
  cout << "\n===================================\n";

  student x;              // inicia com a classe base
  x.create_rec("Joe Smith",1);     // preenche-a
  x.display();                     // ... e mostra-a

  cout << "\n===================================\n";

  on_campus y;                                  // uma classe derivada
  y.create_rec("Hank Jones",1,"Foll Hall","L304");   // coloca os
  y.display();                                  // valores e exibe-os

  cout << "\n===================================\n";

  off_campus z;       // cria e exibe um objeto de uma outra classe
  z.create_rec("Sally Green",2,"22 Main Str","San
Francisco","CA","94113");
  z.display();
}
```

Esse exemplo contém a hierarquia completa derivada da classe *student*. A diferença aqui é que cada classe tem uma função-membro *display( )* e uma função-membro *create_rec( )*. Dentro do membro *display* de cada classe derivada está uma referência explícita a uma função-base *display( )*. Há também uma referência explícita semelhante à versão *student* de *create_rec( )* na definição desse membro. A função *main( )* ilustra o uso destes objetos.

Questões de Revisão

1. Uma classe derivada refere-se a uma classe _____ predefinida.
2. Uma classe derivada tem acesso aos membros _____ de sua classe-base.
3. A classe derivada contém a _____ informação que se encontra na classe-base, mais a sua própria informação local.

Projeto de Programação

1. Use as três classes *student*, *on_campus* e *off_campus* para criar um sistema de base de dados simples que mantenha a informação sobre a aplicação do estudante.

Classes Derivadas com Construtores e Destrutores

Especificar uma função construtor para uma classe geralmente é apropriado e, algumas vezes, necessário. Classes freqüentemente são muito grandes e complicadas para uma inicialização simples, e a função-membro construtor permite-lhe executar toda a codificação de partida necessária quando se cria um objeto. Por exemplo, se uma classe abre um arquivo e prepara-o para escrita, uma função construtor pode incluir um código de compactação para fazer a *garbage collection* ("coleta de lixo", ou liberação da memória usada para objetos que não são mais necessários) toda vez que for criado novo objeto. Esta é apenas uma entre muitas outras possibilidades. (O Capítulo 4 discutiu o uso das funções construtor no contexto de uma única definição de classe.)

Da mesma forma, a função-membro destrutor, se for idealizada adequadamente, pode desalojar eficientemente uma classe (é uma boa prática em programação tomar de volta os recursos que não são mais necessários). Você pode também estender as tarefas da função destrutor além da simples tarefa de desalojar, até a execução de uma variedade de atividades de encerramento, algumas apenas indiretamente relacionadas com a classe da qual ela é parte. Estes usos podem ser:

fechamento de arquivos, limpeza de buffers, ou mesmo reinicialização de blocos de memória.

Coordenação entre Classe-Base e Derivada

Tanto as funções construtor como destrutor podem também ser uma parte de uma classe derivada. No entanto, algumas pequenas diferenças na sintaxe refletem a situação mais complicada de uma classe sendo dependente de outra. Por exemplo, uma função construtor ou destrutor definida para a classe-base precisa estar coordenada com aquelas encontrada numa classe derivada. Igualmente importante é a movimentação de valores dos membros de uma classe derivada para os membros da classe-base. Particularmente, você deve considerar como a função construtor da classe-base recebe valores da classe derivada com o fim de criar um objeto completo.

O primeiro evento — coordenação — é resolvido por uma decisão astuta da definição de linguagem C++. Se uma função construtor é definida para a classe-base e para a classe derivada, C++ chama em primeiro lugar a função construtor-base; e, após ela terminar sua tarefa, C++ executa a função construtor derivada. O bom senso natural dessa política torna-a fácil de ser lembrada — você cria construtor primeiro. A ordem de execução da função destrutor é também imediata. A função destrutor da classe derivada é executada primeiro; após ela executar suas tarefas de desalojar e limpar, C++ chama a função destrutor da classe-base para finalizar a tarefa. Em outras palavras, a última classe a ser criada é a primeira a ser destruída.

Usando a Função Construtor da Classe-Base

Permanece ainda uma pergunta — como os valores criados na classe derivada chegam à função construtor da classe-base? A resposta não é tão óbvia quanto era no caso da coordenação entre funções construtor. A classe derivada deve fornecer explicitamente os valores iniciais para a classe-base. Isso é conseguido juntando-se os valores à definição da função construtor da classe

derivada. A sintaxe é simples. Por exemplo, se a classe *ex_class* for derivada de uma classe-base, o cabeçalho da definição da função construtor para essa classe derivada pode ter a seguinte aparência:

```
ex_class(int x, int y, int z) : (x,y);
```

As variáveis entre parênteses que vêm após os dois-pontos são os valores que a função construtor da classe-base precisa para criar a classe-base; eles servem para mesma finalidade: chamar a função construtor da classe-base e passar a ela estes valores. Observe que, se a função construtor da classe-base não requere parâmetros de inicialização, você não precisa chamá-la explicitamente a partir da função construtor da classe-base, porque o compilador a solicita automaticamente.

Por definição, uma função destrutor de uma classe não usa parâmetros; portanto, coordenar funções destrutor para classes-base e derivada é o menor problema. Não é necessária chamada explícita da classe derivada para a classe-base. C++ é responsável pela coordenação dessas funções-membro. Lembre-se simplesmente de que a função destrutor da classe derivada é sempre executada antes da função destrutor da classe-base.

O exemplo *student4.c* (Listagem 6.4) contém uma versão revisada da hierarquia da classe *student*. Nesta versão, a classe-base *student*, e as duas classes derivadas *on_campus* e *off_campus* têm funções destrutor. Semelhantemente, ela usa funções destrutor para desalojar recursos ligados a estas classes.

Listagem 6.4 Um exemplo de classes derivadas que usam funções construtor e destrutor: *student4.c*

```
#include <stream.h>
#include <string.h>


const fval = 20;        // especifica um tamanho para o formato de
                        // exibicao da cadeia


////////////////////////////////////////////////
// Define uma classe base
////////////////////////////////////////////////
class student {
  char* name;           // nome do estudante
  int year;             // grau atual; 1=calouro etc.
```

```
public:
  student(char*, int);     // coloca dados num registro
  ~student() { delete name; }       // define um destrutor
  char* display();                  // mostra o nome e o ano
};

student::student(char* n, int y)
{
  name = new char[strlen(n) + 1];    // aloca memoria suficiente
  strcpy(name,n);                    // copia o nome para a classe
  year = y;                          // ... e o ano
}

char* student::display()
{
  char* buffer;              // cria um buffer para exibir a cadeia
  buffer = new char[strlen(name) + fval];
  sprintf(buffer,"\nnome: %s\nano: %d\n",name,year);  // prepara-o
  return buffer;
}


////////////////////////////////////////////////
// Define uma classe derivada de student
////////////////////////////////////////////////

class on_campus : student {
  char *dorm,                // informacao adicional
       *room;                // ao assunto estudante

public:
  on_campus(char*,int,char*,char*);        // uma entrada de dados
  ~on_campus() { delete dorm; delete room; } // um destrutor
  char* display();      // declara um membro para exibicao simples
};

on_campus::on_campus(char* n, int y, char* d, char* r)
{
  dorm = new char[strlen(d) + 1];    // aloca memoria
  strcpy(dorm,d);                    // copia
  room = new char[strlen(r) + 1];    // ... e para este campo
  strcpy(room,r);
}
```

```
char* on_campus::display()
{
  char* buffer;
  int flenght = strlen(student::display()) + strlen(dorm) +
              strlen(room) + fval;
  buffer = new char[flenght];
  sprintf(buffer,"%s\ndormitorio: %s\nquarto:
%s\n",display(),dorm,room);
  return buffer;
}

////////////////////////////////////////////////
// uma outra classe derivada de student
////////////////////////////////////////////////

class off_campus : student {
  char  *street,           // nova informacao para adicionar
        *city,             // aos dados base de student
        *state,
        *zip;

public:
  off_campus(char*,int,char*,char*,char*,char*);    // uma
                                                    // entrada de dados
  ~off_campus();        // declara um destrutor
   char* display();     // declara um membro para exibicao simples
};

off_campus::off_campus(char* n,int y,char* s,char* cty,char*
st,char* z)
{
  street = new char[strlen(s) + 1];   // aloca memoria e
  strcpy(street,s);                   // copia  a nova informacao
  city = new char[strlen(cty) + 1];
  strcpy(city,cty);
  state = new char[strlen(st) + 1];
  strcpy(state,st);
  zip = new char[strlen(z) + 1];
  strcpy(zip,z);
}

inline off_campus::~off_campus()
{
  delete street;
```

```
  delete city;
  delete state;
  delete zip;
}

char* off_campus::display()
{
  char* buffer;
  int flenght = strlen(student::display()) + strlen(street) +
                strlen(city) + strlen(state) + strlen(zip) + fval;
  buffer = new char[flenght];
  sprintf(buffer,"%s\n%s\n%s,%s %s\n",display(),street,city,state,zip);
  return buffer;
}
////////////////////////////////////////////////////////////

main()
{
  cout << "\n==================================\n";

  student x("Joe Smith",1);     // preenche o registro
  cout << x.display();          // ... e mostra-o

  cout << "\n==================================\n";

  on_campus y("Hank Jones",1,"Foll Hall","L304");    // coloca os
                                                     // valores e
  cout << y.display();                               // exibe-os

  cout << "\n==================================\n";

  off_campus z("Sally Green",2,"22 Main Str","San
Francisco","CA","94113");
  cout << z.display();

}
```

A função construtor para a classe-base *student* em primeiro lugar reserva espaço para o elemento *name* (o tamanho do nome é determinado pelo valor do texto introduzido como parâmetro). Devido a essa função construtor não especificar valores assumidos, ela deve inicializar esse campo e o elemento *ano* com os parâmetros fornecidos.

As duas classes derivadas também contêm funções-membro construtor. A função *on_campus* reserva espaço para suas variáveis, *dorm* e *room*, e as inicializa. Observe como essa função construtor da classe derivada fornece os argumentos para a função construtor da classe-base *(student)*:

```
on_campus (char* n,int y,char* d,char* r) : (n,y):
```

os valores *n* (nome) e *y* (ano escolar) são tomados como parâmetro quando é inicializado o objeto *on_campus;* porém, sua verdadeira finalidade é fornecer os valores às variáveis da classe-base *student*, daí a sua aparição no segundo par de parênteses. A função construtor da classe *off_campus* também recebe o nome e o ano durante sua inicialização e passa-os para a classe *student.*

As funções destrutor para as três classes relacionadas desalojam as cadeias de caracteres que armazenam toda essa informação. Você precisa executar essa operação porque essa porção da memória foi reservada usando o operador *new* e continuará existindo até que o programa termine ou até que o operador *delete* a remova.

Classes-Base Privada e Pública

A relação assumida entre uma classe derivada e sua classe-base pode ser um tanto surpreendente. A classe-base é privada à classe derivada. Assim, os elementos públicos da classe-base são tratados como se fossem elementos privados da classe derivada: eles estão disponíveis à classe derivada e a seus membros, mas não fora das classes. Por exemplo, considere a seguinte definição de classe:

```
class base {
     int x;
public:
     base(int);
     int display( );
};
```

e a seguinte classe derivada:

```
class derived : base {
      int y;
```

```
public:
      derived(int,int);
      do_it( );
};
```

Qualquer membro da classe *derivada* pode acessar a função-membro *display( )* na parte pública da classe-*base*. No entanto, a mesma função *display( )* não está disponível a outras classes através da classe derivada — ela não é um membro público da *derivada* — mesmo que as duas classes tenham uma relação fortemente dependente. Sob esse ponto de vista, a função-membro *display( )* não é diferente da variável *y* — ambas estão ocultas na parte privada da classe. Naturalmente, você pode acessar a função *display( )* se criar um objeto do tipo *base*. Tal objeto, no entanto, não teria nenhuma ligação com qualquer objeto da classe derivada.

Classes-Base Públicas

Você pode, porém, alterar a condição anterior assumida declarando a classe-base como sendo pública. Então, a parte pública da classe-base torna-se também parte pública da classe derivada.

Você consegue essa declaração colocando a palavra *public* antes do nome da classe-base. Vamos alterar o exemplo simples da última seção para refletir esta nova situação:

```
class  derived  : public base {
       int y;
public:
       derived(int,int);
       do_it( )
} ;
```

Note a lista de funções-membro para a classe derivada inclui não somente *do_it( )*, mas também *display( )*. Na verdade, um objeto da classe derivada pode acessar *display( )*, mesmo que *display( )* não seja definida como um membro da classe derivada.

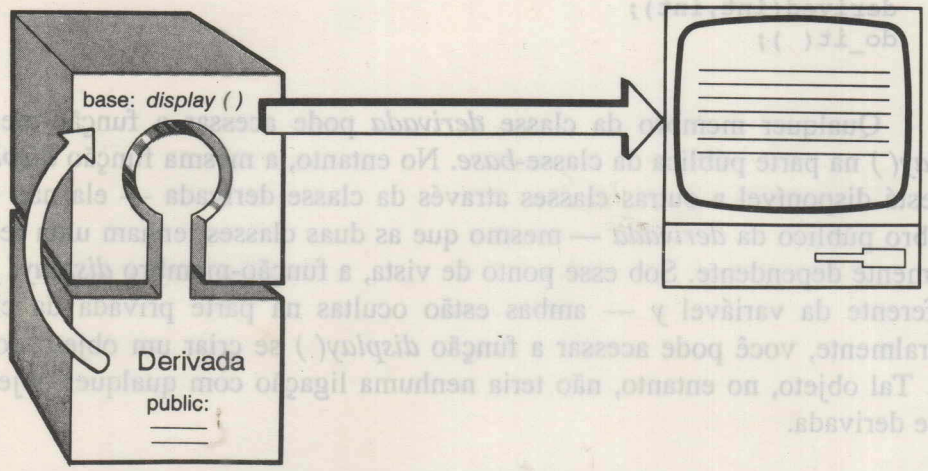


Figura 6.8 A relação entre *display*( ) na classe-base e a classe derivada.

O programa *adress.c* (Listagem 6.5) contém um exemplo prático de uma classe-base pública. A classe *basic_rec* contém um nome composto por três partes: início, meio e fim. Uma função construtor sem valores assumidos inicializa um objeto desta classe, e uma função simples destrutor desaloja seus recursos. A função-membro *mostra_nome( )* resulta numa cadeia de caracteres que contém o nome completo — início, meio e fim na ordem esperada. Um membro simplificado, *mostra_fim( )*, dá como resposta apenas o último nome (fim).

Listagem 6.5 Uma classe derivada com uma classe-base pública: *address.c*

```
#include <stream.h>
#include <string.h>

////////////////////////////////////////////////////////////////
// Define uma classe base para produzir um nome em tres partes
////////////////////////////////////////////////////////////////

class basic_rec {
  char  *first,          // armazena informacao tipica do nome
        *middle,
        *last;

public:
  basic_rec(char*,char*,char*);        // declara um construtor
```

```
  ~basic_rec();          // ... e um destrutor
  char* show_name();     // uma funcao que exibe o nome completo
  char* show_last() { return last; } // ... e uma para exibir
                                     // somente o sobrenome
};

basic_rec::basic_rec(char* f,char* m,char* l)
{
  first = new char[strlen(f) + 1];      // aloca memoria para os
                                        // membros
  middle = new char[strlen(m) + 1];
  last = new char[strlen(l) + 1];

  strcpy(first,f);       // copia os valores iniciais
  strcpy(middle,m);
  strcpy(last,l);
}

basic_rec::~basic_rec()
{
  delete first;         // desaloca as variaveis membros
  delete middle;
  delete last;
}

char* basic_rec::show_name()
{
  char* buffer;         // cria e aloca memoria para buffer
  buffer = new char[strlen(first) + strlen(middle) + strlen(last)
+ 1];
  sprintf(buffer,"%s %s %s",first,middle,last); // formata a saida
  return buffer;                    // retorna saida formatada
}

//////////////////////////////////////////////////
// define uma classe derivada de basic_rec
//////////////////////////////////////////////////

class addr : public basic_rec {
  char *street, // define os elementos para informacao do endereco
       *city,
       *state,
       *zip;

public:
```

```
  addr(char*,char*,char*,char*,char*,char*,char*);  // declara um
                                                    // construtor
  ~addr();                          // declara um destrutor
  char* show_addr();                // declara um membro de exibicao
};

addr::addr(char* n,char* m,char* l,char* s,char* cty,char*
st,char* z) : (n,m,l)
{
  street = new char[strlen(s) + 1];    // aloca os membros
  city = new char[strlen(cty) + 1];
  state = new char[strlen(st) + 1];
  zip = new char[strlen(z) + 1];

  strcpy(street,s);                 // inicializa as variaveis membros
  strcpy(city,cty);
  strcpy(state,st);
  strcpy(zip,z);
}

inline addr::~addr()
{
  delete street;          // desaloca os membros privados
  delete city;
  delete state;
  delete zip;
}

char* addr::show_addr()
{
  char *buffer;     // aloca e inicializa o buffer
  buffer = new char[strlen(street) + strlen(city) +
                    strlen(state) + strlen(zip) + 5];
  sprintf(buffer,"%s\n%s, %s %s\n",street,city,state,zip);
                                                    // formata a saida
  return buffer;                    // retorna buffer formatado
}

//////////////////////////////////////////////////////////
main()
{
  addr x("Sally","Joan","Brown","123 Main Str","San
Francisco","CA","94113");
```

```
  cout << x.show_name() << "\n"  << x.show_addr() << "\n";
  cout << "Sobrenome = " << x.show_last() << "\n";
}
```

Esse exemplo também inclui a classe derivada *addr*, que contém a seguinte informação de endereço: número e rua, cidade, estado e CEP. Esta classe usa funções-membro construtor e destrutor diretas. (Observe que a função construtor chama explicitamente a função construtor da classe-base e passa a ela os valores iniciais necessários.) O programa inclui uma função-membro adicional — *show_addr( )* — que resulta numa versão formatada da informação contida nas variáveis.

A pequena função *main( )* demonstra como as duas classes funcionam juntas. O programa cria e inicializa um objeto do tipo *addr*. Depois usa ambas as funções-membro *display* para exibir esta informação. Observe que todos os membros de *basic_rec* são acessados através da classe derivada *addr*.

Criando uma Classe-Base com uma Seção Protegida

Embora a relação entre uma classe derivada e sua classe-base seja especial, você pode acessar qualquer parte pública da classe-base com uma função-membro da classe derivada. No entanto, a seção privada da classe-base permanece fechada aos membros da classe derivada. Nesse aspecto, os membros da classe derivada não são diferentes do resto do programa e não é permitido nenhum acesso direto.

Esse "fechamento" da seção privada da classe-base pode levar a uma codificação mal construída e, em alguns casos, pode até funcionar contra o tipo de abstração de dados que a linguagem C++ deveria proporcionar. Este último evento é um problema óbvio — para acessar qualquer parte da seção privada da classe-base, você precisa incluir membros públicos na definição da classe-base. No entanto, o mesmo membro público que permite o acesso à classe derivada permite-o também ao resto do programa; não há nenhum recurso que impeça isso. De certo modo, quanto mais você necessita de abstração de dados — para executar operações delicadas "dentro" dos dados — menos proteção é oferecida. Se você

precisa manipular diretamente a estrutura de uma classe, o único mecanismo disponível aparenta abrir aquela estrutura interna ao programa todo.

No entanto, há uma solução para o problema do acesso privado contra o acesso público da classe-base. A linguagem C++ tem um mecanismo que lhe permite definir um terceiro nível de segurança para classes-base e derivada. Uma classe-base pode ter uma seção "protegida". Esta parte do programa é tratada exatamente como a seção privada da classe até o ponto em que interessa o acesso pelo resto do programa; no entanto, por sua classe-base, ela está disponível a quaisquer classes derivadas. Você define uma seção protegida colocando a palavra-chave *protected:* em uma linha. O campo de validade desta seção se estende até que seja especificado outro rótulo — por exemplo, *public:* — ou até o colchete que termina a definição da classe. Por exemplo, considere a seguinte definição de uma classe-*base* simples:

```
class base {
      int x;
protected:
      int y;
public:
      base(int);
      show_x_y( );
};
```

e a seguinte classe dela derivada:

```
class derived:   base {
      int z;
public:
      derived(int,int,int);
      show_it( );
};
```

Aqui, qualquer função-membro da classe *derivada* pode acessar diretamente sua própria variável privada *z* e a variável protegida da base *y*. No entanto, o membro *x* ainda está indisponível (é assumido como privado).

Você pode agora definir três níveis de abstração numa série de classes. O acesso à parte privada de uma classe está restrito aos membros imediatos daquela classe. A seção protegida está disponível somente aos membros de uma classe e às funções-membro em suas classes derivadas. Finalmente, a seção pública, como

sempre, está disponível ao restante do programa e define uma interface entre ela e a classe.

O programa *address2.c* (Listagem 6.6) é uma versão modificada da listagem anterior. A classe-base contém agora uma seção protegida, constituída por informações encontradas na parte privada da classe formatada numa única cadeia de caracteres. A função-membro construtor inicializa o membro protegido, *full_name*, com o restante da classe. A classe-base também contém uma função *display*, *show_name( )*, e uma função destrutor.

Listagem 6.6 Uma classe-base com uma seção protegida: *address2.c*

```
#include <stream.h>
#include <string.h>

////////////////////////////////////////////////////////////////
// Define uma classe base para produzir um nome em tres partes
////////////////////////////////////////////////////////////////

class basic_rec {
  char  *first,          // armazena informacao tipica do nome
        *middle,
        *last;

protected:
  char* full_name;    // a informacao formatada

public:
  basic_rec(char*,char*,char*);        // declara um construtor
  ~basic_rec();                        // ... e um destrutor
  char* show_name() { return full_name; } // uma funcao que
                                          // exibe o nome completo
};

basic_rec::basic_rec(char* f,char* m,char* l)
{
  first = new char[strlen(f) + 1];      // aloca memoria para
                                        // os membros
  middle = new char[strlen(m) + 1];
  last = new char[strlen(l) + 1];

  strcpy(first,f);        // copia os valores iniciais
  strcpy(middle,m);
  strcpy(last,l);
```

```
  full_name = new char[strlen(first) + strlen(middle) +
strlen(last) + 5];
  sprintf(full_name,"%s %s %s",first,middle,last);
}

basic_rec::~basic_rec()
{
  delete first;              // desaloca as variaveis membros
  delete middle;
  delete last;
  delete full_name;
}


////////////////////////////////////////////////
// define uma classe derivada de basic_rec
////////////////////////////////////////////////

class addr : public basic_rec {
  char *street, // define os elementos para informacao do endereco
       *city,
       *state,
       *zip;

public:
  addr(char*,char*,char*,char*,char*,char*,char*);  // declara um
                                                   // construtor
  ~addr();                          // declara um destrutor
  char* show_addr();                // declara um membro de exibicao
};

addr::addr(char* n,char* m,char* l,char* s,char* cty,char*
st,char* z) : (n,m,l)
{
  street = new char[strlen(s) + 1];   // aloca os membros
  city = new char[strlen(cty) + 1];
  state = new char[strlen(st) + 1];
  zip = new char[strlen(z) + 1];

  strcpy(street,s);              // inicializa as variaveis membros
  strcpy(city,cty);
  strcpy(state,st);
  strcpy(zip,z);
}
```

```
inline addr::~addr()
{
  delete street;        // desaloca os membros privados
  delete city;
  delete state;
  delete zip;
}

char* addr::show_addr()
{
  char *buffer;     // aloca e inicializa o buffer
  buffer = new char[strlen(street) + strlen(city) +
                    strlen(state) + strlen(zip) + 5];
  sprintf(buffer,"%s\n%s, %s %s\n",street,city,state,zip);
                                          // formata a saida
  return buffer;                    // retorna buffer formatado
}

////////////////////////////////////////////////////////

main()
{
  addr x("Sally","Joan","Brown","123 Main Str","San
Francisco","CA","94113");

  cout << x.show_name() << "\n"  << x.show_addr() << "\n";

}
```

A classe derivada *addr* acrescenta informação básica de endereço ao nome que já está na classe-base. A diferença aqui é que, em vez de acessar a função-membro *show_name* da parte pública da classe-base, a classe derivada lê diretamente o membro protegido, *full_name*. A função *main( )* ilustra o uso desta classe.

Questões de Revisão

1. A função construtor para uma classe derivada deve _____ chamar a função construtor para sua classe-base e fornecer seus _____.

2. Um programa chama em primeiro lugar uma função destrutor da classe _____, depois chama a função destrutor da classe _____.

3. Se uma classe é derivada de uma classe-base _____, os membros públicos da classe-base são também membros públicos da classe derivada.

4. Se uma classe-base tem uma seção _____, os membros dessa seção são públicos para a classe derivada, mas privados para o resto do programa.

Projetos de Programação

1. Usando as classes de *address.c* como modelo, idealize um sistema de lista telefônica.

2. Crie uma série de classe para implementar um pequeno sistema de registro de pedidos.

Funções Virtuais

Outra característica importante das classes derivadas é a função "membro virtual". Às vezes, quando você deriva um grupo de classes a partir de uma classe-base, certas funções-membro precisam ser diferentes em cada classe. Por exemplo, a função *display( )* que mostra os valores da classe-base pode ter de exibir valores muito diferentes se for solicitada por uma classe derivada. Talvez algumas das classes derivadas possam usar a função *display( )* do modo como foi definido na base, enquanto outras podem precisar de itens adicionais ou mesmo de uma formatação completamente nova. O mecanismo da função *virtual* permite-lhe acomodar eficazmente essas mudanças.

Você poderia dar nomes diferentes a cada função-membro display em cada classe diferente. O membro da classe-base poderia ser *display( )*, enquanto as classes derivadas correspondentes teriam um membro chamado *show( )*; embora essa solução funcione, está longe do ideal. A necessidade de criar muitos nomes

particulares para funções-membro desfaz o estilo unificado e a codificação resultante fica mais difícil de ser lida. Além disso, essa técnica de dar nomes diferentes aos membros vai contra a filosofia da linguagem C++, que é sobrecarregar nomes que executam funções semelhantes.

Uma solução melhor — usada na Listagem 6.3 — é sobrecarregar o nome da função-membro. Você poderia, por exemplo, criar a função-membro *display( )* na classe-base e na classe derivada. Depois, você poderia usar o operador (::) de referência indireta para se referir ao membro da classe-base. Essa construção usa a sobrecarga da maneira como ela foi idealizada; o mesmo nome representa duas operações diferentes, porém semelhantes. Essa solução é adequada para uma simples série de classes, mas num sistema com muitas classes independentes, os detalhes se afundam numa confusão de referência e o programa torna-se uma "macarronada".

A melhor solução é identificar aquelas funções-membro na classe-base que você precisa personalizar para serem usadas em classes derivadas, e então declará-las como funções *virtuais*. Essa declaração proporciona um mecanismo simples para redefinir os membros em qualquer classe derivada e tem uma importante vantagem sobre o procedimento simples de sobrecarga discutido acima. Quando você usa uma função *virtual*, se uma função-membro em particular não está definida numa classe derivada, então é usada a primeira definição (base). Portanto, a definição da função na classe-base constitui-se numa definição assumida que o programa usa sempre que não precisar de uma função-membro personalizada.

Para declarar uma função *virtual*, simplesmente coloque a palavra-chave *virtual* antes da declaração da função-membro. No entanto, você tem de fazer isto na linha em que faz a declaração da função na classe-base, mesmo antes de mencionar os dados. Por exemplo, a linha a seguir:

```
virtual char* display( )
```

define uma função *display virtual* no contexto de uma definição de classe-base. Observe que não se usa a palavra-chave *virtual* na definição da classe derivada; você simplesmente a declara como se fosse qualquer outra função-membro.

Você deve definir uma função *virtual* na classe-base porque o programa deve ter uma função assumida para se apoiar. Você pode definir ou não novas versões da função *virtual* em classes derivadas subseqüentes, embora possa declará-las em qualquer classe. Devido a ela ter de ser definida na classe-base, um

programa sempre pode acessar uma função-membro *virtual* através de uma classe-base, assumindo que ela é um membro público da classe. Qualquer redefinição da função *virtual* numa classe derivada particular faz o compilador substituir a definição original pela nova, sempre que a função for usada naquela classe. Assim, é usada a definição que está no objeto que acessa a função. Somente quando não existir nenhuma nova definição é que o programa se volta para a definição original.

O programa *student5.c.* (Listagem 6.7) contém uma classe-base e duas classes derivadas. A função-membro *virtual display( )* é definida tanto na classe-base quanto nas classes derivadas. A pequena função *main( )* demonstra a operação desse sistema.

Listagem 6.7 Classes derivadas que utilizam funções virtuais: *student5.c*

```
#include <stream.h>
#include <string.h>

const fval = 20;        // especifica um tamanho para o formato de
                        // exibicao da cadeia

////////////////////////////////////////////////
// Define uma classe base
////////////////////////////////////////////////

class student {
  char* name;           // nome do estudante
  int year;         // grau atual; 1=calouro etc.

public:
  student(char*, int);     // coloca dados num registro
  ~student() { delete name; }       // define um destrutor
  virtual char* display();          // mostra o nome e o ano
};

student::student(char* n, int y)
{
  name = new char[strlen(n) + 1];     // aloca memoria suficiente
  strcpy(name,n);                     // copia o nome para a classe
  year = y;                           // ... e o ano
}
```

```
char* student::display()
{
  char* buffer;              // cria um buffer para exibir a cadeia
  buffer = new char[strlen(name) + fval];
  sprintf(buffer,"\nnome: %s\nano: %d\n",name,year); // prepara-o
  return buffer;
}

////////////////////////////////////////////////
// Define uma classe derivada de student
////////////////////////////////////////////////

class on_campus : student {
  char  *dorm,                // informacao adicional
        *room;                // ao assunto estudante

public:
  on_campus(char*,int,char*,char*);        // uma entrada de dados
  ~on_campus() { delete dorm; delete room; }  // um destrutor
  char* display();      // declara um membro para exibicao simples
};

on_campus::on_campus(char* n, int y, char* d, char* r) : (n,y)
{
  dorm = new char[strlen(d) + 1];    // aloca memoria
  strcpy(dorm,d);                    // copia
  room = new char[strlen(r) + 1];    // ... e para este campo
  strcpy(room,r);
}

char* on_campus::display()
{
  char* buffer;
  int flenght = strlen(student::display()) + strlen(dorm) +
                strlen(room) + fval;
  buffer = new char[flenght];
  sprintf(buffer,"%s\ndormitorio: %s\nquarto: %s\n",display(),
                 dorm,room);
  return buffer;
}

////////////////////////////////////////////////
// uma outra classe derivada de student
////////////////////////////////////////////////
```

```
class off_campus : student {
  char  *street,                  // nova informacao para adicionar
        *city,                    // aos dados base de student
        *state,
        *zip;

public:
  off_campus(char*,int,char*,char*,char*,char*);
                                              // uma entrada de dados
  ~off_campus();          // declara um destrutor
   char* display();       // declara um membro para exibicao simples
};

off_campus::off_campus(char* n,int y,char* s,char* cty,char* st,
                       char* z) : (n,y)
{
  street = new char[strlen(s) + 1];    // aloca memoria e
  strcpy(street,s);                    // copia a nova informacao
  city = new char[strlen(cty) + 1];
  strcpy(city,cty);
  state = new char[strlen(st) + 1];
  strcpy(state,st);
  zip = new char[strlen(z) + 1];
  strcpy(zip,z);
}

inline off_campus::~off_campus()
{
  delete street;
  delete city;
  delete state;
  delete zip;
}

char* off_campus::display()
{
  char* buffer;
  int flenght = strlen(student::display()) + strlen(street) +
                strlen(city) + strlen(state) + strlen(zip) + fval;
  buffer = new char[flenght];
  sprintf(buffer,""%s\n%s\n%s,%s %s\n",display(),street,city,state,zip");
  return buffer;
}

//////////////////////////////////////////////////////////
```

```
main()
{
  cout << "\n==================================\n";

  student x("Joe Smith",1);      // preenche o registro
  cout << x.display();           // ... e mostra-o

  cout << "\n==================================\n";

  on_campus y("Hank Jones",1,"Foll Hall","L304");     // coloca os
                                                      // valores e
  cout << y.display();                               // exibe-os

  cout << "\n==================================\n";

  off_campus z("Sally Green",2,"22 Main Str","San Francisco","CA","94113");
  cout << z.display();

}
```

A classe-base *student* tem uma função inicializadora construtor e uma função destrutor. Ela contém apenas uma função-membro, *display( )*, que é uma função-membro *virtual*. Este membro formata a informação contida na parte privada para exibir na tela e armazena o resultado numa cadeia de caracteres. A função então resulta nesta cadeia de caracteres.

As duas classes derivadas, *on_campus* e *off_campus*, têm redefinições únicas da função-membro-base *display( )*. Cada uma mostra o endereço próprio de sua classe em particular. Na função *on_campus*, obtemos o nome do alojamento e o número do quarto; na função *off_campus*, obtemos rua e nº. Em ambos os casos, a mensagem é formatada e copiada numa cadeia de caracteres, que é resultante do nome da função.

Observe o uso interessante do operador de escopo de referência indireta (::). Na função-membro *display( )* da classe *on_campus*, ele possibilita que o programa contorne a redefinição permitida pela declaração dessa função como *virtual* e retome a definição original na classe-base. (Naturalmente, isso é usado apenas como parte da definição desta função-membro.) A classe *off_campus* usa uma chamada "fora do escopo" semelhante.

Sistemas Complicados de Classes

O conceito de classe derivada não é absoluto. A definição de uma classe não contém nada que possa distinguir entre a definição de uma classe-base e uma classe que não tenha classes derivadas. Sem dúvida, você pode criar classes derivadas a partir de classes existentes, mesmo aquelas às quais você tem um mínimo acesso, como, por exemplo, classes pré-compiladas. Qualquer classe pode servir de base para uma classe derivada — até mesmo outra classe derivada.

A correspondência entre base e derivada é relativa e por isso elas podem ser usadas como instrumentos de projeto para produzir programas altamente estruturados. Por exemplo, você pode criar hierarquias de classes onde a classe derivada de uma torna-se a classe-base de outra. Não há limite lógico para a extensão dessa cadeia de classes inter-relacionadas. Você pode usar essa propriedade para modularizar estruturas de classes arbitrariamente complexas.

O exemplo *customer.c* (Listagem 6.8) contém um sistema de três classes relacionadas: uma classe *customer*, que contém uma cadeia de identificação e um nome; uma classe *account* (derivada de *customer*), que contém outra cadeia de identificação e um saldo; e uma classe *cdeposit*, que contém uma data e uma taxa de juros. A classe *cdeposit* é derivada da classe *account*, e não da classe original *customer*.

Listagem 6.8 Uma hierarquia de classe derivada: *customer.c*

```
#include <stream.h>
#include <string.h>

////////////////////////////////////////////////////////////
// define uma classe base com um construtor, destrutor, e
//                          uma funcao virtual de exibicao
////////////////////////////////////////////////////////////

class customer {
  char  *id,              // identificacao do cliente
        *name;            // seu nome

public:
  customer(char*,char*);                    // declara um construtor
  ~customer() { delete id; delete name; }  // define um destrutor
```

```
  virtual char* display();      // declara um membro para exibicao
};

customer::customer(char* i,char* n)
{
  id = new char[strlen(i) + 1];      // aloca memoria para id
  strcpy(id,i);                      // copia o valor inicial
  name = new char[strlen(n) + 1];    // ... e para o nome
  strcpy(name,n);                    // copia o nome tambem
}

char* customer::display()
{
  char* buffer;            // aloca memoria para exibicao da cadeia
  buffer = new char[strlen(id) + strlen(name) + 5];
  sprintf(buffer,"%s\n%s\n",id,name); // copia os valores correntes
  return buffer;                     // devolve-os
}

////////////////////////////////////////////////////////////
// define uma classe account derivada da classe customer  //
////////////////////////////////////////////////////////////

class account : customer {
  char* acctid;            // identificacao do numero da conta
  long balance;            // saldo corrente em pennies

public:
  account(char*,char*,char*,long =0);     // declara um construtor
  ~account() { delete acctid; }           // define um destrutor
  char* display();   // nos precisamos de um membro para exibicao
};

account::account(char* cid,char* cname,char* acid,long b) :
(cid,cname)
{
  acctid = new char[strlen(acid) + 1];   // aloca e inicializa
  strcpy(acctid,acid);                   // o numero da conta
  balance = b;                     // e o saldo
}

char* account::display()
{
  char* buffer;
  int len = strlen(customer::display()) + strlen(acctid) + 20;
```

```
  buffer = new char[len];
  sprintf(buffer,"%s%s\n%d\n",customer::display(),acctid,balance);
  return buffer;
}

////////////////////////////////////////////////////////////////
// define uma outra classe derivada cuja classe base e' account
// - uma classe base que e' ela propria uma classe derivada
////////////////////////////////////////////////////////////////

class cdeposit : account {
  char* date;
  int percent;
public:
  cdeposit(char*,char*,long,char*,int); // inicializa um objeto
  ~cdeposit() { delete date; }          // define um destrutor
  char* display();                      // declara um novo membro
};

cdeposit::cdeposit(char* i,char* n,char* ia,long b,char* d,int r)
: (i,n,ia,b)
{
  date = new char[strlen(d) + 1];   // aloca memoria para a data
  strcpy(date,d);                   // ... e copia-a
  percent = r;                  // inicializa a taxa
}

char* cdeposit::display()
{
  char* buffer;
  int len=strlen(customer::display())+strlen(date)+20; // que
                                                       // grande?
  buffer = new char[len]; // aloca espaco para exibicao da cadeia
  sprintf(buffer,"%s%s\n%d\n",account::display(),date,percent);
  return buffer;
}

////////////////////////////////////////////////////////////////
// uma funcao driver simples para demonstrar as definicoes
// para a classe
////////////////////////////////////////////////////////////////

main()
{
```

```
  cdeposit x("L1234","Joe Smith","ACC1234",0,"Jan 22, 1988",5);

  cout << x.display();
}
```

A classe-base original, *customer*, contém informação que identifica um indivíduo. (Para simplificar, essa classe requer mínima informação — simplesmente um nome e uma cadeia de caracteres arbitrária que contém uma identificação.) A função construtor não contém valores assumidos, e, portanto, cada elemento tem de ser inicializado explicitamente. Essa é uma programação coerente porque o programa manipula valores únicos. É definida somente uma função-membro *display( )*; observe que ela é declarada como uma função *virtual*. A função destrutor da classe executa a desalocação básica.

A classe derivada *account* contém os valores de uma conta bancária. Novamente, para simplicidade, ela contém somente uma identificação arbitrária e um saldo da conta. A função construtor aceita como parâmetros todos os valores necessários para inicializar esta e a classe-base. A definição da função construtor chama explicitamente o inicializador da classe-base. Uma nova versão da função *display* formata e mostra a informação nesta classe. Uma função destrutor básica desaloja a cadeia de identificação da conta. Como o membro *balance* é uma simples variável *long*, não é necessária uma desalocação explícita.

Finalmente, o programa define uma terceira classe — *cdeposit* — que é do tipo *account*. Ela não se refere à classe-base original, mas, em vez disso, evolui a partir de um nível intermediário. Esta definição de classe consta da data e de uma taxa de juros específica para este tipo de conta. A data é o dia de aniversário do depósito. Essa função construtor contém os valores necessários não apenas para inicializar sua classe-base (*account*) — e, indiretamente, a classe original *costumer*. A função-membro *display( )* é redefinida para reconhecer as peculiaridades deste tipo. A Figura 6.9 resume essas relações de classe.

Questões de Revisão

1. Uma função-membro *virtual* tem de ser definida na classe _____

2. Uma função-membro _____ é definida na classe-base e pode ou não ser redefinida em classes derivadas subseqüentes.

3. A diferença entre uma classe derivada e uma classe-base é _____. Você pode usar uma classe derivada como base para outra classe.

Projetos de Programação

1. Dê uma "engordada" na hierarquia das classes derivadas em *customer* com funções-membro adicionais que implementam operações bancárias comuns. Tente identificar aquelas funções-membro que precisam ser declaradas como *virtual*.

2. Use a série de classes de operações bancárias para criar um sistema simples de administração bancária.

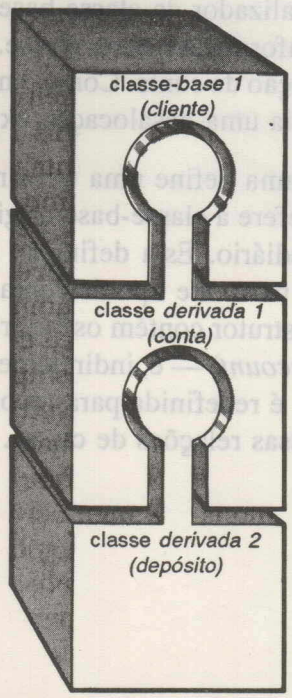


Figura 6.9 Um sistema complexo de classes.

Resumo

Este capítulo explorou a capacidade que a linguagem C++ tem para criar sistemas de classes relacionadas. O mecanismo para isso é o da derivação de uma classe a partir de outra. A classe derivada contém toda a informação de sua base — ela, ao mesmo tempo — "é" aquela classe, e é ainda mais. A utilidade da derivação de classe é ilimitada porque você pode usar qualquer classe já existente como base, mesmo que o seu acesso a ela esteja restrito a uma versão pré-compilada. A classe derivada permite que o programador divida um programa em módulos cada vez mais específicos para uma solução mais fácil e transparente.

Além da relação resultante do conceito-base derivada, foram descritos também outros aspectos das classes derivadas. Entre eles, os principais são funções-membros *virtuais* e membros protegidos. Ambos os recursos geram classes mais claras e promovem a modularidade. As funções *virtuais* são funções-membros que, uma vez definidas na classe-base, podem ser redefinidas em classes derivadas subseqüentes. Os membros de uma parte protegida de uma classe-base são privados em relação ao resto do programa, mas disponíveis aos membros de qualquer classe derivada.

Finalmente, a noção de sistemas de classes foi ampliada ilustrando-se a criação de uma hierarquia de classes relacionadas. Esta hierarquia é fundamentada na premissa de que classes derivadas e classes-base são relativas.

Makron
Books

Capítulo 7
Usando o Sistema C++ de Entrada/Saída

A biblioteca-padrão C tem muitas rotinas bem desenvolvidas de entrada e saída. A estas, a linguagem C++ acrescenta um novo e conveniente recurso para colocar e tirar valores de um programa — a biblioteca stream. Esta série de tipos de dados e funções inter-relacionadas proporciona a conveniência das instruções embutidas de I/O enquanto mantém a flexibilidade das funções definidas pelo usuário. A biblioteca C contém algumas funções I/O gerais que fazem tudo para todos os tipos de dados; a biblioteca stream tem uma série de pequenas funções, cada uma ligada a tipos de dados específicos. Além disso, as funções dessa biblioteca são mais eficientes do que aquelas mais comumente usadas em C. Essa eficiência manifesta-se também pela maior legibilidade nas linhas de saída de C++.

Este capítulo apresenta a biblioteca stream C++ e seus objetos mais comuns — os fluxos-padrão de I/O (entrada/saída) *cin*, *cout* e *cerr*. Ele mostra também como estender o modelo stream aos arquivos de disco e como manusear

as proposições resultantes daquela transformação. Finalmente, o capítulo oferece muitos exemplos concretos que mostram a biblioteca stream em ação.

Entrada e Saída Básicas em C++

Para melhor entender porque a biblioteca stream em C++ é mais conveniente do que sua correspondente em C, vamos primeiro rever como a linguagem C manipula a entrada e saída. Primeiramente, lembre-se de que C não tem instruções embutidas de entrada ou saída; as funções como *printf( )* são parte da biblioteca-padrão, mas não são parte da própria linguagem. Da mesma forma, C++ não tem recursos embutidos de I/O. A ausência de I/O embutido da ao programador maior flexibilidade para produzir a interface de usuário mais eficiente para o padrão de aplicação que tem em mãos.

O problema da solução C com entrada e saída reside na sua implementação destas funções de I/O. Há pouca consistência entre elas em termos de retorno de valores e seqüências de parâmetros. Devido a isso, os programadores tendem a se tornar dependentes das funções formatadas de I/O — *printf( ), scanf( )*, e assim por diante — especialmente quando os objetos que estão sendo manipulados são números ou outros valores não-caractere. Estas funções de I/O formatadas são convenientes e, na sua maioria, compartilham de uma interface consistente, mas elas são também grandes e desajeitadas porque têm que manipular muitos tipos de valores.

A Abordagem C++ para I/O

A linguagem C++ usa o mecanismo da classe para proporcionar soluções modulares às necessidades de manipulação de dados. Continuando com esta abordagem, a biblioteca-padrão C++ fornece três classes I/O incômodas e de uso geral de C. Estas classes contêm definições para o mesmo par de operadores — >> para entrada e << para saída — que são otimizados para tipos específicos de dados. Estes operadores são sobrecarregados, de maneira que você pode acrescentar

definições para acomodar novas situações sem renunciar à sua sintaxe simples e conveniente. Assim, você pode fácil e radicalmente modularizar o sistema de entrada/saída com muitas funções pequenas de I/O que têm uma interface consistente. Você não precisa barganhar conveniência com tamanho de código.

À tradicional biblioteca C de I/O, a linguagem C++ acrescenta a biblioteca stream-padrão, que contém uma série de funções pequenas e específicas mais eficientes do que as funções C de uso geral. A biblioteca stream-padrão contém funções específicas para executar conversões entre os três sistemas de números suportados — decimal, hexadecimal e octal — bem como funções especializadas que colocam cadeias e caracteres individuais dentro de campos especificados. Ela contém ainda uma função *form( )*, que é comparável à família *printf( )* em C.

C++ também suporta entrada de números, caracteres e cadeia de caracteres mais conveniente e mais eficientemente do que C. Embora você não possa explicitamente formatar a entrada como o faz na função *scanf( )*, a nova biblioteca contém muito da funcionalidade de *scanf( )*. Para manter os programas antigos e acomodar os programadores tradicionais, a biblioteca original está, naturalmente, disponível ao programa C++.

Os Operadores I/O << e >>

Do ponto de vista do programador, o ponto-chave da biblioteca stream é o operador de saída, que substitui a familiar chamada de função. Em vez de escrever

```
printf("%d\n",x);
```

para mostrar uma mensagem na tela, em C++ você usa a forma mais concisa:

```
int x=123;
cout << x << "\n";
```

Isso significa "dar como resultado o valor de *x* e em seguida um caractere de mudança de linha". Observe que não é preciso especificar um formato para o valor em *x* — o próprio sistema o determina. Esta forma é mais conveniente e óbvia do que a função *printf( )*, mas essa não é toda a sua vantagem. Além da sua economia de expressão, o método C++ introduz também melhoras mais significativas. Por exemplo, no trecho do programa anterior, o valor armazenado na

variável x (123) é facilmente enviado à tela. Usando a biblioteca tradicional C, você precisaria primeiro converter o valor numérico numa forma de cadeia de caracteres, porque essa é a única forma pela qual os dados podem ser mostrados num dispositivo de saída como uma tela de um terminal de vídeo. Usualmente, você invocaria a função *print( )*, que é conveniente, mas um tanto "inchada". A função *print( )*, sendo uma função de conversão de uso geral, contém todos os códigos necessários para manipular as muitas combinações possíveis de transformações do tipo valor-para-saída. Essa codificação extra é conectada em cada programa, não importando se ele precisa dela ou não. Com a biblioteca stream, C++ inclui somente a codificação necessária para uma conversão específica. Geralmente, isso torna a codificação para as operações I/O menor e mais rápida do que a codificação equivalente C.

A biblioteca stream também fornece um operador semelhante *input*. O trecho de programa a seguir ilustra sua operação:

```
int x;
cin >> x;
```

Este segmento de programa coloca o valor digitado no teclado — ou introduzido por qualquer outro método — na variável *x*. Por ser *x* declarada como uma variável inteira, o programa espera por um valor inteiro. O operador de entrada executa automaticamente a conversão. Este operador substitui a instrução de entrada formatada convencional, *scanf( )*, como fez seu colorário no sistema de saída. No entanto, isso não é uma substituição, porque o operador de entrada apenas faz as conversões de dados. A função *scanf( )* é um recurso de entrada completo e formatado, que lhe permite descrever a especificação da linha de entrada. C++ não tem nenhuma função de entrada stream comparada à função de saída *form( )*. No entanto, lembre-se de que você sempre pode usar *scanf( )* — ou qualquer outra das funções básicas de I/O — sempre que precisar delas. As vantagens do uso do operador de entrada stream são as mesmas vantagens resultantes das funções de saída — a codificação resultante é menor, mais específica e mais eficiente.

Há uma maneira fácil de lembrar qual operador é para entrada e qual é para saída: os operadores I/O de C++ são análogos aos operadores de redirecionamento do MS-DOS e do UNIX. Então, da mesma forma que o símbolo < indica que um comando recebe seu valor não do teclado, mas, sim, de um arquivo,

também o símbolo << indica que um arquivo (geralmente *cout*) recebe seu valor de uma variável ou de uma expressão. Por exemplo, o seguinte comando MS-DOS:

```
dir < file
```

executa o comando *dir* com os argumentos contidos em *file*. Semelhantemente, o seguinte comando C++:

```
cout << x;
```

envia o valor armazenado em *x* para a saída-padrão *cout*. Existe a mesma relação entre > e >>. O comando MS-DOS abaixo:

```
dir > file
```

manda o resultado do comando a um arquivo e a instrução C++ a seguir:

```
cin >> x;
```

coloca o valor da entrada-padrão na variável *x* (veja Figura 7.1).

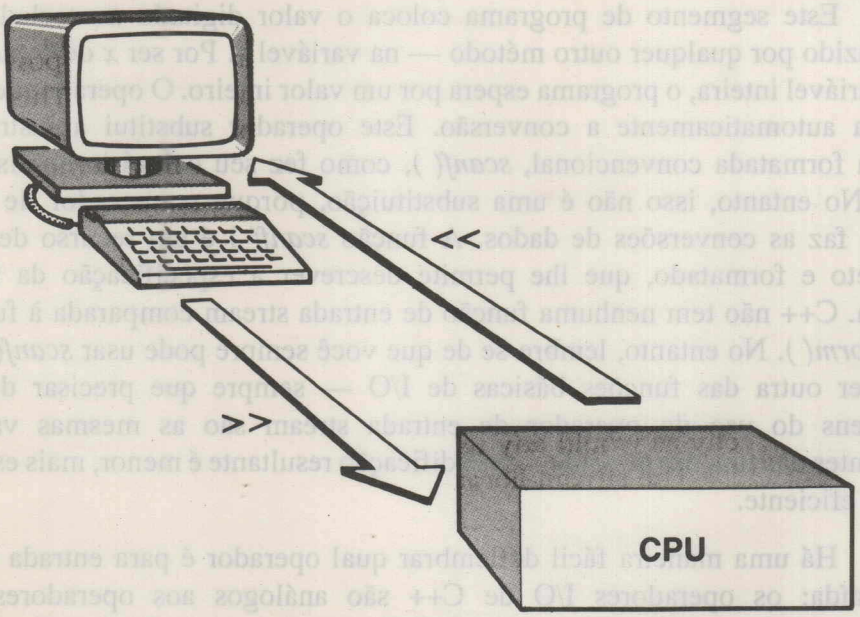


Figura 7.1 Fluxo de dados com operadores << e >>.

Como conveniência de notação, estes operadores oferecem uma economia de expressão, e sua implementação é mais eficiente do que a abordagem de uso geral de C. No entanto, o impacto total dessa nova biblioteca stream vai além dessas vantagens. Vamos agora liberar todo o poder dessas rotinas.

A Classe Stream

O Capítulo 3 mostrou como uma classe C++ integra dados e funções relacionadas numa única estrutura ou objeto. Portanto, você não deve se surpreender ao perceber que o poder e a simplicidade da biblioteca stream derivam de sua implementação como uma série de classes relacionadas. Estas classes incluem rotinas de entrada e saída definidas para cada tipo de dado-padrão em C++. Além disso tudo, usando funções *virtuais* e classes derivadas (veja Capítulo 5), você pode facilmente acrescentar suas próprias rotinas I/O para tipos de dados definidos pelo usuário. Naturalmente, estas facilidades se estendem ao I/O para disco bem como para outros dispositivos.

O Modelo Stream I/O

C++ é tão consistente quanto possível com sua linguagem-mãe, e portanto trata todas as operações de I/O como se fossem uma operação com uma cadeia de caracteres. Essa é a origem do termo *stream* — um fluxo de caracteres. Ambas as linguagens também tratam arquivos de disco como streams mesmo quando estes incluem valores binários (não texto). Lembre-se dessa metáfora útil quando for examinar algumas das partes mais sutis e potencialmente confusas da biblioteca.

C++, semelhantemente aos sistemas operacionais como o UNIX, usa o arquivo como um modelo básico para todas as entradas e saídas, mesmo aquelas que se destinam ao hardware do terminal de vídeo em vez do disco. Na verdade, C++ abre automaticamente os arquivos-padrão de entrada e saída chamados *cin* e *cout* sempre que você carrega um programa para execução. Você pode manipular

estes arquivos exatamente como faria com qualquer arquivo de dados definido pelo usuário.

A biblioteca stream é baseada nestas novas classes:

- *streambuf* — um tipo de buffer que cria e gerencia uma stream para I/O
- *istream* e *ostream* — tipos de stream de entrada e saída que definem as operações necessárias para movimentar dados entre o programa e o usuário.

A classe *streambuf* contém as funções básicas para "bufferização" de dados, tais como alocar um buffer e manter registro da posição corrente no buffer. Obserque que o tipo *streambuf* simplesmente manipula um buffer; ele não sabe de onde os valores vieram e nem para onde vão. Duas classes definidas, *istream* e *ostream*, estão associadas com o tipo *streambuf* e contêm as operações reais que implementam entrada e saída. Devido ao fato de algumas operações de entrada serem diferentes daquelas necessárias para a saída, C++ usa classes separadas. (Um tipo simples de uso geral teria sido muito grande e desajeitado e anularia a finalidade de definir classes especiais.) A diferença principal na implementação das duas classes ocorre nas definições do operador para os símbolos >> e <<.

As Streams de I/O-Padrão

Quando um programa começa a ser executado, C++ automaticamente abre três streams — *cin* (entrada-padrão), *cout* (saída-padrão) e *cerr* (erro-padrão). Estas streams tornam-se a interface entre o programa e o usuário. Quase todos os programas, até certo ponto, precisam usar estas streams.

A stream *cin* manipula toda a entrada do teclado do terminal e é do tipo *istream*. Em constraste, *cout* manipula a tela do terminal e é do tipo *ostream*. Outra stream-padrão, *cerr*, também do tipo *ostream*, proporciona ao programador um meio secundário de relatar condições de erro quando a mistura de mensagens de erro e mensagens normais poderia causar problemas. Estas três streams-padrão correspondem aos arquivos *stdin*, *stdout* e *stderr* de C.

Você pode manipular as streams-padrão como faria com suas correpondentes C:

- Você pode redirecioná-las para outros dispositvos, até mesmo um arquivo de disco
- Você pode fechá-las e reabri-las
- Você pode combiná-las entre si ou com outros arquivos

Na verdade, devido a *cerr* e *cout* estarem ligadas ao subsistema do terminal de vídeo, a maioria dos programas, na realidade, mostra as mensagens de erro através de uma stream diferente da saída ordinária. As duas streams estão separadas logicamente e poderiam ser atribuídas a dispositivos diferentes, ou mesmo a um arquivo de disco. [Sem dúvida, você pode convenientemente atribuir *cerr* a um arquivo de disco em programas destinados a rodar em processamento batch (em lotes).] Em programas que devem abrir vários arquivos de entrada e saída durante o curso do processamento, você simplesmente declara os arquivos adicionais com classes *istream* ou *ostream* e depois trata-as como se fossem streams *cin* ou *cout*.

O Operador de Entrada >> e a Stream de Entrada-Padrão

Vamos usar a stream de entrada-padrão, *cin*, para examinar a ação do operador de entrada >> em detalhe. A maior parte da discussão que segue também se aplica a outros objetos *istream*.

Note que >> é um operador sobrecarregado. C++ fornece uma definição de função separada para entrada de cada tipo de dado-padrão — inteiro, real e caractere-orientado. Você pode usar o operador >> para direcionar a entrada de qualquer valor do teclado; porque este operador é sobrecarregado, a definição correta da função para aquele tipo de dado é sempre executada. (Naturalmente, você ainda tem que ter certeza de que o usuário introduziu um valor legítimo, mas isso é um axioma da programação na vida real.) Você não precisa mais fazer conversão ou verificação explícita de tipo, e isso é um melhoramento em relação

à biblioteca-padrão C, na qual a entrada orientada não-caractere deve ser convertida ao tipo de dado necessário.

Flexibilidade do Operador >>

A conveniência do operador de entrada é óbvia — para o usuário introduzir um valor inteiro, o programa requer apenas as seguintes instruções:

```
int x;
cin >> x;
```

[Lembre-se de que o símbolo >> foi escolhido de maneira mnemônica para representar a direção do fluxo de informações da stream de entrada (*cin*) para a variável.] No entanto, esse operador pode fazer muito mais. Para maior flexibilidade, você pode usar os operadores em cascata. Por exemplo, o seguinte trecho de programa:

```
int x,y;
cin >> x >> y;
```

possibilita que o usuário introduza dois valores inteiros num programa. Um vai para a variável *x* e outro para a variável *y*. Este tipo de cascata não tem limite embutido ou prático. A codificação a seguir:

```
cin >> x >> y >> z;
```

introduz três valores num programa. O mesmo procedimento permanece válido para números reais — *float* ou *double*:

```
double x,y;
cin >> x >> y;
```

(Note que você usa sempre o mesmo operador e a mesma sintaxe, independente do tipo de dado envolvido.) Quando você usa múltiplos valores de entrada, a ordem de operação é simples e não apresenta novidades: a primeira variável mencionada é a primeira a ser preenchida, e as outras são preenchidas na ordem em que aparecem.

Devido ao operador >> estar sobrecarregado com codificação operacional explícita para cada tipo de dado que ele suporta, não há restrições quanto aos tipos

de linhas de entrada que você pode construir dentro de uma expressão *cin*. C++ suporta qualquer combinação de tipos de dados na mesma linha de código. Por exemplo, as linhas a seguir:

```
char ch;
double x;
cin >> ch  >> x;
```

geram os valores esperados nas variáveis corretas.

A Função de Entrada de Uso Geral Istream

Tão útil quanto o operador de entrada, ela não manipula todos os tipos de entradas. Para criar uma biblioteca flexível e útil, o projetista de C++ escolhe uma implementação que reflita o uso mais comum de tal rotina. Esta é uma rotina de entrada simples, sem contra-senso. O programador individual deve criar os "adornos" e o código para casos especiais. A natureza básica do operador de entrada torna-se mais evidente quando você o utiliza para manusear a entrada de cadeia de caracteres. O funcionamento dele com esse tipo de dado pode surpreender um programador descuidado.

Semelhantemente a *scanf( )* da biblioteca C, >> lê uma cadeia somente até a primeira ocorrência de um espaço em branco. Considere o seguinte trecho de programa:

```
char ch [80];
cin >> ch;
```

Se o usuário introduzir a cadeia de caracteres "isto é apenas um teste" através do teclado, somente a primeira palavra ("isto") é colocada na variável string *ch*. O operador de entrada vê o espaço em branco após uma palavra como um sinal de fim de entrada. O caractere nulo, que convencionalmente assinala o fim de uma cadeia em C, é então afixado à extremidade de *ch*. Assim, o operador >> é inadequado para entrada orientada por linha que possa conter espaços embutidos ou outros caracteres de espaço em branco.

A classe *istream* contém uma função cadeia de caractere, *get( )*, a qual serve como rotina básica de entrada que pode recuperar uma cadeia completa,

espaços em branco, e tudo o mais. A função *get( )* não faz conversões; ela simplesmente aceita uma cadeia de caracteres e coloca-a na variável especificada. Na realidade, *get( )* consiste em três funções sobrecarregadas que executam entrada básica não-interpretada. O quadro a seguir lista a forma geral dessas funções. Além de ser uma função cadeia de caractere, *get( )* suporta também entrada básica a um buffer gerenciado e a uma variável de caractere único.

Note que todas as versões de *get( )* permitem-lhe especificar um caractere que termina uma entrada. Quando você usa o terminador assumido, *\n*, a função aceita entradas (incluindo espaços) até que seja excedido o comprimento especificado da variável ou do buffer, ou até que seja enviada nova linha.

A Função *get( )* na Classe *istream*

get(char **string*, int *tamanho*, char *terminador*)

onde: string é uma variável cadeia de caracteres;
tamanho é o comprimento máximo da cadeia de entrada;
terminador indica um caractere especial que encerra a operação de entrada. Valor assumido é *'\n'*.
Esta função aceitará caracteres até que seja atingido o comprimento máximo ou seja introduzido um caractere de terminação.

get(streambuf& *buffer*, char *terminador*)

onde: buffer é um objeto streambuf;
terminador é o caractere que pára a entrada.
Aqui, também, o assumido é *'\n'*.
Esta função colocará a entrada num buffer gerenciado.

get(char& *c*)

onde: c é uma referência a uma variável caractere.
Esta função produz entrada única de caractere.

O Operador de Saída << e a Stream de Saída-Padrão

Como complemento ao objeto *istream* que importa valores para dentro de um programa, a biblioteca stream define também um tipo *ostream* que exporta valores de um programa. O ponto vital dessa classe é o operador de saída sobrecarregado << que lhe possibilita exibir tipos de dados embutidos. Esse operador pode executar a função básica de saída para a maioria dos programas.

Você pode também usar o operador de saída em cascata, e ele mostra os valores na ordem em que eles estão especificados na linha. Assim, a expressão abaixo:

```
int x=123;
cout <<"x=" << x;
```

mostrará o seguinte na tela:

```
x = 123
```

ou em qualquer outro destino de saída que você especificar. Da mesma forma como acontece com o operador de entrada, não há limite embutido no número de elementos que você pode colocar em cascata; no entanto, o bom senso e a boa prática de programação podem ditar um limite de três ou quatro. Naturalmente, você pode combinar livremente diferentes tipos de dados numa única linha. Por exemplo, o código a seguir:

```
int x=123;
double f=1.23;
cout << "x=" << x << "f=" << f;
```

criam uma mensagem com os rótulos apropriados. Observe que esta codificação é muito mais simples do que a instrução *print( )*, que precisaria de especificadores de formato.

O operador de saída tem um problema que resulta do fato de que C++ e C não têm um tipo de dado caractere distinto. Os caracteres são tratados como se fossem pequenos números inteiros, e você deve convertê-los explicitamente ou com a codificação do programa, ou usando uma das funções de biblioteca. Embora o operador de saída << não seja definido para caracteres únicos, ele *pode* aceitar

e exibir uma cadeia de caracteres. Portanto, você pode exibir uma constante caractere (constante literal) como uma cadeia de caracteres que tem apenas um caractere. No entanto, em geral, a exibição adequada de valores de caractere de variáveis e expressões deve depender de técnicas de formatação de saída ou das funções de saída mais básicas disponíveis na classe *ostream*.

As Funções de Saída Ostream

A classe *ostream* também contém duas funções de saída primitivas. A primeira delas, *put( )*, executa a operação de saída mais básica — ela manda um único caractere. A rotina *put( )* aceita uma expressão que resulta num único caractere e o envia para o dispositivo de exibição especificado. Essa função é equivalente à *putchar( )* na biblioteca-padrão C, e usa a seguinte sintaxe:

```
char ch='a';
cout.put(ch);
```

Recorde que *cout* é um objeto da classe *ostream*. A função *put( )* é um membro dessa classe, de modo que você tem que acessá-la através da notação ponto (.) ou através da notação *flecha* para variáveis de ponteiro (->).

A segunda função primitiva de saída *ostream*, *flush( )*, apaga do buffer de saída quaisquer caracteres que estejam esperando para serem enviados à saída de destino. Usualmente, você usa essa rotina com uma *ostream* atada a um arquivo de disco, a fim de garantir que todos os caracteres do buffer tenham sido explicitamente escritos num arquivo antes que você o feche ou reescreva sobre ele. No entanto, porque C++ automaticamente chama a função *flush( )* antes que ela destrua um objeto *stream*, você raramente precisa chamar explicitamente esta função. (Esse é outro exemplo das ações úteis de uma função destrutor de uma classe.)

Há muitos programas que não precisam da ajuda das funções *ostream*. No entanto, lembre-se de que elas estão disponíveis e que são particularmente úteis para enviar dados para arquivos de disco ou outros destinos não-periféricos.

Funções de Saída Formatadas

Incluídas na biblioteca stream — mas fora da classe *ostream* — há uma série de funções de saída formatada que dão ao programador maior controle sobre o mecanismo de conversão e exibição de valores. Estas funções de formato (descritas no quadro a seguir) foram criadas inicialmente para serem usadas com o operador de saída.

As Funções de Saída Formatadas

Estas funções podem ser usadas juntamente com o operador de saída para controlar a posição dos valores a serem mostrados. Elas tomam a forma geral:

<tipo><nome> (*<valor>*,*<largura>*);

char* oct(long *num*, int *larg*);	Mostra *num* como um número octal num campo de *larg* caracteres.
char* dec(long *num*, int *larg*);	Mostra *num* como um valor decimal no campo especificado.
char* hex(long *num*, int *larg*);	Mostra *num* como um valor hexadecimal na área especificada.
char* chr(long *num*, int *larg*);	Mostra *ch* como um caractere na posição especificada.
char* str(const *char*ch*, int *larg*);	Mostra uma cadeia de caracteres *ch* usando um campo de tamanho *larg*.

As funções de saída formatada dividem-se em dois grupos. O primeiro permite que o programador especifique somente a largura para exibição do campo. Cada função executa uma conversão específica de valores octal, decimal ou hexadecimal, bem como caracteres separados ou cadeias. Por exemplo, o código a seguir:

```
int=123;
cout << dec(x,4);
```

mostra o valor de *x* (123) num campo que tem largura de 4 caracteres (*dec* significa base 10, ou decimal). Se o campo especificado for pequeno demais para conter o

valor, C++ não trunca o valor, mas, sim, aumenta a largura do campo para acomodá-lo. Se você não especificar a largura do campo, C++ usa o valor assumido de 0 e o número tomará apenas o espaço necessário para ser mostrado. Uma largura de campo negativa faz com que o valor seja alinhado à esquerda no campo. A condição assumida é o alinhamento à direita. Note que C++ fornece tanto funções de um só caractere quanto funções cadeia de caractere.

O segundo grupo de recursos de saída é oferecido pela função *form( )*, que tem um recurso de formatação completo semelhante à função *printf( )* na biblioteca-padrão C. A função consiste em duas partes principais — uma cadeia de formato, com a qual você especifica a aparência da linha, o tamanho e o formato dos valores a serem exibidos, e uma lista de expressões que fornecem aqueles valores. Por exemplo, a codificação a seguir:

```
int x=123;
double f=1.23;
char ch='A';
cout << form("x=%d\nf=%f\nch=%c\n",x,f,ch);
```

apresenta cada valor e seu título apropriado em linhas separadas, como segue:

```
x = 123
f = 1.23
ch=A
```

A cadeia de formato é criada combinando três tipos de elementos:

- caracteres regulares, que são impressos como são escritos
- caracteres de formato, que especificam o formato de saída de um valor
- caracteres de controle, que servem como especificadores de controle do carro da impressora

Os caracteres de formato mais usados comumente são as seguintes combinações de dois caracteres: %d, %f, %c e %s, que representam, respectivamente, inteiro, ponteiro flutuante, caractere e cadeia de caracteres. Assim como os caracteres de formato são sempre precedidos de %, também os caracteres de controle são sempre precedidos de barra invertida ('\'). A seqüência de controle mais comum, '\n', especifica uma nova linha. A função *form( )* também permite ao programador especificar a largura de campo e, no caso de números reais e cadeias de caracteres, a precisão. Se você já estiver familiarizado com a operação

de *printf( )*, não terá dificuldade para aprender a usar *form( )*. O quadro que segue apresenta a definição dessa função e um resumo de suas opções.

Saída Formatada: A Função *form( )*

A função *form( )* pode ser usada juntamente com o operador de saída para especificar uma gama de formatos opcionais e posições de linha para exibição da saída; ela tem a forma geral:

*form(char*formato_string,<lista de expressões>);*

A forma geral de um especificador de formato é

%<especif.de larg.><modificador><cod.de formato>

Códigos de formato

Números inteiros;	u	Inteiro sem sinal
	d	Notação decimal
	o	Valores octais
	x	Número hexadecimal
Ponto flutuante ou dupla precisão	e	Notação ponto flutuante
	f	Display em ponto decimal
	g, d, e ou f	Minimiza o espaço, enquanto mantém a precisão especificada
Displays caractere-orientados:	c	Um único caractere
	s	Uma cadeia de caracteres terminada por nulo

Modificadores de Formato

	Exibe alinhando à esquerda — alinhamento à direita é condição assumida
*	Usa o próximo valor na lista de expressões como especificador de largura de campo

h	Inteiro "short"
l	Inteiro "long"
<num>	Exibição de campo com <num> caracteres
<num>.<prec> (p/ pto flut. ou dupla precisão)	Exibição de campo com <num> caracteres, <prec> dígitos após o ponto decimal
<num>.<prec> (p/ cadeia de caracteres)	Um campo de <num> caracteres mas mostrando somente os primeiros <prec> caracteres da cadeia

Usando os Objetos Stream-Padrão

Agora que você sabe como a biblioteca stream se relaciona com *cin* (entrada) e *cout* (saída), vamos examinar todos os detalhes num exemplo concreto completo. Um programa de uma calculadora, *calcio.c* (Listagem 7.1), ilustra o uso quotidiano destes objetos I/O. Esse programa aceita uma expresão aritmética que usa um único operador — por exemplo, 23+45 — e retorna o resultado da operação especificada. Ele continua até que o usuário introduza um comando *stop* explícito.

Listagem 7.1 Emulação de uma calculadora que ilustra os objetos stream: *calcio.c*

```
#include <stream.h>
#include <string.h>


main()
{
  int x=0, y=0, eval(int*,int*,char*);
  char op;

  while (eval(&x,&y,&op))
    switch (op) {
      case '+' : cout << "=" << dec(x+y) << "\n";
                 break;
      case '-' : cout << "=" << dec(x-y) << "\n";
```

```
                break;
      case '*' : cout << "=" << dec(x*y) << "\n";
                 break;
      case '/' : if (y !=0)
                   cout << "=" << dec(x/y) << "\n";
                 break;
    }
}
int eval(int *x,int *y,char *top)
{
  char buffer[80],*opr,*find_tok(char*,char*);

  cout << "->";
  cin >> buffer;      // aceita uma expressao do teclado

  if (!strcmp(buffer,"stop"))
    return 0;

  if ((opr=find_tok(buffer,"+-*/")) == 0)    // busca o operador
    return 0;

  *op = *opr;                 // salva o operador
  *opr = '\0';                // de uma cadeia faz duas
  *x = atoi(buffer);          // cria o primeiro operando
  *y = atoi(opr+1);           // ... e o segundo

  return 1;
}
char *find_tok(char* str,char* tstr)
{
  char *prt = str, *tok;

  for (; *prt!='\0'; prt++) {
    tok = tstr; // reinicializa a cadeia de simbolos pesquisados
    for (; *tok!='\0'; tok++ )
      if (*prt == *tok)          // se encontrar
        return prt;              // retorna a localizacao do operador
    }

  return 0;
}
```

O programa é formado por três funções: *main( )* controla a operação do programa e executa o cálculo real; *eval( )* aceita entrada do teclado, avalia e retorna valores úteis para *main( )*; e, finalmente, *find_tok( )* extrai o operador da cadeia de caracteres que foi introduzida.

Note que você deve incluir ambos os arquivos de cabeçalho *stream.h* e *string.h* para suportar as funções usadas no programa. C++ é menos tolerante do que C quando um programa tenta usar uma função que não foi declarada. Você deve até mesmo declarar aquelas funções que retornam um valor inteiro antes de usá-las no programa.

A função *main( )* consiste em um loop *while* que é controlado pelo valor retornado da função *eval( )*. Essa função passa valores para três parâmetros — *x*, *y* e *op*. A última variável controla a instrução *switch* no corpo do *while*. Esta instrução *switch* contém uma cláusula para uma das quatro operações suportadas. (A favor da simplicidade, o programa omite o caso assumido e a codificação para manuseio de erro.) O objeto *cout* fornece a saída. Note que o programa usa a função de formato *dec( )* para permitir-lhe maior controle sobre a tela do que seria possível com o operador <<. (Você poderia também substituir essa função com *oct( )*, ou *hex( )*, se precisar trabalhar com estes sistemas numéricos.) O caso da divisão tem um condicional adicional porque é preciso verificar a tentativa de divisão por zero.

O coração do programa é a função *eval( )*. Ela avisa o usuário para executar a entrada e depois a aceita como uma cadeia de caracteres. A vantagem de usar entrada em forma de cadeia de caracteres é a flexibilidade que ela proporciona para verificar e converter valores — a primeira instrução executada após a entrada, por exemplo, verifica a ocorrência da cadeia de caracteres "stop". Note que tudo o que você precisa são os simples operadores << e >> para solicitar e receber os valores.

Se o valor de entrada for o sinal de "stop", a função retorna zero; caso contrário, ela chama *find_tok( )*, que retorna a posição do operador na cadeia de entrada (a cadeia tem o formato de "número" "operador" "número"). Se a função *find_tok( )* não consegue encontrar um operador válido, ela retorna um zero para indicar que há problema; assim, o manuseio de erro é passado à função solicitante que trata com ele posteriormente. Se a função *find_tok( )* retorna um número positivo, o programa copia aquele caractere no parâmetro *op* e depois altera a cadeia de entrada mudando o caractere para um nulo ('0'). Esse procedimento divide a entrada em duas novas cadeias:

- uma cadeia que começa na primeira localização da cadeia de entrada e termina na posição antes do operador; e
- outra cadeia que começa na posição imediatamente seguinte ao operador e termina na última posição da cadeia de entrada original.

Cada uma dessas cadeias derivadas representa um operando de expressão. O programa então usa a função de biblioteca *atoi( )* para converter estas subcadeias em valores numéricos. Se a conversão tiver sucesso, retorna o valor 1 para a função que solicitou.

A função *find_tok( )* é similar à função *strtok( )* na biblioteca-padrão de C e C++; no entanto, sua operação é mais especializada. O programa passa à função *find_tok( )* duas cadeias de caracteres — uma cadeia a ser pesquisada e uma cadeia consistindo em caracteres importantes, ou símbolos. A função pesquisa a primeira cadeia, caractere por caractere, e tenta casar cada um desses caracteres com um caractere da segunda cadeia. Se a função encontra igualdade, retorna o endereço do caractere corrente. O retorno de um valor zero indica que os caracteres não são iguais. A variável *prt* serve como um cursor local para permitir o movimento através da cadeia.

O programa de calculadora é um exemplo simples que pode ser bastante melhorado. Em primeiro lugar, sua verificação de erro é mínima e precisa ser expandida. Uma interface de usuário mais caprichosa permitiria expressões com múltiplos operadores. O operador de entrada >> também restringe indevidamente o formato da entrada que o programador aceitaria. Especialmente, a expressão que o usuário introduz não pode conter espaços porque esse operador aceita uma cadeia de caracteres apenas até o primeiro caractere branco. Mesmo assim, o programa consegue ilustrar o bom senso dos objetos-padrão *cin* e *cout*.

Entrada e Saída em Arquivos de Disco

Uma conseqüência importante do projeto da biblioteca stream em duas partes — em um objeto *streambuf* e em uma stream — é que o mesmo modelo que executa as operações de I/O para o teclado e terminal funciona igualmente

bem quando aplicado a um arquivo de disco. Os mesmos operadores e operações se comportam exatamente da mesma maneira. Isso simplifica grandemente a tarefa de programação que tem sido sempre difícil e confusa.

Para facilitar operações de I/O a arquivos de disco, a biblioteca stream define um objeto *filebuf*, que é uma derivada do tipo padrão *streambuf*. Semelhantemente a seu tipo-mãe, *filebuf* gerencia um buffer, mas, neste caso, o buffer está ligado a um arquivo de disco. A declaração contém itens adicionais que C++ precisa para tratar com o dispositivo de armazenamento secundário e com a interface de seu sistema operacional. As outras funções de gerenciamento são herdadas do tipo *streambuf*. Entre as novas funções-chave específicas para *filebuf* estão *open( )* e *close( )*; no entanto, note também que novas funções construtor são definidas especialmente para inicializar esse tipo derivado.

Abrindo um Arquivo

A primeira função que você deve usar com arquivos de disco é *open( )*. Essa função faz a conexão entre o arquivo no disco e o arquivo que está sendo manipulado dentro do porgrama. Ao contrário de C, em que você deve mandar a variável à função, em C++ a própria variável de arquivo contém a função para abri-lo; sua forma geral é a seguinte:

```
buffer_nome.open(char*arquivo_nome, open_mode mode);
```

file_name(arquivo_nome) é uma cadeia de caracteres que contém o nome de um arquivo como aparece no sistema operacional; esse nome de diretório pode conter uma especificação de caminho bem como o nome do arquivo. O outro parâmetro do comando é o especificador *mode*, que indica um dos seguintes tipos de acesso — ler, escrever ou incluir.

Mode é do tipo *open_mode*, um tipo já mencionado que tem somente os seguintes três membros:

```
enum open_mode(input=0, output=1, append=2);
```

Cada um dos membros acima representa um modo válido de arquivo. Entrada toma os valores do arquivo e coloca no programa; saída manda os dados para o disco. Se você abre um arquivo para entrada, não pode usá-lo também para

saída. Se tentar o modo saída, apagará o arquivo existente, apagando tudo o mais que estiver lá. O modo inclusão permite que um programa coloque dados no fim de um arquivo sem apagar os dados que já estão no arquivo. Se você tenta abrir para escrever um arquivo não existente, usando o modo saída ou o modo inclusão, C++ cria aquele arquivo. O exemplo a seguir é uma instrução de abertura típica:

```
filebuf file1;
if(file1.open("file.dat", output)==0)
     exit (1);
```

A expressão *file1.open( )* retorna zero se o arquivo não puder ser aberto, e , por isso, o programa pode facilmente verificar o sucesso ou falha desta operação de arquivo. Se a operação tiver sucesso, a função retorna um ponteiro para o próprio objeto *filebuf*.

Ligando um Arquivo a uma Stream

Antes de abrir fisicamente um arquivo, você deve ligá-lo a uma stream — uma *ostream* para um arquivo aberto para fornecer dados, ou uma *istream* para um arquivo aberto para receber dados. Por exemplo, a codificação a seguir:

```
ostream file_out(file1);
```

associa o objeto *ostream file_out* a um arquivo aberto previamente. Após o programador criar este objeto stream, todas as operações e funções tornam-se disponíveis, só que, em vez da saída do arquivo se destinar à tela, ela vai para um arquivo de disco. A mesma afirmação é verdadeira para um objeto *istream*, mas, naturalmente, os valores provêm do arquivo e não do teclado.

Saída e Entrada com um Arquivo

A biblioteca stream permite expandir os benefícios das operações I/O-padrão aos arquivos. Você pode usar o operador de saída para colocar valores num arquivo, desde que você ligue o arquivo a um objeto *ostream*, da seguinte maneira:

```
int x=123;
file_out << form("este é um valor de teste=%d\n",x);
```

Semelhantemente, o trecho de programa a seguir mostra como você pode ligar um objeto *istream* a um arquivo de disco e receber dados dele:

```
filebuf f_in;
if( f_in.open("file.dat",input)==0)
      exit(1);
istream file_in(&f_in);
int x;
file_in >> x;
```

Este exemplo preenche *x* com o próximo valor de *file.dat.* Note o uso de um endereço como parâmetro para os membros *istream.*

Devido a *filebuf* e os objetos stream serem objetos tipo classe, você não precisa explicitamente fechar um arquivo. No fim de um programa, ou quando uma função termina, todos os objetos stream e *streambuf* são descartados. A classe *ostream* tem um destrutor que limpa o buffer associado. O destrutor *filebuf* fecha seu arquivo antes que a função saia. Isso é suficiente para a maioria dos casos. No entanto, você pode também chamar diretamente as funções *flush( )* e *close( )*. Você precisaria ter acesso direto a estas funções, por exemplo, se você quisesse mudar a designação atual para um arquivo de disco diferente.

Testando o Estado de uma Stream

Um atributo importante do objeto istream ou ostream é seu estado atual, ou estado corrente. C++ define quatro possíveis condições:

- *_good* indica que o último acesso teve sucesso e que não foi alcançado o fim do arquivo
- *_eof* sinaliza que foi atingido o fim do arquivo associado
- *_fail* é estabelecida quando o último acesso não teve sucesso, mas especifica que não houve perda de dados
- *_bad* indica falha catastrófica do arquivo

Estes valores todos são enumerados no tipo de dado a seguir:

enum state_value {_good=0, _eof=1, _fail=2, _bad=4};

e numa variável de estado correspondente em cada seção privada das declarações *istream* e *ostream*. Você pode acessar o estado atual da stream através de uma série de funções stream específicas. Por exemplo, se a stream estiver no fim de um arquivo, *eof( )* retorna um valor verdade. Da mesma forma, *fail( ), bad( )* e *good( )* testam quanto às suas condições particulares. Como exemplo, o trecho de programa a seguir pode ler o conteúdo do arquivo stream *in_file*:

```
for(;;){
  if(in_file.eof( ))
     break;
  in_file >> ch;
  cout << ch;
}
```

Este loop continua até chegar à condição *end-of-file* (fim-de-arquivo) e então a instrução *break* tira o programa do loop. Você pode acessar diretamente a variável de estado através de *rdstate( )*. Essa função retorna um valor que você pode comparar com os valores do tipo de dado *state_value*. (Usar a função-membro mantém a modularidade da classe e protege seu programa contra quaisquer mudanças futuras na sua definição.)

Além das funções booleanas disponíveis dentro da stream, uma função realmente permite que o programador acesse a estrutura interna da própria stream. A função *clear( )* estabelece diretamente o estado da stream. O valor assumido é *_good*, mas você pode passar qualquer outro valor legítimo como parâmetro para a função. Use *clear( )* moderadamente e com cuidado porque ela altera o valor de uma variável que normalmente é estabelecida para refletir a condição da stream. As circunstâncias nas quais você usaria a função incluem planejar uma falha recuperável:

```
if(in_file.fail( ))  {
      cout << "falha recuperavel\n";
      in_file.clear( );
}
```

ou forçar uma condição que o programa requer, como, por exemplo, um prematuro fim de arquivo:

```
while(!in_file.eof( ))  {
    if(test( ))  {
```

```
        cout << "saindo do arquivo prematuramente\n";
        in_file.clear(_eof);
    }
    .
    .
    .
```

Em ambos os exemplos, a função *clear( )* oferece uma maneira ordenada e eficiente para mudar o estado de uma stream.

Um Programa Prático de Arquivo

O programa *comp.c* (Listagem 7.2) ilustra o uso de streams de arquivo. Esse programa aceita dois nomes de arquivo e compara os arquivos caractere por caractere. Ele relata quaisquer diferenças encontradas, listando a posição e o conteúdo de cada arquivo em cada ponto em que ocorre a diferença. O programa termina quando for atingido o final do menor dos dois arquivos.

Listagem 7.2 Um programa que compara dois arquivos e relata suas diferenças: *comp.c*

```
#include <stream.h>

main(int argc,char* argv[])
{
  if (argc < 3)       // verifica se a entrada e suficiente
    cout << "erro! voce deve especificar dois arquivos para
comparar\n";

  filebuf f_out1;       // abre-o ou sai
  if (f_out1.open(argv[1],input) == 0) {
    cout << form("Desculpe! Eu nao consegui abrir %s\n",argv[1]);
    exit(1);
    }
  filebuf f_out2;       // este tambem ...
  if (f_out2.open(argv[2],input) == 0) {
    cout << form("Desculpe! Eu nao consegui abrir %s\n",argv[2]);
    exit(1);
    }

  istream file1(&f_out1), file2(&f_out2);  // conecta a stream
```

```
  cout << "comparando arquivos: " << str(argv[1],10) <<
str(argv[2],8) << '\n';
  char ch1, ch2;
  int n=0;

  for (;; n++) {
    if (file1.eof() || file2.eof())   // loop ate final de arquivo
      break;
    file1.get(ch1);
    file2.get(ch2);
    if (ch1 != ch2)
      cout << form("\nposicao: %2d\t\t%c\t%c",n,ch1,ch2);
    }
}
```

Note que o programa toma a entrada através dos argumentos da linha de comando (este procedimento é o mesmo tanto em C++ como em C). Em primeiro lugar, o programa verifica a variável contador *argc* para saber se o usuário forneceu entrada suficiente. Se a contagem não indicar que foram introduzidos dois nomes de arquivos, o programa sai com uma mensagem de erro. Caso contrário, o programa tenta abrir o primeiro arquivo criando e abrindo uma variável chamada *filebuf*. Se essa tentativa falha, o programa imprime uma mensagem de erro e sai. Esse procedimento é repetido com o segundo arquivo. Após ambos os arquivos terem sido abertos com sucesso, o programa associa-os a duas variáveis *istream*, *file1* e *file2*.

Em seguida, o programa coloca uma informação na tela e começa a ler os arquivos, caractere por caractere. Note que as variáveis temporárias *ch1* e *ch2* são declaradas antes de entrar no loop; isso está de acordo com a boa prática de programação C++. O loop *for* incrementa o contador de linhas *n* e termina quando ele alcança o fim de qualquer um dos dois arquivos. A parte importante do loop é a condição que testa cada caractere para verificar a igualdade. Se não ocorrer igualdade, o programa mostra a posição e os caracteres diferentes. Não é executada nenhuma chamada explícita à função *close( )* porque o fim da seqüência do programa gentilmente fecha todos os arquivos.

Este programa em particular foi escrito como um "filtro" UNIX. O usuário introduz nomes de arquivos através do dispositivo-padrão de entrada e o resultado volta através da saída-padrão. Você pode usar o sistema operacional para redirecionar ambos — por exemplo, o programa poderia tomar suas entradas de

um arquivo de disco e retorná-las para outro sem nenhuma mudança na codificação do programa.

As Funções de I/O-Padrão e a Biblioteca Stream

Este capítulo não deverá lhe dar a impresão de que a biblioteca stream pretende substituir a série de funções de entrada e saída cuidadosamente desenvolvidas — e depuradas! — na biblioteca C. Estas funções não estão disponíveis apenas em C++, mas, sim, podem ser misturadas livremente com funções da biblioteca stream.

O tipo de dado stream foi desenvolvido para solucionar um problema particular — I/O básico de programa. Devido à ênfase ser na conveniência e modularidade, as necessidades de introdução de dados complexos ou projetos de saídas sofisticadas podem não ser satisfeitas usando apenas esta biblioteca. No entanto, para a grande quantidade de programas que requerem apenas entrada e saída simples do teclado para a tela, os objetos stream oferecem uma ferramenta segura e conveniente para rápido desenvolvimento de software.

Usando a Biblioteca Stream com Tipos de Dados Definidos pelo Usuário

O programador pode desfrutar de todo o poder da biblioteca stream não apenas para tipos de dados embutidos, tais como inteiros e números reais, mas também para classes que são definidas dentro dos programas. A mesma conveniência e mesma notação concisa podem ser compartilhadas pelos tipos definidos pelo usuário. Você pode usar *cout* e *cin* e os operadores << e >> com estes tipos, como os usaria para saídas comuns.

A vantagem de usar um sistema consistente de I/O é clara — ele torna óbvio o algoritmo básico, particularmente a alguém que esteja lendo o programa pela primeira vez. O uso de funções de entrada/saída geralmente é um procedimento complicado. Freqüentemente, estas operações são únicas para uma implementação específica ou estão ligadas a alguma peculiaridade do hardware. Mesmo sem esse tipo de problema, os programas cheios de chamadas de funções especializadas de I/O tendem a ser enroscados e confusos. A biblioteca stream — própria da filosofia de C++ — permite ao programador ocultar estes detalhes dentro de uma interface-padrão.

Como projetista do programa, você busca somente um resultado — examinar os novos tipos de dados para dedicir qual o formato apropriado de I/O para eles. Uma vez tomada essa decisão, você apenas precisa escrever a codificação que implementa esse formato. Devido ao fato de tanto o operador de saída (<<) quanto o operador de entrada (>>) serem funções sobrecarregadas, sua codificação torna-se simplesmente outra versão dessas funções. Assim, você não precisa alterar nada na biblioteca stream. Sua nova função simplesmente acrescenta uma interface alternativa a ela.

Redefinindo Ostream

Para criar uma função de saída para uma nova classe, você precisa redefinir a seguinte função:

```
ostream& operator << (ostream&, <class>)
```

para criar a saída desejada para a nova classe. Nesta sintaxe, <class> representa uma declaração de classe. Você pode conseguir essa redefinição através de uma entre duas maneiras.

A situação mais simples ocorre quando uma classe já tem uma função de saída. O programa *dateout.c* (Listagem 7.3) ilustra esse caso com a classe de datas *julian* que foi apresentada no Capítulo 4. Neste exemplo, a nova função operador << simplesmente chama a função de saída existente.

Listagem 7.3 Uma classe *date* que usa o operador de saída <<: *dateout.c*

```
#include <stream.h>
#include <string.h>

// define alguns valores constantes uteis

const int months[]={0,31,59,90,120,151,181,212,243,273,304,334,365};

const char* mname = {"","Janeiro","Fevereiro","Marco",
                     "Abril","Maio","Junho","Julho",
                "Agosto","Setembro","Outubro",
                "Novembro","Dezembro"};

class julian {
  int days;          // numero de dias desde Janeiro 1
public:
  julian(int =0, int =0); // primeiro construtor: igual a 12,3

  char *current_date(); // exibe a data corrente no formato mes-dia
};

julian::julian(int mon, int day)
{
  days=(mon<1 || mon>12) ? 0 : (mon==1) ? day : months[mon-1]+day;
}

char* julian::current_date()
{
  int mn,dy;

  if (days <= 31) {        // verifica se o mes e Janeiro
    mn = 1;
    dy = days;          // nao precisa calcular a data juliana
    }
  else
    for (int i=2; i<=12; i++)
    if (days <= months[i]) {
      mn = i;                    // define o numero de dias
      dy = days - months[i-1];       // calcula o dia
      break;
    }
  char *buffer = new char[20];
```

```
  sprintf(buffer,"%s %d,mnames[mn],dy);    // converte para
                                           // formato texto
  return buffer;                       // retorna a data como texto
}
ostream& operator<<(ostream& op,julian x)
{
  return op << x.current_date();
}

main()
{
  julian today(1,31);
  cout << today;
}
```

Esse exemplo usa uma classe *julian* simplificada. Seu construtor inicializa uma variável declarada como sendo desse tipo. A data é armazenada como o número de dias desde 1 de janeiro (para maior simplicidade, não são levados em conta os anos bissextos). A função-membro de saída *current_date( )* retorna o valor corrente de um objeto da classe. Esse valor é colocado numa cadeia de caracteres na forma *mês-dia.*

Embora a função-membro *current_date( )* possa produzir saída para si própria, você pode também chamá-la com uma função operador *ostream.* É isso que o exemplo faz. A função seguinte:

```
ostream& operator(ostream& op, julian x)
```

redefine o operador de saída de maneira que este se torne disponível diretamente atraves de *cout.* Em vez de usar este formato:

```
julian x(1,2);
cout << x.current_date( );
```

você pode usar agora esta versão:

```
cout << x;
```

que mantém a consistência da interface criada pela biblioteca-padrão.

A função *operator<<( )* chama o membro *current_date* para recuperar uma forma convertida do valor armazenado no membro *days* da classe. O valor é recuperado como uma cadeia de caracteres, que por sua vez, é mandada de volta através da instrução de retorno que a converte num objeto do tipo *ostream*. A função *main( )* é uma simples função driver que ilustra o uso destes novos operadores.

Note que, devido à função operador não ser um membro da classe *julian*, ela não pode acessar a parte privada da classe. A única maneira de você conseguir uma saída direta é chamando uma função-membro como *current_date( )*. Esse operador redefinido não pode ser um membro da classe (porque ele já pertence à classe *ostream*), no entanto, ele pode ser uma função *amiga*. Neste caso, ele tem total acesso à parte privada da classe *julian*. E, o que é mais importante, esse tipo de acesso permite-lhe programar o operador << diretamente, de maneira que você não precisa usar uma função-membro intermediária como, por exemplo, *current_date( )*. O programa *dateout2.c* (Listagem 7.4) ilustra esse último caso.

Listagem 7.4 Uma classe *date* que usa o operador de saída como uma função *amiga*: *dateout2.c*

```
#include <stream.h>
#include <string.h>

// define alguns valores constantes uteis

const int months[]={0,31,59,90,120,151,181,212,243,273,304,334,365};
const char* mnames[] = {"","Janeiro","Fevereiro","Marco",
                        "Abril","Maio","Junho","Julho",
                        "Agosto","Setembro","Outubro",
                        "Novembro","Dezembro"};
class julian {
  int days;          // numero de dias desde Janeiro 1

public:
  julian(int =0, int =0);     // primeiro construtor: igual a 12,3

  friend ostream& operator<<(ostream&,julian);
};

julian::julian(int mon, int day)
```

```
{
  days = (mon<1 || mon>12) ? 0 : (mon == 1) ? day : months[mon-1]+day;
}

ostream& operator<<(ostream& op,julian x)
{
  int mn,dy;

  if (x.days <= 31) {       // verifica se o mes e Janeiro
    mn = 1;
    dy = x.days;          // nao precisa calcular a data juliana
    }
  else
    for (int i=2; i<=12; i++)
    if (x.days <= months[i]) {
      mn = i;                    // define o numero de dias
      dy = x.days - months[i-1];       // calcula o dia
      break;
    }
  char *buffer = new char[20];

  sprintf(buffer,"%s %d,mnames[mn],dy);      // converte para
                                             // formato texto
  return op << buffer;                  // retorna a data como texto
}

main()
{
  julian today(1,31);

  cout << today;
}
```

O novo exemplo é semelhante ao anterior (Listagem 7.3), mas, em vez de usar uma função-membro para definir o formato de saída, ele declara a função *operator<<( )* como *amiga* da classe *julian*. Essa função executa a conversão diretamente, sem a sobrecarga de outra chamada de função. Ela converte o valor em dias numa cadeia de caracteres, *buffer*, que usa o formato *mês-dia*. Esta, então, torna-se o objeto da instrução de retorno. Como antes, a instrução de retorno converte a cadeia de caracteres num objeto do tipo *ostream*. Como na Listagem 7.3, uma simples função driver ilustra o novo uso do operador.

A Entrada Direta de Tipos Definidos pelo Usuário

Exatamente como você pode redefinir os operadores de saída stream para manipular tipos de classes criados recentemente, você pode também personalizar a entrada. As técnicas são semelhantes. Por exemplo, a função seguinte:

```
istream& operator>>(istream&<class>&)
```

tem uma variável de referência para uma classe definida pelo usuário e a codificação apropriada para converter a entrada normal do teclado nos valores necessários para aquela classe. A variável de referência permite que uma função retorne um parâmetro definindo uma situação de chamada por referência. Do mesmo modo como acontece com a função *ostream*, essa função *operator* pode permanecer sozinha ou ser uma *amiga* da nova classe. o programa *datein.c* (Listagem 7.5) ilustra o último caso.

Listagem 7.5 Uma classe *date* que usa o operador de entrada >>: *datein.c*

```
#include <stream.h>
#include <string.h>

// define alguns valores constantes uteis

const int months[]={0,31,59,90,120,151,181,212,243,273,304,334,365};

const char* mnames[] = {"","Janeiro","Fevereiro","Marco",
                        "Abril","Maio","Junho","Julho",
                        "Agosto","Setembro","Outubro",
                        "Novembro","Dezembro"};

class julian {
  int days;              // numero de dias desde Janeiro 1
public:
  julian(int =0, int =0);     // primeiro construtor: igual a 12,3

  void new_date(char*);       // declara um membro para alterar a
                              // data corrente
  char *current_date();       // exibe a data corrente no formato
                              // mes-dia
};
```

```
julian::julian(int mon, int day)
{
  days = (mon<1 || mon>12) ? 0 : (mon == 1) ? day : months[mon-1]+day;
}

void julian::new_date(char* rdate)
{
  char *mon, *dy;
  mon = strtok(rdate," ");
  dy = strtok(0," ");

  for (int i=1; i<=12; i++)     // busca o mes correto
    if (!strcmp(mnames[i],mon))
      break;

  if (i > 12)     // condicao de erro
    days = 0;
  else if (i == 1)          // e Janeiro, days e dy sao o mesmo
    days = atoi(dy);
  else
    days = months[i-1]+atoi(dy);      // calcula o numero de dias
                                      // desde Janeiro 1
}

char* julian::current_date()
{
  int mn,dy;

  if (days <= 31) {          // verifica se o mes e Janeiro
    mn = 1;
    dy = days;          // nao precisa calcular a data juliana
    }
  else
    for (int i=2; i<=12; i++)
      if (days <= months[i]) {
        mn = i;                         // define o numero de dias
        dy = days - months[i-1];        // calcula o dia
        break;
      }
      char *buffer = new char[20];

      sprintf(buffer,"%s %d,mnames[mn],dy);     // converte para
                                                // formato texto
      return buffer;                    // retorna a data como texto
```

```
}

ostream& operator<<(ostream& op,julian x)
{
  return op << x.current_date();
}

istream& operator>>(istream& ip,julian& x)
{
  char *temp;
  temp = new char[20];

  ip.get(temp,20);

  x.new_date(temp);
  delete temp;
  return ip;
}

main()
{
  julian today(1,31);

  cout << today << "\n";
  cout << "entre com nova data: ";
  cin >> today;
  cout << today << "\n";
}
```

Esse exemplo requer uma função-membro que seja capaz de mudar o valor na parte da data da classe porque a função *operator>>( )* está definida fora da classe. Essa função-membro é chamada *new_date( )*. Ela aceita uma cadeia de caracteres na forma *mês-dia* e a converte na data juliana solicitada através do valor de *days*. A função *operator>>( )* recebe a entrada pelo teclado, da stream *ip* através da função *get( )*. (Lembre-se de que este membro de *istream* permite a entrada de uma cadeia de caracteres que inclui espaços em branco embutidos, e de que o operador >> somente aceita entrada para o primeiro caractere de espaço em branco.) O formato da data inclui um espaço entre o nome do mês e o dia do mês. A entrada do teclado é armazenada na variável *temp*, que o programa apaga após fazer a mudança. Note que a stream *ip* deve ser retornada para manter a consistência do operador. Se você usar os operadores em cascata, assim:

```
cin >> x >> y;
```

a instrução *return* garante que a cadeia não será quebrada.

Para analisar o texto do parâmetro, *new_date( )* em primeiro lugar separa o número correpondente ao mês usando a função *strtok( )* da biblioteca-padrão. (O valor correspondente ao mês é a primeira parte de *rdate* e é separado do resto do texto por um espaço.) O programa então usa *strcmp( )* para comparar cada elemento do vetor constante *mnames* do texto *mon*. Quando os números forem iguais, o número do índice representa o número do mês. Esse valor é então usado com o vetor inteiro, *months*, para extrair a data juliana no início do mês anterior. O programa então acrescenta esse valor ao dia do mês, e o cálculo produz a data juliana.

O programa *datein2.c* (Listagem 7.6) age em conjunto com a função-membro e define a função *operator>>( )* como *amiga* da classe. Do mesmo modo como acontecia com o operador de saída, isso poupa o trabalho de chamar a função e permite que a função acesse diretamente a parte privada da classe *julian*. Neste exemplo, os valores necessários *mon* (nome do mês) e *dy* (dia) são acessados como dois elementos usando o operador de entrada como ele está definido pelas cadeias de caracteres:

```
ip >> mon >> dy;
```

Aqui, aproveita-se a vantagem do fato de que os dados introduzidos pelo usuário incluem um espaço entre o mês e o dia. A rotina de conversão é idêntica àquela do exemplo anterior. Novamente, a variável *istream*, *ip*, deve ser retornada para manter a "ligação" entre os operadores stream.

Listagem 7.6 Uma classe *date* que usa o operador de entrada >> como função *amiga*: *datein2.c*

```
#include <stream.h>
#include <string.h>

// define alguns valores constantes uteis

const int
months[]={0,31,59,90,120,151,181,212,243,273,304,334,365};

const char* mnames[] = {"","Janeiro","Fevereiro","Marco",
                        "Abril","Maio","Junho","Julho",
                        "Agosto","Setembro","Outubro",
                        "Novembro","Dezembro"};
```

```
class julian {
  int days;          // numero de dias desde Janeiro 1
public:
  julian(int =0, int =0);     // primeiro construtor: igual a 12,3

  friend ostream& operator<<(ostream&,julian);
  friend istream& operator<<(istream&,julian&);

};

julian::julian(int mon, int day)
{
  days = (mon<1 || mon>12) ? 0 : (mon == 1) ? day :
months[mon-1]+day;
}

ostream& operator<<(ostream& op,julian x)
{
  int mn,dy;

  if (x.days <= 31) {       // verifica se o mes e Janeiro
    mn = 1;
    dy = x.days;          // nao precisa calcular a data juliana
    }
  else
    for (int i=2; i<=12; i++)
      if (x.days <= months[i]) {
          mn = i;                          // define o numero de dias
          dy = x.days - months[i-1];       // calcula o dia
          break;
      }
    char *buffer = new char[20];

    sprintf(buffer,"%s %d,mnames[mn],dy);      // converte para
                                               // formato texto
    return op << buffer;                       // retorna a data
                                               // como texto
}


istream& operator>>(istream& ip,julian& x)
{
  char *mon,*dy;
  mon = new char[20];
```

```
  dy = new char[20];

  ip >> mon >> dy;

  for (int i=1; i<=12; i++)   // busca o mes correto
    if (!strcmp(mnames[i],mon))
      break;

  if (i > 12)    // condicao de erro
    x.days = 0;
  else if (i == 1)     // e Janeiro, x.days e dy sao o mesmo
    x.days = atoi(dy);
  else
    x.days = months[i-1] + atoi(dy);    // o numero de dias desde
                                        // Janeiro 1
  return ip;
}


main()
{
  julian today(1,31);

  cout << today << "\n";

  cout << "entre com a data de hoje: ";
  cin >> today;
  cout << today << "\n";
}
```

Questões de Revisão

1. Quais são os três principais componentes que formam a biblioteca stream?
2. Quais os objetos stream que correspondem a *stdin*, *stdout* e *stderr* em C?
3. Pode o operador de entrada >> ser colocado em cascata para permitir a entrada de mais do que um item de dado?
4. Quais as duas maneiras pelas quais os valores de saída podem ser formatados antes de serem enviados ao objeto *cout*?
5. Como um arquivo de disco é aberto e preparado para entrada e saída?
6. Qual a função que permite ao programador verificar uma condição de *fim-de-arquivo*?
7. O que faz a função *clear( )*?

Projetos de Programação

1. Expanda o programa da calculadora da Listagem 7.1 de maneira que ele possa mostrar a saída em formatos hexadecimal e octal. Acrescente a possibilidade de tratar com números reais. Certifique-se de que colocou verificação de erros.
2. Modifique o programa de calculadora de modo que ele possa manipular expressões que contenham múltiplos operadores.
3. Modifique o programa de comparação de arquivo na Listagem 7.2 de modo que ele possa comparar dois, três, ou mais arquivos.
4. Usando a classe *julian* como modelo (*datein2.c*), crie uma classe para a hora do dia. Certifique-se de ter definido essa classe para operadores I/O << e >>.

Resumo

Este capítulo examinou a operação da biblioteca stream, uma biblioteca-padrão C++ de funções que podem ser usadas para gerar entrada e saída simples.

Ela foi idealizada como um adendo àquelas funções comuns a C e C++, tais como *printf( ), scanf( )*, e, o que é mais importante, rotinas de manipulação de arquivo.

No domínio do simples I/O, a biblioteca stream oferece vantagens substanciais:

- as funções são menos gerais e mais específicas e portanto produzem código de programa menor e mais eficiente
- a sintaxe é mais direta, particularmente com o uso de operadores sobrecarregados
- um modelo de interface mais controlada e consistente unifica tanto as operações de dispositivo quanto de arquivo de disco

Quase todas as funções encontradas na biblioteca tradicional C têm uma correspondente na biblioteca stream, incluindo o seguinte:

- entrada de caractere, inteiro e número real
- saída formatada
- I/O de caractere e cadeia de caractere

Os benefícios da biblioteca stream podem também ser estendidos às classes definidas pelo usuário, redefinindo as funções *ostream&operator<<( )* e *istream&operator>>( )*.

Makron
Books

Capítulo 8
Usando C++ com MS-DOS

Este capítulo examina o desenvolvimento de programas C++ no ambiente MS-DOS. Se você está acostumado com o desenvolvimento de software em sistemas operacionais "maiores", tais como UNIX, logo descobrirá que o MS-DOS não proporciona instrumentos padronizados e integrados que suportem uma metodologia de projeto consciente e estruturado. Por outro lado, se você trabalha regularmente no ambiente MS-DOS, provavelmente está familiarizado com as muitas bibliotecas C alienadas que tentam preencher esse vazio. No entanto, estes aperfeiçoamentos têm a desvantagem de que meramente suportam programas e, portanto, não proporcionam uma solução integral. A linguagem C++ direciona o problema de um design controlado no ambiente MS-DOS proporcionando ao programador uma abordagem construída na estrutura da própria linguagem.

Como você já viu, C++ tem todas as vantagens de C, incluindo eficiência e acesso em baixo nível, mais estruturas de alto nível que suportam diretamente um projeto estruturado. O tipo de dado *class* em C++ permite que o programador crie objetos que podem ser combinados em projetos maiores e também podem ser

guardados e reutilizados em projetos futuros. Os recursos hierárquicos inerentes das classes derivadas abrem um campo ainda maior para a criação de projetos coerentes. Ao mesmo tempo, baixo nível, sistema operacional e detalhes específicos de hardware podem ser encapsulados com segurança nas partes privadas dos objetos. Todo esse poder é conseguido sem prejuízo da memória ou da velocidade do programa. O tamanho de um programa C++ difere ligeiramente de seu análogo C, e a velocidade do programa é controlada de maneira semelhante.

A linguagem C++ também oferece ao programador MS-DOS a vantagem de ser uma linguagem de programação que foi idealizada para um ambiente só um pouco maior do que o ambiente MS-DOS. Ela não é um subsistema de um complexo gigante que foi recortado para se encaixar numa pequena máquina. Na verdade, a maioria das implementações de C++ roda bem mesmo no hardware dos PCs atuais.

Os capítulos anteriores exploraram as complicações de C++ de uma forma geral. Até aqui, todos os exemplos de programas rodam numa grande variedade de máquinas — incluindo aquelas que suportam o MS-DOS. Este capítulo, no entanto, examina em detalhe alguns exemplos que usam características específicas do mundo do IBM-PC. Estes exemplos ilustram como C++ pode simplificar as tarefas de programação cada vez mais complexas, necessárias para esse grupo de máquinas.

O Projeto: Criando um Objeto Porta Serial

Uma das partes mais simples, embora muito útil, de um PC é a porta serial. A América está se tornando rapidamente uma nação interligada e é cada vez mais raro encontrar um proprietário de um computador pessoal que não use um modem para se comunicar com o mundo externo. Além disso, muitos projetos envolvem comunicação direta entre computadores ou entre um computador e um dispositivo periférico. Geralmente, estas conexões têm que ser estabelecidas através da porta serial. Como programador, você provavelmente será chamado com freqüência para implementar alguma forma de comunicação serial, seja num programa dedicado a essa finalidade seja num módulo para algum outro programa.

No entanto, acessar uma porta serial é um processo geralmente complicado e pouco entendido. A necessidade de velocidade — e portanto a necessidade de acesso em baixo nível — vai contra as técnicas usuais de programação estruturada. Acrescentar interface serial a um programa geralmente conduz a uma codificação difícil de seguir e que tende a obscurecer toda a estrutura do programa. A melhor estratégia para o programador seria ocultar os detalhes do acesso serial das partes de nível mais alto do programa.

Embora existam técnicas tradicionais bem-sucedidas para criar uma biblioteca de funções de acesso de porta (existem até alguns "pacotes" comerciais), a criação de um objeto-porta como uma classe C++ dá a você toda a estrutura que as outras técnicas proporcionam, mas oferece também a flexibilidade de usar uma porta serial num programa com a mesma facilidade com que você usa um arquivo, um vetor, ou qualquer estrutura comum de dados. Agora, vamos explorar essa definição em detalhe.

Elementos Básicos da Porta Serial

Vamos começar com o básico — o conceito simples de comunicação de computador para computador. Se você já estiver familiarizado com a mecânica de transferência serial de dados, pode até querer pular as duas próximas seções.

O procedimento geral para esse tipo de comunicação é o seguinte: um caractere é codificado como um número binário (seu código ASCII), e o número é transmitido um bit de cada vez. (Veja Figura 8.1.) O computador e seu sistema operacional cuida da tradução do caractere em um conjunto de bits. O que você precisa controlar é a conexão física entre os dois sistemas — um cabo (na realidade, uma série de fios) e a conexão de software (os comandos e sinais que instruem o hardware a colocar um padrão de bits nos fios). Há muitos bons livros que cobrem estes tópicos em detalhe, incluindo: *Understanding Data Communications* e *Modern Connections Bible*, ambos publicados por Howard W. Sams & Company. No entanto, este capítulo examina somente o número mínimo de detalhes de que você precisa para desenvolver um modelo básico que funcione.

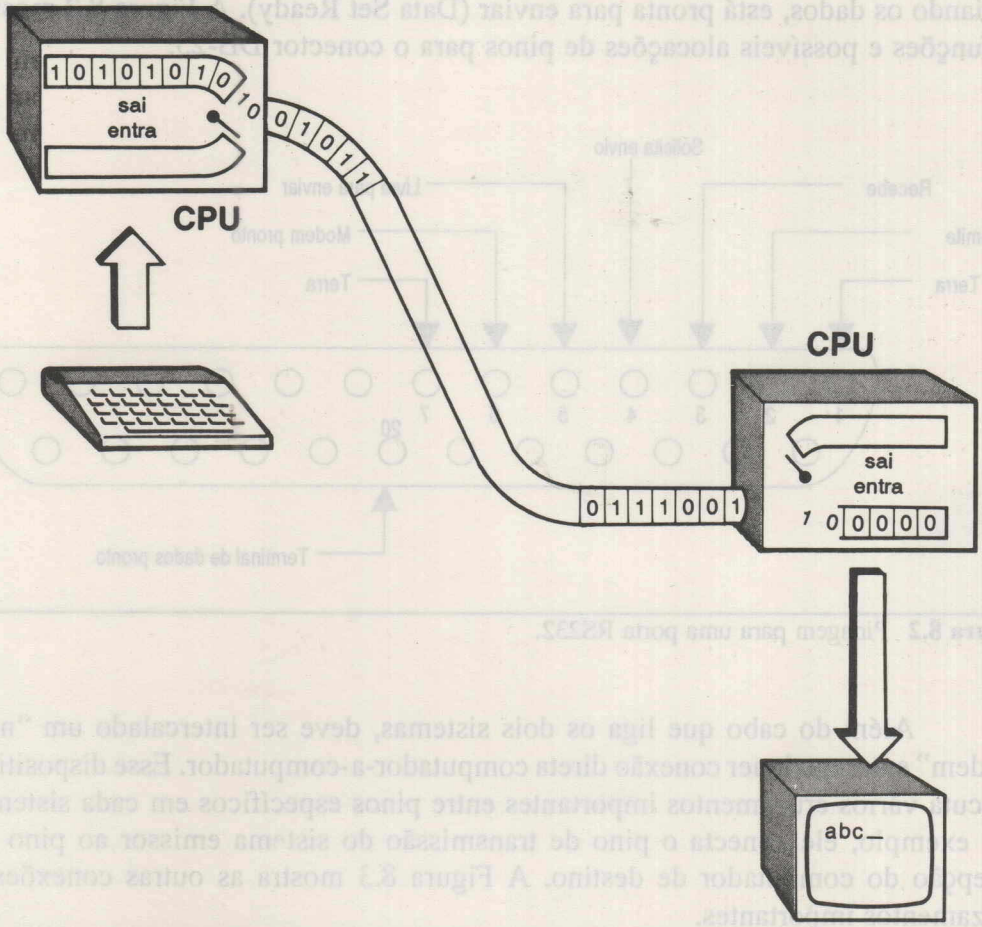


Figura 8.1 Um esboço geral da transmissão bit-a-bit.

Os detalhes físicos que você precisa conhecer são simples e óbvios. Ao conectar dois sistemas quaisquer, o tipo mais comum de plug é um conector de 25 pinos conhecido popularmente como conector DB25. Podem também ser usados conectores DB39 que têm apenas 9 pinos. A discussão que segue supõe que você esteja usando um conector DB25, mas você precisa adaptar a informação para que se enquadre dentro da sua situação local.

Cada pino do conector executa um serviço específico; um pino transmite o bit; o outro o recebe. Outro pino indica que a porta física está pronta para receber

o novo bit (Clear To Send), e outro indica que a porta, na máquina que está enviando os dados, está pronta para enviar (Data Set Ready). A Figura 8.2 mostra as funções e possíveis alocações de pinos para o conector DB-25.

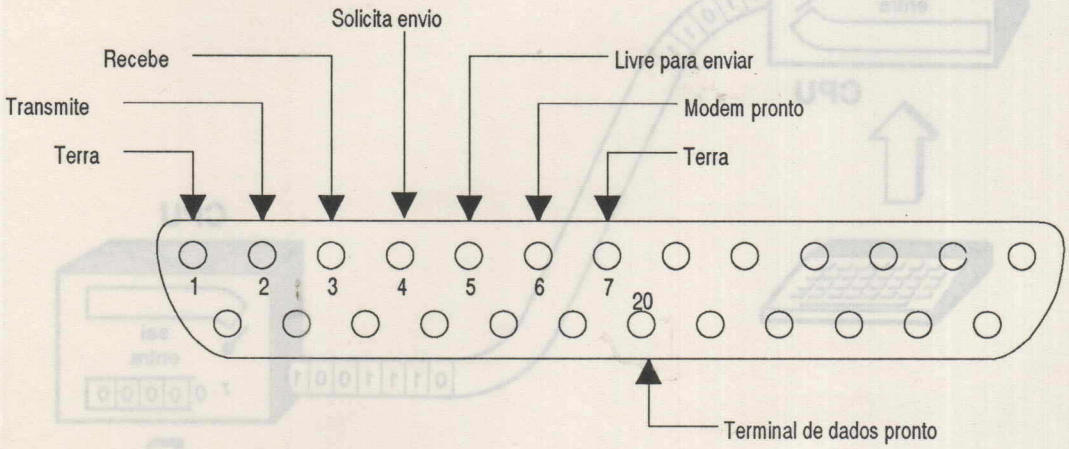


Figura 8.2 Pinagem para uma porta RS232.

Além do cabo que liga os dois sistemas, deve ser intercalado um "null modem" entre qualquer conexão direta computador-a-computador. Esse dispositivo executa vários cruzamentos importantes entre pinos específicos em cada sistema. Por exemplo, ele conecta o pino de transmissão do sistema emissor ao pino de recepção do computador de destino. A Figura 8.3 mostra as outras conexões e cruzamentos importantes.

Com exceção de algumas aplicações especializadas, o esquema de cruzamento é simétrico, de maneira que ambas as máquinas podem transmitir ou receber dados.

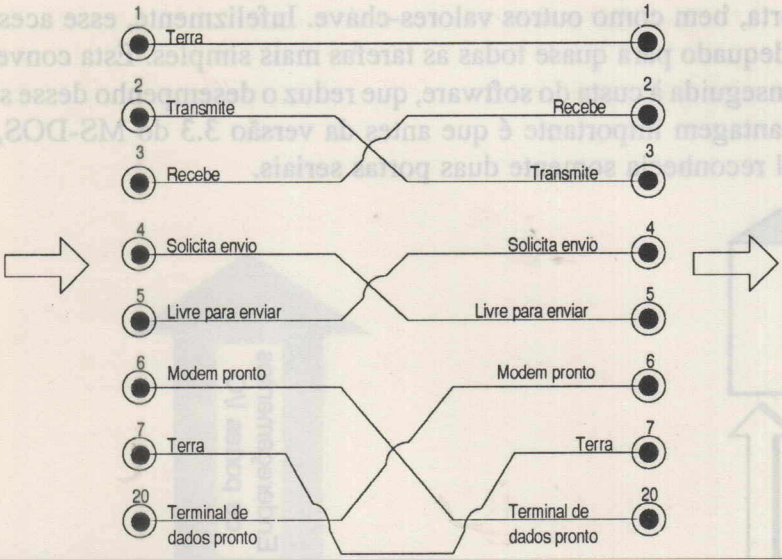


Figura 8.3 Funções do null modem.

O Software da Porta

O hardware que conecta dois sistemas de computador é, na realidade, somente o início das comunicações de dados. Após preparar o cabo e conectar fisicamente as máquinas, você deve voltar sua atenção para a porta como sendo uma entidade de software — uma espécie de dispositivo programável. A configuração e o acesso à porta é uma colaboração entre o hardware subjacente e o software do sistema operacional. Esse é o domínio no qual as vantagens de C++ se tornam aparentes. No entanto, antes de examinar estas vantagens, você precisa entender mais detalhadamente o "software da porta".

Um IBM-PC ou outro computador compatível cria, na verdade, duas portas para o usuário — *COM1*: e *COM2*: (algumas máquinas mais modernas têm mais). Você pode acessar estes dispositivos lógicos de alto nível de uma maneira relativamente direta. O sistema operacional MS-DOS configura e inicializa estas portas e permite-lhe alterar muitos de seus parâmetros. Você pode usar o comando *mode*, por exemplo, para mudar a velocidade de transferência de dados, ou "baud

rate" da porta, bem como outros valores-chave. Infelizmente, esse acesso em alto nível é inadequado para quase todas as tarefas mais simples. Esta conveniência de acesso é conseguida à custa do software, que reduz o desempenho desse subsistema. Outra desvantagem importante é que antes da versão 3.3 do MS-DOS, o sistema operacional reconhecia somente duas portas seriais.

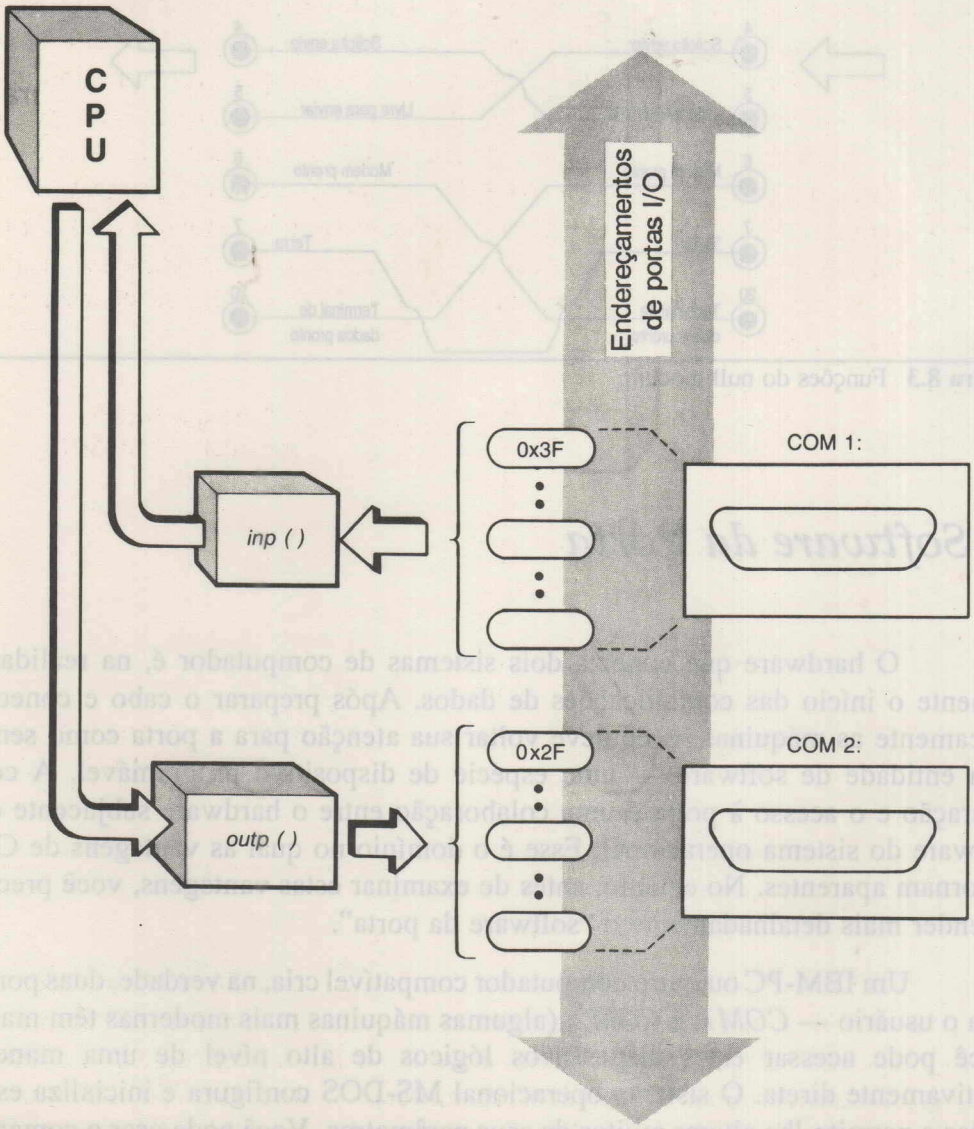


Figura 8.4 Mapeamento de endereçamentos de hardware.

Na realidade, o termo "porta" é ambíguo. As portas serial e paralela são dispositivos de alto nível. No entanto, o próprio chip da CPU usa um espaço de endereçamento especial que está associado a dispositivos externos. Esse espaço de endereçamento de porta é, de certa forma, igual ao espaço de memória direta que contém as áreas de armazenamento em ROM (Read Only Memory) — memória apenas de leitura — e RAM (Ramdom Access Memory) — memória de acesso aleatório. Cada uma dessas localizações de porta de baixo nível está associada a um endereço. A CPU proporciona acesso a estes endereços com uma instrução especial em linguagem de máquina; enviar um valor a esse tipo de porta é semelhante a colocar um valor na memória principal. Você poderia criar um módulo em Assembler que executa esse acesso e depois liga-o a um programa C++. No entanto, a maioria das implementação de C (e correspondentemente C++) já tem uma função que lhe permite acessar estas portas. Os exemplos desta seção usam as funções Microsoft C *outp( )* para escrever e *inp( )* para ler neste espaço de endereço (veja Figura 8.4).

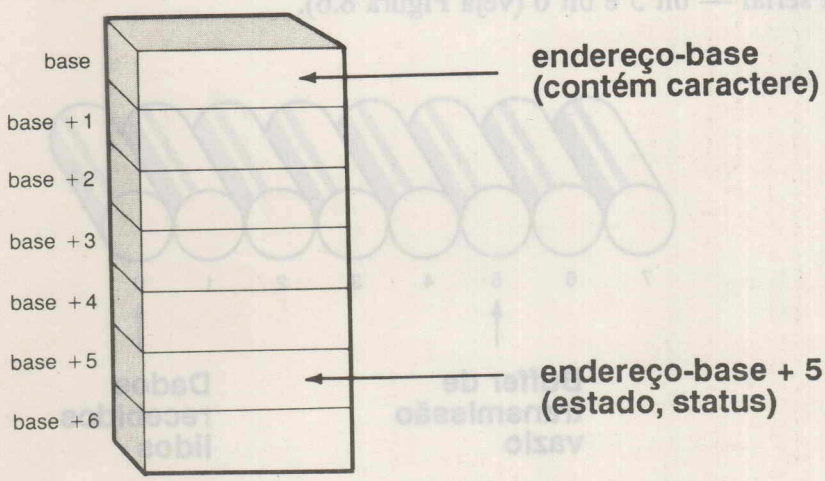


Figura 8.5 Organização de uma porta serial.

No nível abaixo dos dispositivos COM, você pode rapidamente acessar a porta de hardware permitindo ainda que o sistema execute a inicialização e a configuração. O coração da porta serial é um dispositivo conhecido como UART (Universal Asynchronous Receiver Transmitter) — Receptor Transmissor Assíncrono Universal. Do ponto de vista do programador, este dispositivo é uma série de sete localizações contíguas de um byte que são endereçadas através das portas

físicas da CPU, ou localizações absolutas no barramento (bus) do processador. A primeira localização contém o caractere que foi enviado, ou o caractere que será enviado. As demais localizações permitem-lhe acessar as várias funções de controle e flags de status do UART. Colocando valores especiais nestas localizações de controle, você pode ativar ou desativar as funções. A análise do status desses valores indica o estado da porta.

Estas localizações da porta são comumente conhecidas como *registros*. A Figura 8.5 esquematiza a organização da porta serial. Cada porta serial tem associado um endereço-*base* — *COM1:* está em 0x3F8, e *COM2:* está em 0x2F8 — que é a localização do registro que contém os dados propriamente ditos. Os registradores de controle e status são endereçados como valores equivalentes desta localização. O exemplo a seguir usa apenas dois destes — o registrador de dados no *endereço-base* e o registrador de status no *endereço-base* + 5. Na verdade, neste último byte, somente dois entre os oito bits são importantes para o programa da porta serial — bit 5 e bit 0 (veja Figura 8.6).

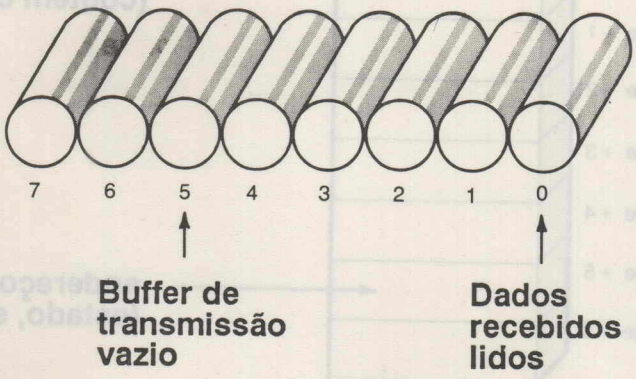


Figura 8.6 Bits importantes no registrador de status de porta.

Enviando um Byte de uma Porta a Outra

Até aqui, esta breve discussão do hardware precisava conectar dois sistemas de computador e a configuração de software da porta serial não estabeleceu o algoritmo básico para fazer qualquer espécie de comunicação. Você pode usar vários métodos para enviar e trazer valores entre dois sistemas de computador. O MS-DOS contém duas chamadas de sistema — uma destinada a mandar um caractere para uma porta serial e outra para recuperar um caractere. A vantagem em usá-las é que todos os programas que empregam essas funções são portáteis. No entanto, o nível de desempenho deste arranjo é baixo e inaceitável.

Por exemplo, essas funções reconhecem somente a primeira porta, *COM1:*, e não oferecem nenhuma maneira de controlar seus parâmetros. Uma solução melhor envolve ir diretamente às portas de hardware e criar uma função que está ligada ao sistema de interrupção da CPU. A vantagem desse método é que o programa produzido será eficiente e funcionará em qualquer velocidade de transmissão de dados. A desvantagem é que ela produz funções complicadas que dependem das rotinas de baixo nível de entrada/saída do *BIOS*. Isso pode causar problemas, porque geralmente há diferenças pequenas mas significativas entre o BIOS do IBM-PC e o "BIOS compatível" em alguns PC clones, especialmente os mais antigos.

Esse capítulo demonstra as vantagens que C++ tem sobre as linguagens de programação tradicionais quando ela tenta conquistar o hardware e facilitar o tratamento com ele. Esses programas traçam um curso intermediário entre o uso das funções básicas I/O do MS-DOS e as funções correspondentemente complexas que dependem da interrupção. Os programas acessam diretamente as portas de hardware, mas eles dependem de uma codificação C++ clara e eficiente para efetuar as transferências, e não de funções mais eficazes que dependem da interrupção. Você tem muitas fontes boas a consultar sobre a maneira de criar estes sistemas mais sofisticados. A natureza modular da linguagem C++ permitirá que você converta facilmente estes exemplos para usar com sistemas que dependem da interrupção.

Transmitindo um Caractere

Para mandar um caractere para outra porta, você precisa executar três ações distintas:

- inicializar a porta receptora de maneira que ela esteja de acordo com a porta transmissora em todos os aspectos
- esperar que a porta receptora esteja pronta para receber o novo caractere
- enviar o caractere através da porta transmissora

Esse simples processo em três etapas esconde uma grande quantidade de detalhes em baixo nível; mas, felizmente, o programador pode com segurança ignorar a maior parte deles.

A inicialização de portas é um tipo de operação que se faz "uma vez"; isto quer dizer uma vez que você estabelece uma porta para uma configuração específica, ela permanece naquela configuração até que você a mude. Numa situação típica, uma única configuração de porta é usada para muitas transmissões de caracteres. Essa inicialização requer vários acessos às portas do hardware em baixo nível que compõem uma porta serial de alto nível.

A segunda etapa no processo de transmissão é testar o estado da porta. A porta receptora avisa que está pronta para receber outro caractere, ativando o bit 5 no status register na localização base address + 5 (*endereço-base* + 5). O programa transmissor deve monitorar continuamente esta localização do hardware, de maneira que ele saiba quando deve enviar outro caractere.

O passo final é enviar o caractere propriamente dito. Este é um processo simples — você simplesmente executa uma operação de saída para o endereço da classe-base da porta. O subsistema de hardware subjacente envia o caractere bit-a-bit através da linha para a porta receptora. O sistema operacional coordena automaticamente a transmissão dos bits; o programa da porta serial precisa apenas coordenar em nível de caractere.

Vamos traduzir estes três passos em codificação C++:

```
for(;;) {
    status=inp(com + 5);
```

```
    if((status & 0x20) != 0) {
      outp(com,ch);
      break;
    }
}
```

Recorde que a função *inp( )* aceita um endereço de uma porta de hardware como parâmetro e retorna o valor que está armazenado no momento, naquela porta. Aqui, o registrador de estado está armazenado numa variável chamada *status*. (Essa função faz parte da biblioteca Microsoft C; outros compiladores podem fornecer esta função com outro nome.) Para extrair o valor do bit apropriado, o programa executa uma operação lógica *and* (e) com o byte de status e uma variável-máscara que contém zeros em todas as posições, exceto no 5º bit. (O valor hexadecimal 0x20 produz o padrão de bits apropriado — 0010.) Se o 5º bit não estiver ativado, o valor resultante será 0. No entanto, se a porta receptora estiver pronta para novo caractere, o bit de status está ativado, e o valor que resulta da operação *and* é maior do que zero.

Observe que estes exemplos não executam nenhuma inicialização explícita. Devido aos programas usarem as portas seriais-padrão, eles dependem do sistema operacional para executar esse passo necessário no processo. Lembre-se de que você não abre e fecha portas como faz com os arquivos; as portas permanecem abertas e disponíveis enquanto o computador estiver rodando.

Quando a variável *status* se torna maior do que zero, o programa pode mandar outro caractere pela porta. Isso é conseguindo usando-se a função porta de saída de hardware *outp( )*, que toma dois parâmetros — o endereço-base da porta e o caractere a ser enviado. O subsistema UART executa a transmissão bit-a-bit como fez com *inp( )*. Uma instrução *break* faz sair do loop de espera. Se o resultado da operação de mascaramento for zero, o controle passa para a instrução *for* e o processo começa novamente.

Recebendo um Caractere

Receber um caractere na porta serial — operação complementar de transmissão — requer um processo semelhante em três passos:

- inicializar a porta

- esperar até que a porta transmissora esteja pronta para enviar um novo caractere
- executar uma operação de entrada na porta

O passo de inicialização é o mesmo para a porta receptora e para a porta transmissora. Até mesmo o loop de espera é semelhante. Naturalmente, o objetivo, neste caso, é executar uma operação de entrada, e a codificação para isso é a seguinte:

```
for(;;) {
    status=inp(com + 5);
    if((status & 0x01) != 0) {
      ch = inp(com);
      return ch;
    }
}
```

Do mesmo modo como acontece na transmissão, estabelece-se um loop de espera para ler o estado (status) da porta. Desta vez, a parte importante do registrador de estado é o bit 0, que contém a condição de "data ready" (dado pronto). Assim que essa condição é estabelecida, a porta pode ler um novo caractere. Isso é conseguido chamando-se a função *inp( )* para ler o endereço-base. O valor resultante retorna, completando assim a saída do loop de espera.

Uma Classe Porta Serial

Agora que você entendeu o algoritmo básico, vamos integrar esse acesso ao software de baixo nível com as características de baixo nível de C++. Um problema da porta serial em qualquer programa de linguagem C é que ela sempre aparece como uma entidade especial, como alguma coisa que deve ser tratada diferentemente de variáveis, localizações de memória, ou mesmo arquivos de disco. Também, muitos dos seus detalhes são visíveis, ou seja, ela carrega uma "bagagem" excessiva para uma operação simples como essa de enviar um caractere de um sistema para outro. A melhor maneira de lidar com uma porta é descobrir um jeito de "empacotá-la", de modo que a maioria dos seus detalhes possa ser ignorada e

o acesso possa ser reduzido à mesma simples operação que manipula qualquer outro dado.

A solução está numa classe *port*, com a qual você pode definir variáveis do tipo *port* e usá-las em qualquer lugar no programa, onde você precisar desse tipo de acesso. E, o que é melhor ainda, você não precisa explicitamente abrir a porta, mandar os valores, e depois fechar a porta. Pode ocultar todos esses detalhes dentro da definição da porta.

Quais elementos você inclui na classe *port*? Você tem um campo vasto aqui e deverá escolher tendo em vista seu eventual uso e expansão. Vamos passar por uma primeira definição de uma classe *port*:

```
class port {
   unsigned com;
   char* name;
public:
   port(int,char*);
   ~port( );
   void pchar(int);
   int gchar( );
};
```

Você precisa incluir o endereço-base da porta, ou, pelo menos, alguma maneira de identificá-lo para o sistema; essa informação vai para *com*. É bom ter um nome definido pelo usuário, e o exemplo acrescenta uma simples cadeia de caracteres chamada *name*. Você deverá colocar ambos estes membros na parte privada da classe por várias razões, no mínimo para ocultar do resto do programa a real configuração de uma porta serial.

Assim que a classe esteja configurada, as funções-membro disponíveis para acesso são funções de uso geral, consistentes com a vasta gama de configurações da porta serial. Além de tudo, nesse nível básico, você simplesmente quer mandar um caractere através de uma porta [*pchar( )*] ou ler o próximo caractere na porta [*gchar( )*]. É difícil imaginar uma porta serial de uso geral que não permitisse essas duas operações. A codificação, naturalmente, está oculta na implementação da classe. Se você obtiver um novo hardware serial, ou, mais provavelmente, se você decidir usar uma rotina de acesso mais sofisticada, o resto do programa não precisa ser alterado. O exemplo *port.h* (Listagem 8.1) mostra uma implementação dessa classe.

Listagem 8.1 Uma classe simples *port*: *port.h*

```
// NO_EXT_KEYS e' um flag necessario ao Microsoft C
// Versao 5

#define NO_EXT_KEYS

// inclui os arquivos cabecalho necessarios
#include "\c5\include\conio.h"
#include <string.h>

const com1 = 0X3F8,     // estabelece o endereco padrao das portas
      com2 = 0X2F8;

class port {
  unsigned com;     // endereco base da porta
  char* name;       // nome da porta fornecido pelo usuario

public:
  port(int,char*);                  // inicializa a porta
  ~port() { delete name; }          // faz a "limpeza"
  char *wport() { return name; }    // qual e o nome?
  void pchar(int);          // envia um unico caractere para a porta
  int gchar();                  // traz um unico caractere da porta
};

port::port(int pnum,char* n)
{
  name = new char[strlen(n) + 1];       // estabelece o membro
                                        // nome da porta
  strcpy(name,n);                       // inicializa-o com o valor
                                        // dado pelo usuario
  com = (pnum != 1) ? com1 : com2;      // estabelece o endereco
                                        // da porta
}

void port::pchar(int ch)
{
  unsigned status;
  for (;;) {            // loop de espera
    status = inp(com+5);                // obtem o status
    if ((status & 0x20) != 0) {         // verifica-o
      outp(com,ch);                     // envia o caractere
      break;                            // sai do loop de espera
      }
```

```
    }
}
int port::gchar()
{
  int ch;
  unsigned status;
  for (;;) {                          // inicializa loop de espera
    status = inp(com+5);              // obtem o status
    if ((status & 0x01) != 0) {       // verifica se o caractere
                                      // esta pronto
      ch = inp(com);                  // pega-o da porta
      return ch;                      // envia-o a quem o solicitou
      }
    }
}
```

Esta listagem mostra a definição completa de uma classe simples *port*. Várias linhas deste arquivo precisam ser esclarecidas. O sinalizador *#define* no topo do arquivo é um valor necessário nos arquivos-cabeçalho Microsoft; sem ele, o pré-processador C++ não pode estabelecer adequadamente a compilação. Ao longo das mesmas linhas, você deverá chamar explicitamente o arquivo-cabeçalho *conio.h*. Dependendo da maneira pela qual você estabelece seus diretórios, C++ não pode reconhecer este arquivo como um arquivo-padrão. O arquivo-cabeçalho *string.h* é necessário porque você deve usar as funções de manipulação de cadeias na biblioteca-padrão. (Talvez você tenha que alterar estes detalhes para adaptar ao compilador que o seu C++ usa.)

O MS-DOS 3.1 e versões mais antigas reconhecem apenas duas portas seriais. Esta classe *port* é definida de maneira semelhante, embora você possa facilmente acrescentar qualquer número de portas desde que saiba o endereço de hardware do UART. O lado bom de *COM1*: e *COM2*: é que elas estão sempre nos mesmos endereços — 0x3F e 0x2F, respectivamente. Estes valores são atribuídos a um par de constantes que a classe usa posteriormente. Note que você deve sempre "amarrar" este tipo de informação na classe em vez de exigir que os programas que a usam conheçam as portas e os endereços. Como você verá na definição do construtor, toda função solicitante precisa fornecer 0 para *COM1*: e 1 para *COM2*:.

A função construtor para a classe, *port( )*, toma dois parâmetros — um número para indicar a porta e um nome para a porta. Uma pequena função-membro, *wport( )*, pode acessar este nome, mas não faz nada com ele neste exemplo. Esse

tipo de redundância é ainda útil, porque proporciona ao projetista outra maneira para se referir a este objeto. Um programa poderia facilmente criar mais de um objeto *port*, e referi-los todos à mesma porta física, e isso, na verdade, é uma situação típica. Usando somente o endereço da porta, você poderá ter dificuldades para distinguir estes objetos. Usando nomes, a distinção será mais direta.

A classe *port* não abre realmente a porta no sentido estrito da palavra. As portas estão disponíveis assim que o sistema operacional é carregado, e permanecem abertas até que você desligue o computador. Esta classe simplesmente lhe proporciona o acesso a estes recursos. Como resultado, o passo da inicialização, que é parte da tarefa da função construtor, faz pouco mais do que estabelecer adequadamente o endereço da porta no membro privado *com*. Ela realiza isso usando um operador de condição para testar o número da porta (*pnum*) fornecido como parâmetro. Observe que, se o usuário introduzir um número de porta ilegítimo, o programa usa *COM1:* como valor assumido. A função construtor também aloca e copia em *name* o valor *n* fornecido pelo parâmetro. A função destrutor também tem pouco a fazer; ela simplesmente devolve ao banco de memória disponível as localizações usadas por *name*.

Dentro da classe *port* estão duas funções I/O (entrada/saída) — *pchar( )* envia um caractere através da porta especificada por *com*; *gchar( )* pega o caractere que está esperando nesta porta. O código que implementa estas funções-membro usa os algoritmos discutidos anteriormente. Um loop de espera testa continuamente o estado da porta, e as funções *inp( )* e *outp( )* acessam diretamente as portas de hardware.

Uma Classe Port Ampliada

Agora que você solucionou o problema básico de movimentar um único caractere entre dois sistemas, pode trabalhar em cima deste arcabouço para criar um tipo de dado que transfere uma linha inteira de texto. Essa é uma consideração de projeto importante porque, na vida real, textos são geralmente manuseados linha por linha, em vez de caractere por caractere. O exemplo *port2.h* (Listagem 8.2) mostra a declaração de uma nova classe *port*.

Listagem 8.2 Uma declaração de uma classe *port* ampliada: *port2.h*

```
class port {
  unsigned com;              // endereco base da porta
  char* name;                // nome da porta fornecido pelo usuario
  void pchar(int);           // envia um unico caractere para a porta
  int gchar();               // traz um unico caractere da porta

public:
  port(int,char*);                    // inicializa a porta
  ~port() { delete name; }            // faz a "limpeza"
  char *wport() { return name; }      // qual e o nome?
  void send(char*);                   // envia uma linha para a porta
  char* receive();                    // traz uma linha da porta
};
```

Há duas diferenças entre esta nova declaração e a original. A diferença mais marcante é a adição de duas novas funções-membro, *send( )* e *receive( )*. No entanto, essa não é a única alteração. As funções *pchar( )* e *gchar( )* mudaram da parte pública para a parte privada da classe, de maneira que estão disponíveis apenas às funções-membro.

Faz sentido isolar as duas funções de caractere básicas. Estes membros são muito sensíveis ao hardware subjacente — se este se altera, elas têm que se alterar também. A parte pública de uma classe deve servir apenas como interface para o resto do programa. Retirando dela essas rotinas de acesso de baixo nível, você pode estar seguro de que nenhuma outra parte do programa virá a depender das idiossincrasias desta implementação particular.

O código para a nova função-membro *send( )* está na Listagem 8.3. Essa função simples é construída sobre chamadas repetidas ao membro agora privado *pchar( )*. A cadeia de caracteres a ser enviada ao sistema remoto passa para *send( )* como um parâmetro. Um loop *for* move-se através da cadeia, caractere por caractere, passando cada um deles para a porta, um de cada vez. Devido a ela não retornar nenhum valor, a função *send( )* é declarada *void*.

Listagem 8.3 A nova função-membro *send( )*

```
void port::send(char* buffer)
{
  for (int i=0; i<=strlen(buffer); i++)
    pchar(buffer[i]);
}
```

O complemento da função *send( )* é *receive( )*, cuja codificação está na Listagem 8.4. Esta função faz o inverso de *send( )*. Ela constrói uma cadeia acrescentando cada caractere a um buffer logo que este chega pela linha de comunicação.

Listagem 8.4 A função-membro *receive( )*

```
char* port::receive()
{
  char buffer[80];

  for (int i=0; i<80; i++)                    // loop para varredura
                                              // do buffer
    if ((buffer[i] = gchar()) == '\0')        // preenche caractere
                                              // a caractere
      break;
    return buffer;                            // devolve-o
}
```

A função *receive( )* não usa parâmetros. Ela declara um buffer temporário que é preenchido no decorrer de um loop *for* por chamadas repetidas à função-membro *gchar( )*. Um problema que você tem para resolver é saber qual deve ser o tamanho desse buffer. 80 caracteres parece ser o tamanho mais razoável para esse buffer, porque essa porta foi criada para um objeto orientado por linha. Na verdade, o loop continua até que seja recebido o sinal *end-of-string* (fim-de-cadeia) '\0'. Neste ponto, o novo valor do buffer é retornado à função solicitante. (A favor da simplicidade, estas funções omitem o manuseio de erros.)

Para usar esta nova classe, você precisa de alguns programas de teste. O programa *sendstr.c* (Listagem 8.5) envia pela porta uma cadeia de caracteres específica. Você precisa especificar a cadeia na linha de comando e o programa testa-a antes de continuar. Se a cadeia tiver brancos embutidos, você deve incluí-los dentro de aspas duplas (") pois, caso contrário, o programa os interpreta como mais do que um argumento de linha de comando. Note que você deve ajustar a declaração da porta à sua configuração local. Este exemplo usa *COM1:*

Listagem 8.5 Um programa de emissão, baseado na classe *port* ampliada: *sendstr.c*

```
#include <stream.h>

main(int argc,char* argv[])
{
  port x(0,"com1:");          // declara uma variavel port
```

```
  if (argc == 1) {          // apenas mensagem de linha de comando
    cout << "Por favor, inclua mensagem\n");
    exit(1);
    }
  x.send(argv[1]);          // envia-a para a porta
}
```

O programa complementar de recepção, *recvstr.c*, está na Listagem 8.6. O computador receptor é preparado para receber os dados em *COM2:*. Esse programa em primeiro lugar declara um buffer e uma porta. Uma rápida chamada ao membro *receive( )* de *port*, dentro de uma função *strcpy( )*, transfere o valor da linha para o *buffer*. Você exibe o valor, nesta variável tipo cadeia de caracteres, na forma usual.

Listagem 8.6 Um programa de recepção baseado na classe *port* ampliada: *recvstr.c*

```
#include <stream.h>

main()
{
  char buffer[80];                  // cria um buffer de mensagem
  port x(1,"com2:");                // declara uma porta
  strcpy(buffer,x.receive());       // preenche o buffer
  cout << form("%s\n",buffer);      // exibe-o na tela
}
```

Projetos de Programação

1. Expanda a definição de classe da Listagem 8.2 para incluir inicialização explícita no construtor. Modifique o destrutor para voltar a porta à sua configuração normal.

2. Reimplemente os membros *port send( )* e *receive( )* como operadores sobrecarregados.

Uma Hierarquia de Classes Port

A porta serial de um IBM-PC é obviamente um dispositivo útil. Você pode usá-la para comunicar-se com um mainframe ou outro PC — diretamente,

ou através de um modem — ou mesmo comunicar-se com dispositivos periféricos, como impressoras ou mouses. Estas aplicações terão alguns elementos em comum, e alguns elementos que são únicos para situações específicas. Uma impressora pode precisar de seqüências especiais de *setup* (configuração); um modem pode precisar de um caractere especial para se comunicar com o próprio dispositivo. O exemplo de classe *port* já evoluiu desde a transferência de um simples caractere até o envio e recepção de uma cadeia de caracteres. Como satisfazer à necessidade de desenvolver novas características sobre uma base comum de funcionalidade?

Sempre que você se deparar com uma situação em que um núcleo de funções comuns pode ser usado para várias aplicações especializadas, a classe C++ oferece uma solução eficiente e estruturada. Em vez de criar uma família de objetos díspares, cada um desligado dos outros, mesmo que compartilhando o mesmo código, você deverá isolar as operações comuns e colocá-las numa classe-base. Após criar esta base, você pode derivar uma série inteira de classes especializadas a partir dela. Cada uma conterá operações especiais para uma tarefa particular sem a desnecessária duplicação da configuração básica.

A Classe-Base

A classe-base para uma porta serial é semelhante à classe *port* original. No entanto, é necessário fazer várias modificações para que ela se torne mais geral. Para a declaração revisada, veja *port3.h* (Listagem 8.7).

Listagem 8.7 Uma *port* modificada para servir como uma classe-base: *port3.h*

```
class port {
  unsigned com;              // endereco base da porta
public:
  port(int) { com = (pnum != 1) ? com1 : com2; }    // estabelece
                                                    // o endereco
  void pchar(int);           // envia um unico caractere para a porta
  int gchar();               // traz um unico caractere da porta
};
```

Os membros *send( )* e *receive( )* foram removidos — serão membros das classe derivada. Saiu também o membro *name*. A nova *port* é idealizada como um dispositivo serial mais genérico, algo de muito básico, sobre o qual você possa

construir dispositivos únicos e especializados. Devido a não haver mais o membro *name*, deve ser removida também *wport( )*. O construtor é modificado adequadamente, de modo que seu único parâmetro seja o número da porta. A classe-base não tem uma função destrutor.

Essa classe-base despojada inclui agora somente *com*, o endereço do hardware que é subjacente à porta, uma função construtor, que define este endereço em *com*, e as funções básicas de caractere, *pchar( )* e *gchar( )*. Essa seleção salienta o objetivo final do projeto que é encontrar um mínimo de funções comuns a qualquer porta.

As funções *pchar( )* e *gchar( )* têm a mesma implementação que tinham na definição original. O membro construtor, entretanto, é mais simples porque precisa apenas definir o endereço da porta. Na verdade, ele é agora suficientemente simples para que possa ser definido como uma função *inline*.

A Classe Derivada Serial

A classe-base é uma antiga e mais "primitiva" definição de *port* que conserva somente a propriedade mais elementar de enviar e receber através da porta. Agora, vamos configurar uma porta mais realística, que pode transmitir mensagens inteiras entre dois sistemas de computador.

Em primeiro lugar, você precisa criar uma nova classe que trata a porta serial como uma cadeia de caracteres. A declaração para esta nova classe, *serial*, está em *serial.h* (Listagem 8.8).

Listagem 8.8 Uma declaração de uma classe derivada *serial*: *serial.h*

```
class serial : port {
  char *dname;
public:
  serial(int,char*);
  ~serial() { delete dname; }
  char *wport() { return dname; }   // mostra o nome da porta
  void send(char*);  // envia uma linha para o dispositivo serial
  char* receive();   // traz uma linha do dispositivo serial
};
```

A declaração indica que a classe *serial* é derivada da classe *port*, e que essa é uma classe-base privada. Este atributo está consistente com o critério de projeto básico — a classe-base deve conter somente funções comuns. Não é desejável nem necessário a qualquer tipo *port* específico permitir acesso direto de caractere à porta. Para a maioria deles, as classes especializadas definirão modos de acesso altamente restritos. Você não deseja que outro programador tenha a possibilidade de "quebrar" a especificação, penetrando por baixo da estrutura até a funcionalidade fundamental do próprio hardware básico; isso anularia a finalidade de se definir uma classe *port*. Declarando a base como privada, você restringe as funções de acesso de caractere ao uso interno pelas classes derivadas. Portanto, estas funções não estarão disponíveis diretamente através de uma variável declarada como *serial*.

O único membro privado de *serial* é o nome da porta, *dname*. Nenhuma parte do programa responsável pela transmissão de caracteres utiliza este valor, mas você pode usá-lo como um identificador para programas que abrem e fecham muitas portas diferentes que compartilham do mesmo dispositivo físico. (Isso pode ser útil num sistema operacional PC multitarefa como o OS/2.) De qualquer modo, está incluído o membro *wport( )* para permitir-lhe acessar esta cadeia de caracteres. A função destrutor é a mesma da classe original *port*, e, certamente, é suficientemente pequena para ser definida como *inline*.

A função construtor *serial( )* é semelhante à função construtor na primeira classe *port*. No entanto, ela executa uma tarefa adicional: ela não somente aceita o parâmetro *pnum* para si própria como também passa o parâmetro para sua classe-base *port* para inicialização. A única tarefa que ela executa além disso é estabelecer *dname* (veja Listagem 8.9). As funções-membro *send( )* e *receive( )* são idênticas às suas duplicatas na classe original *port*.

Listagem 8.9 O construtor para a classe *serial*

```
serial::serial(int pnum,char* device) : (pnum)
{
  dname = new char[strlen(device) + 1];    // estabelece o nome
                                           // da porta
  strcpy(dname,device);                    // inicializa-a
}
```

Finalmente, vamos testar esta nova classe num programa. O programa *sendstr2.c* (Listagem 8.10) envia uma linha de texto para a porta serial. Você tem que incluir a linha de texto como um argumento de comando, pois, do contrário,

o programa sai com uma mensagem de erro. O exemplo supõe que a porta serial seja *COM1:*, mas você pode, naturalmente, reescrevê-lo para se adaptar a qualquer configuração.

Listagem 8.10 Amostra de um programa emissor para testar a classe *serial*: *sendstr2.c*

```
#include <stream.h>

main(int argc,char* argv[])
{
  serial x(0,"com1:");        // declara um objeto serial

  if (argc == 1) {          // reclama falta de mensagem
    cout << "Por favor, inclua mensagem\n";
    exit(1);
    }
  x.send(argv[1]);                  // envia-a
}
```

O complemento do programa emissor, *recvstr2.c*, está na Listagem 8.11. Este aceita uma cadeia de caracteres provenientes da porta e mostra-os na tela. Esse programa está definido para *COM2:*, mas você pode usá-lo com qualquer porta que seja suportada por seu sistema.

Listagem 8.11 Um programa de recepção para complementar o programa emissor: *recvstr2.c*

```
#include <stream.h>

main()
{
  char buffer[80];

  serial x(1,"com2:");              // declara um tipo serial
  strcpy(buffer,x.receive());       // obtem um texto e copia-o
  cout << form("%s\n",buffer);      // exibe-o
}
```

Projetos de Programação

1. Reimplemente a classe-base *port* (Listagem 8.7) de maneira que ela use o sistema de interrupção do MS-DOS.

2. Usando *port* como classe-base, defina uma classe que manipule uma impressora serial. Inclua séries alternativas de caracteres e a necessária inicialização periférica.

Compactando uma Classe

Esse capítulo focalizou um exemplo interessante que usa C++ dentro do contexto do MS-DOS. Ele mostrou a interação importante entre o hardware, o sistema operacional e C++, e como projetar alguns novos tipos de dados, as classes *port* e *serial*. Você pode usar estas classes como recurso para muitos programas diferentes que precisam acessar a porta de comunicação serial. Você precisa apenas resolver um problema uma vez; depois você pode criar um objeto que possa ser usado repetidas vezes.

Precisamos discutir outro aspecto desse assunto: como compactar a classe para reutilizá-la. Até aqui, a finalidade principal deste capítulo foi examinar os detalhes das especificações e definições. Finalizando, tanto a especificação da classe — variáveis-membro e funções-membro — quanto o código que implementa as funções-membro foram incluídos no mesmo arquivo. No entanto, esta abordagem simples não é nem a única possível nem a melhor. Vamos investigar agora uma organização diferente.

Duas Tarefas na Criação de um Dado Tipo Classe

A criação de um dado tipo *classe* pode ser dividida em duas tarefas relativamente bem definidas:

- a declaração da classe

- a definição das funções-membro

A declaração define a estrutura básica e a organização do novo tipo. Você identifica os tipos e nomes de todas as variáveis-membros da classe. Você precisa especificar também o tipo que retorna de cada função-membro bem como sua lista de parâmetros. A planta da operação da classe está completa quando você designa cada membro às seções privada, pública, ou protegida, da classe.

A definição das funções-membro, em contraste com a declaração da classe como um todo, envolve a escrita do código que implementa cada função. Esta tarefa tradicional de programação não precisa estar fisicamente justaposta à declaração. Após você criar o molde para uma classe, o código de implementação somente precisa se referir ao molde. Faz sentido colocar a definição da classe num arquivo separado. Após você compilar este último arquivo, pode usá-lo na sua forma-objeto. O arquivo-cabeçalho serve, então, como uma espécie de ponto entre o programa que usa a classe e as instruções reais que foram usadas para compilá-lo.

Porém, há algumas restrições a este procedimento. A declaração da classe deve vir antes de quaisquer definições de função. Você precisa especificar também algumas funções *inline* dentro desta declaração. Embora estas duas restrições pareçam ser contraditórias, elas na realidade estão ligadas a duas coisas diferentes.

Compilando e Esquecendo

Após você criar um novo tipo de dado — como, por exemplo, a classe *port* — e depurá-lo, você não precisa recompilar o código cada vez que for usá-lo. Para usar essa classe, tudo o que você precisa realmente fazer é ligar o arquivo-objeto que contém suas funções de implementação com o novo programa. Em outras palavras, você pode compilar uma definição de classe para um arquivo-objeto — com a extensão *.obj* — e usar este enquanto mantém o arquivo-fonte em lugar seguro. Se o seu ramo de atividade é a criação de módulos de software, estes procedimentos fazem ainda mais sentido — você não precisa abrir mão de seus segredos porque não estará distribuindo código-fonte.

Naturalmente, a criação de uma forma-objeto de uma classe não é tão simples como induz o parágrafo anterior. Está faltando um elemento importane. Você precisa criar também um módulo de interface para servir de ponte entre o

código-objeto e o programa que o usará. A declaração da classe é essa ponte. Além do código-objeto, você precisa fornecer ao usuário do módulo um arquivo que contenha essa declaração. O exemplo *port4.h* (Listagem 8.12) mostra o conteúdo de um arquivo de interface para uma classe simples *port*. O arquivo de interface é tipicamente um cabeçalho — ou arquivo *.h* — que deve ser incluído em cada arquivo que contenha o código que se refere à nova classe.

Listagem 8.12 Um arquivo interface para uma classe simples *port*: *port4.h*

```
const com1 = 0X3F8,    // estabelece os enderecos padrao das portas
      com2 = 0X2F8;

class port {
  unsigned com;         // endereco base da porta
  char* name;           // nome da porta fornecido pelo usuario

public:
  port(int,char*);                    // inicializa a porta
  ~port() { delete name; }            // faz a "limpeza"
  char *wport() { return name; }      // qual e o nome?
   void pchar(int);                   // envia um unico caractere para a porta
  int gchar();                        // traz um unico caractere da porta
};
```

Projetos de Programação

1. Recompacte a classe *port* como um biblioteca, isto é, um arquivo com o código-objeto que contém a implementação das funções-membro da classe e um arquivo-cabeçalho que serve como interface para estas funções.
2. Faça o mesmo com a classe *serial*. Lembre-se de implementar esta classe usando a nova versão de *port*.

A Listagem Completa das Classes Port e Serial

Para sua conveniência no uso do código da porta serial em seus próprios projetos, a Listagem 8.13 — *portclas.h* — contém toda a declaração e definição da classe *port*. De maneira semelhante, *serclas.h* (Listagem 8.14) mostra a classe *serial* completa. A bem da clareza, a discussão neste capítulo omitiu algumas partes de menor importância destas definições.

Listagem 8.13 A definição completa da classe *port*: *portclas.h*

```
#define NO_EXT_KEYS

#include "\c5\include\conio.h"
#include <string.h>

const com1 = 0X3F8,     // estabelece o endereco padrao das portas
    com2 = 0X2F8;
class port {
  unsigned com;       // endereco base da porta
  char* name;         // nome da porta fornecido pelo usuario
  void pchar(int);    // envia um unico caractere para a porta
  int gchar();        // traz um unico caractere da porta

public:
  port(int,char*);                    // inicializa a porta
  ~port() { delete name; }            // faz a "limpeza"
  char *wport() { return name; }      // qual e o nome?
  void send(char*);  // envia uma linha para o dispositivo serial
  char* receive();          // traz uma linha do dispositivo serial
};

void port::pchar(int ch)
{
  unsigned status;
  for (;;) {              // loop de espera
    status = inp(com+5);                // obtem o status
    if ((status & 0x20) != 0) {         // verifica-o
      outp(com,ch);                     // envia o caractere
      break;                            // sai do loop de espera
      }
    }
}

int port::gchar()
{
  int ch;
  unsigned status;
  for (;;) {                  // inicializa loop de espera
    status = inp(com+5);              // obtem o status
    if ((status & 0x01) != 0) {       // verifica se o caractere
                                      // esta pronto
```

```
      ch = inp(com);                    // pega-o da porta
      return ch;                        // envia-o a quem o solicitou
      }
    }
}

port::port(int pnum,char* n)
{
  name = new char[strlen(n) + 1];       // estabelece o membro
                                        // nome da porta
  strcpy(name,n);                       // inicializa-o com o valor
                                        // dado pelo usuario
  com = (pnum != 1) ? com1 : com2;      // estabelece o endereco
                                        // da porta
}

void port::send(char* buffer)
{
  for (int i=0; i<=strlen(buffer); i++)     // esvazia o buffer

    pchar(buffer[i]);                   // para a porta
}

char* port::receive()
{
  char buffer[80];

  for (int i=0; i<80; i++)
    if ((buffer[i] = gchar()) == '\0')  // loop ate o fim da
cadeia
      break;
    return buffer;                      // retorna buffer
preenchido
}
```

Listagem 8.14 A definição completa da classe *serial*: *serclas.h*

```
#define NO_EXT_KEYS

#include "\c5\include\conio.h"
#include <string.h>

const com1 = 0X3F8,     // estabelece o endereco padrao das portas

    com2 = 0X2F8;
```

```
////////////////////////////////////////////////////////////////
// CLASSE BASE: port
////////////////////////////////////////////////////////////////

class port {
  unsigned com;      // endereco base da porta
public:
  port(int p) { com = (p!=1) ? com1 : com2; }     // construtor
  void pchar(int);  // envia um unico caractere para a porta
  int gchar();      // traz um unico caractere da porta
};

void port::pchar(int ch)
{
  unsigned status;
  for (;;) {                    // envio de caractere
    status = inp(com+5);
    if ((status & 0x20) != 0) {
      outp(com,ch);
      break;
      }
    }
}

int port::gchar()
{
  int ch;
  unsigned status;              // entrada de caractere
  for (;;) {
    status = inp(com+5);
    if ((status & 0x01) != 0) {
      ch = inp(com);
      return ch;
      }
    }
}

////////////////////////////////////////////////////////////////
// CLASSE DERIVADA : serial
////////////////////////////////////////////////////////////////

class serial : port {
  char *dname;
public:
  serial(int,char*);
```

```
  ~serial() { delete dname; }
  char *wport() { return dname; }
  void send(char*);  // envia uma linha para o dispositivo serial
  char* receive();   // traz uma linha do dispositivo serial
};

serial::serial(int pnum,char* device) : (pnum)
{
  dname = new char[strlen(device) + 1];   // estabelece o nome da
                                          // porta
  strcpy(dname,device);                   // inicializa-a
}

void serial::send(char* buffer)
{
  for (int i=0; i<=strlen(buffer); i++)    // envia caractere por
    pchar(buffer[i]);                      // caractere
}

char* serial::receive()
{
  char buffer[80];

  for (int i=0; i<80; i++)
    if ((buffer[i] = gchar()) == '\0')    // constroi a cadeia
                                          // no buffer
      break;
    return buffer;
}
```

Resumo

Este capítulo ilustrou a criação de três classes simples que acessam a porta serial do IBM-PC. Cada versão era uma ampliação da anterior. O primeiro exemplo explorou a transmissão básica de caracteres. Uma classe simples com um construtor, um destrutor e duas funções-membro caractere-orientadas — *pchar( )* e *gchar( )* — permitem-lhe enviar um único caractere pela porta e recebê-lo do outro lado. A esta aptidão caractere por caractere, o próximo exemplo acrescentou

membros especiais para a classe que permitiam que fosse enviada uma única cadeia de caracteres pela porta. Isso criou um tipo de dado mais útil. Finalmente, o último exemplo usou o conceito de classe derivada para criar uma hierarquia de tipos de dado *port*. Começando com um simples núcleo de recursos de entrada/saída — essencialmente os mesmos que a classe *port* original — como base,o capítulo demonstrou como definir classes *port* especializadas para manusear tipos específicos de situações.

Foram discutidos também alguns esboços importantes e novos estilos. Você viu como usar a estrutura inerente de uma definição de classe — sua natureza em duas partes, da declaração e definição — para criar um programa ainda mais modular e estruturado.

Makron
Books

Capítulo 9
Usando C++ no Sistema UNIX

E. Willian Leggett

Nos últimos anos, o sistema operacional UNIX tornou-se um ambiente popular de desenvolvimento de software. Veja, por exemplo, a obra de Waite, Martin, and Prata — *UNIX System V Primer*, publicada por Howard W. Sams & Co. A razão principal desse crescimento é que o sistema proporciona muitos compiladores, assemblers (montadores), e outras ferraments de desenvolvimento de software incluindo uma grande variedade de utilitários para manuseio de arquivos. Agora que temos versões mais baratas do UNIX disponíveis para PCs, você pode até mesmo desenvolver software de MS-DOS no ambiente mais rico do UNIX. Na verdade, muitos dos programas em MS-DOS estão baseados em utilitários similares que têm estado disponíveis durante anos no sistema UNIX.

Todos os programas deste livro (com exceção daqueles do Capítulo 8) funcionam sob o UNIX bem como no PC sob o MS-DOS. No entanto, programas de "sistemas" mais avançados, tais como alguns utiliários de manuseio de arquivo,

ou programas de telas gráficas sofisticadas, geralmente usam chamadas de funções hardware específicas, ou OS específicas (OS = Operacional System — Sistema Operacional), e características de manuseio de dispositivos. Por exemplo, para um programa executar uma tarefa razoável com gráficos do tipo "bit-mapped", ou com gerenciamento de janela, ele deve usar ou um estrato de funções hardware-específicas, ou usar um "pacote", como por exemplo, *X Windows MIT*. Esse capítulo mostra-lhe como desenvolver recursos que ajudam os programas UNIX a se comunicarem uns com os outros. Para que você entenda completamente o que isso acarreta, vamos em primeiro lugar examinar rapidamente as características básicas do sistema UNIX.

Breve Introdução ao UNIX

O sistema UNIX foi desenvolvido no início da década de 70 nos Laboratórios Bell AT&T especificamente como um ambiente de desenvolvimento de software. Através dos anos, ele adquiriu uma variedade de ferramentas de desenvolvimento de software como linguagem de programação, recursos de comunicações em redes, ferramentas de preparação de documentos, editores, controle de versão de código-forte, e outras utilidades muito numerosas para serem mencionadas aqui. Nos primeiros tempos de seu desenvolvimento, ele foi amplamente reescrito na linguagem C então recentemente desenvolvida. A maioria de seus instrumentos está escrita em C e pode facilmente ser conduzida de um sistema UNIX para outro. Em virtude de seus muitos recursos de desenvolvimento de software e de sua disponibilidade nas universidades, ele tornou-se bastante popular.

UNIX é um sistema operacional *multitarefa* que permite que muitos programas sejam rodados ao mesmo tempo. Por exemplo, sob o UNIX, um programador pode ler correio eletrônico enquanto roda o construtor de progrma *make* no modo "background". O processador reserva tempo para ambos os processos, mas o programa em "foreground"é o que está associado com a entrada do teclado. Naturalmente, um único processador pode rodar apenas uma tarefa de cada vez. No entanto, vários processos podem ser completados parcialmente de maneira concorrente. Por exemplo, um pode executar instruções da CPU enquanto outro espera pelo acesso ao disco.

Na verdade, muitos usuários do UNIX podem executar várias tarefas como essas simultaneamente, tornando o UNIX um exemplo de um sistema operacional *multiusuário.* Tipicamente, um sistema UNIX tem vários usuários conectados e trabalhando ao mesmo tempo. Um usuário executa cálculos enquanto outros introduzem dados, lêem arquivos, ou tomam um café, otimizando assim o uso dos recursos do computador. No entanto, mesmo num PC de um único usuário, ou numa "work-station" (estação de trabalho), a aptidão do sistema operacional em rodar processos simutâneos em "background" pode ser um recurso tremendo.

Para evitar que esses processos e usuários se intrometam nos dados um do outro, o sistema UNIX utiliza o "virtual memory mapping" (mapeamento da memória virtual), de maneira que os ponteiros de um usuário só poderão acessar e modificar as localizações de memória destinadas àquele usuário. Ou seja, um mapeamento extra garante que os dados de um usuário do endereço AAOO estão mapeados para uma região diferente da memória "real" em relação à localização AAOO de outro usuário (veja Figura 9.1).

Por outro lado, o MS-DOS não é multitarefa: um programa deve esperar enquanto outro é executado, mesmo que a saída do outro programa seja mostrada na tela em outra janela. O sistema operacional OS/2 é multitarefa, mas não multiusuário. Os programas de um único usuário do PC compartilham a CPU, cada um usando o processador por um curto espaço de tempo, num modo "tempo-compartilhado".

O conceito de processo ou *tarefa* — um programa com seu próprio código, dados, pilha e linha de execução — é fundamental para o UNIX e para outros sistemas operacionais multitarefa. A função principal de um sistema operacional multitarefa é proporcionar os serviços de sistemas necessários aos vários processos de maneira equitativa, enquanto protege as tarefas umas das outras.

O UNIX e o OS/2 usam técnicas de gerenciamento de memória para proteger e servir a múltiplas tarefas. Devido ao fato de que a maioria dos programas não escreve intencionalmente no espaço de código do sistema operacional, tanto o UNIX quanto o OS/2 permitem que os processos compartilhem o código, reduzindo assim as necessidades de espaço de memória. (O MS-DOS tipicamente não usa o gerenciamento de memória para proteger processos múltiplos comutados, residentes na memória.)

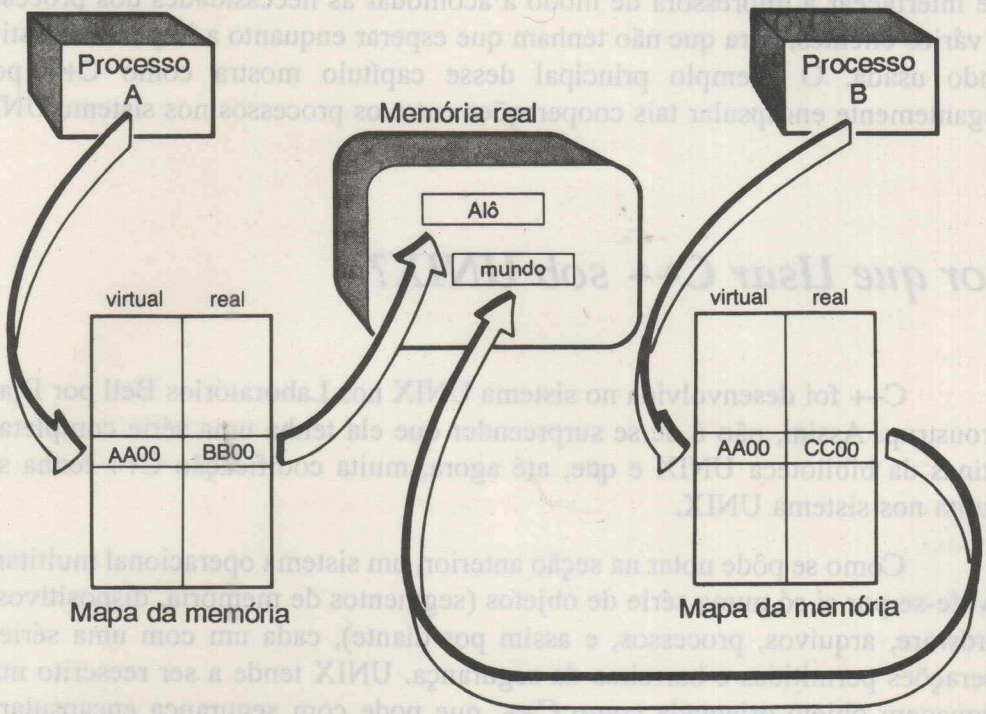


Figura 9.1 Mapa de proteção da memória virtual.

Devido a muitos processos terem que compartilhar recursos (usualmente apenas um processo de cada vez pode usar uma impressora ou ler o teclado), um sistema operacional multitarefa usualmente tem um núcleo (*kernel*) separado, ou coleção de rotinas de sistemas para acessar o disco, decidir qual processo acessa em seguida a CPU (*schedulling*), alocação de memória, e assim por diante. Essas rotinas são acessadas tipicamente por "armadilhas" especiais ou *chamadas de sistema* (system calls). O sistema operacional manipula as interrupções (interrupts) de hardware que indicam eventos, tais como o movimento do mouse (ratinho), ou a leitura de um bloco em um disco, e também emite comandos de I/O para dispositivos em nome de todos os usuários. Isso permite que processos múltiplos utilizem os recursos do sistema de maneira eficiente e, o que é mais importante, protege os processos um do outro.

No entanto, há situações em que você quer que os procesoss interajam, ou que cooperem entre si para executar uma tarefa. Por exemplo, você talvez tenha

que interfacear a impressora de modo a acomodar as necessidades dos processos de vários clientes, para que não tenham que esperar enquanto a impressora estiver sendo usada. O exemplo principal desse capítulo mostra como C++ pode elegantemente encapsular tais cooperações entre os processos nos sistema UNIX.

Por que Usar C++ sob UNIX?

C++ foi desenvolvida no sistema UNIX nos Laboratórios Bell por Bjarne Stroustrup. Assim, não é de se surpreender que ela tenha uma série completa de rotinas da biblioteca UNIX e que, até agora, muita codificação C++ tenha sido escrita nos sistema UNIX.

Como se pôde notar na seção anterior, um sistema operacional multitarefa divide-se por si só numa série de objetos (segmentos de memória, dispositivos de hardware, arquivos, processos, e assim por diante), cada um com uma série de operações permitidas e barreiras de segurança. UNIX tende a ser reescrito numa linguagem objeto-orientada como C++, que pode com segurança encapsular tal funcionalidade. AT&T e SUN Microsystems possivelmente estarão usando C++ para escrever o sucessor do Sistema UNIX V Release 4 (combinação de Systems V e Berkeley Softaware Distribution, BSD, UNIX).

Além do projeto de um sistema ou aplicação (que pode ficar mais claro usando técnicas objeto-orientadas), o obstáculo principal para que se possa produzir uma codificação de qualidade é a falta de uma interface formal entre módulos escritos por diferentes pessoas num projeto de equipe. Mesmo quando o trabalho envolve uma pessoa só, ela pode ser beneficiada por uma clara separação de funcionalidade e especificação precisa das interfaces de objeto, como pode atestar alguém que venha a examinar seu próprio programa um ano mais tarde! C++ proporciona recursos embutidos significativos para ocultamento de dados, que ajudam a evitar tais problemas. Os modelos de funções ANSI C também suportam forte modelagem, mas ANSI C não possui mecanismo de ocultamento de dados.

Usando os mecanismos público/privado reforçados pelo tradutor C++, um programador de sistemas pode, por exemplo, construir um módulo (classe) para manipular um bloco de controle de processo e ter a certeza de que ninguém poderá

usar aquele objeto, de forma alguma, sem permissão explícita. Muitas pessoas que têm trabalhado em aplicações em equipe com limites rígidos (não são todos assim?) concordam que regulamentos e convenções raramente são suficientes. C++ é uma linguagem próspera porque, quando usada corretamente, garante a segurança tipo/classe sem lançar mão de incômodas listas "exportadas", "camisas de força" sintáticas, ou descrições e documentações em linguagem vaga, tediosas de se usar e que facilmente são passadas por cima. C++ permite que os programadores estabeleçam relações e interfaces sucintamente, evitando assim a incerteza e confusão que resulta até mesmo dos melhores manuais de "Padrões de programação".

Programar sistemas, ou escrever os recursos que permitem aos processos de aplicação cooperarem entre si, é uma tarefa difícil de ser executada com perfeição. Nela, as mensagens, processos e comunicações de "resultados interessantes" são os principais candidatos às classes C++ e objetos. Também a estruturação adicional de interface de C++ é ainda mais útil nesse tipo de desenvolvimento. Vamos agora examinar os processos UNIX de multitarefa e cooperação e ver como C++ pode elegantemente encapsular alguns dos "truques" de codificação UNIX System V.

Usando o Sistema UNIX de Multitarefa

O sucesso do sistema UNIX é devido não somente às suas inúmeras utilidades, mas também à filosofia que há por trás de seu desenvolvimento. Essa "filosofia UNIX" é demonstrada com mais eloqüência na obra: *The UNIX Programming Environment* de Kernighan e Pike. A idéia principal que há por trás da filosofia UNIX é criar pequenas e eficientes ferramentas, enquanto proporciona a estrutura para combiná-las em poderosas formas.

O instrumento ideal UNIX tem poucas opções e poucas dependências dos outros utilitários com os quais ele pode ser solicitado a interagir. Por exemplo, os idealizadores do UNIX notaram que a maioria dos processos não tem um arquivo de entrada (dispositivo) e um arquivo de saída (dispositivo) primários. Eles estabeleceram que a conexão dessas streams de entrada e saída com outros

programas ou arquivos deveria ser semelhante à parada (desfile) dos elefantes no circo, onde a tromba de um segura a cauda do próximo. Deste modo, um programa ideal não depende de onde seus dados vêm e nem para onde vão.

Muitas pessoas identificam o sistema UNIX com seu *invólucro*, ou o interpretador de linguagem de comando que serve de interface primária para os usuários UNIX. O *invólucro* fornece uma notação sucinta para interligar o I/O de vários instrumentos e utilitários a outros programas e arquivos. Assim, você pode conectar uma série de programas como um processo "pipeline" (um encanamento), a saída de um tornando-se a entrada do próximo, como mostrado na Figura 9.2.

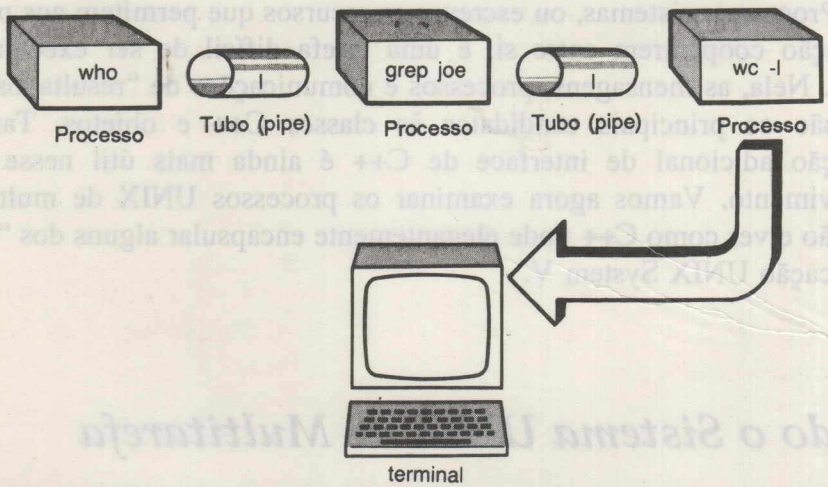


Figura 9.2 Um exemplo de um processo "pipeline" UNIX.

O comando *who* usualmente lista no terminal as pessoas envolvidas no sistema. Neste exemplo, sua saída é direcionada ao programa *grep* (*grep* significa comparador generalizado de padrão regular de expressão) com o argumento "joe", que dá como respostas apenas as linhas que contêm a cadeia "joe". Finalmente, a saída de *grep* é "canalizada" (piped) a *wc*, um programa de contagem palavra-e-linha, que (com a opção "_1") imprime o número resultante de linhas no terminal, que é seu dispositivo de saída. Assim, você pode facilmente descobrir quantas linhas contêm Joe sem construir para isso um utilitário especial.

Os "pipes" apresentam um problema que é o inverso daquele da proteção da memória, isto é, como compartilhar informação num ambiente protegido. No

entanto, os "pipes" não são adequados se os processos UNIX não tiverem um ancestral comum ou se sua comunicação não for previsível. Estes casos necessitam de um outro mecanismo. Assim, se eles precisam cooperar por um mecanismo diferente da "pipeline", como fazer?

Os tipos de programas que precisam comunicar-se por meios mais gerais do que os "pipes" são geralmente programas de sistemas chamados de "demons" os quais controlam dispositivos especiais ou telas de janelas múltiplas ativas (multiactive-window display screens). Devido à concorrência dos programas não poder ser conhecida de antemão, é quase impossível ao sistema proporcionar canais (pipes) entre eles. Eles poderiam comunicar-se através de arquivos. Mas, se os processos múltiplos são protegidos contra alteração de dados um pelo outro, então como um processo sabe quando é seguro escrever num arquivo para ser lido por outro? O sistema UNIX V de bloqueio de arquivo pode ajudar a resolver esse problema, mas em aplicações de tempo real, como, por exemplo, controle de robô, esse método pode ser muito lento. O compartilhamento de recursos onde o tempo não é crítico, como, por exemplo, no uso da impressora, é melhor implementado como simples arquivos bloqueados.

O sistema UNIX V tem vários recursos de comunicação interprocesso (LPC) que você pode usar com processos que precisam cooperar mais eficientemente. Estes recursos são:

- "Named" pipes (canais denominados)
- Mensagens LPC
- Memória compartilhada
- Semáforos

Named Pipes ou FIFOs

Normalmente os canais de Sistema UNIX requerem os dois processos de comunicação para compartilhar um ancestral comum. O processo-pai abre ambas as extremidades do canal e conecta a saída de um processo à entrada do próximo processo aplicativo. Se um processo demon, com um que UNIX executa no início para gerenciar uma fila de espera de impressão, precisa responder aos processo do

usuário à medida que solicitam serviço, isto pode abrir um *named pipe* ou FIFO (primeiro que entra, primeiro que sai) para ir realizando a leitura e pacientemente aguardar a abertura de algum dispositivo de escrita na outra extremidade pelo envio de uma mensagem. Muitos desses dispositivos de escrita podem usar o FIFO para enviar as mensagens aos leitores. Escrever em *named pipes* é garantidamente *atômica*, isto é, parte de uma mensagem não será misturada ao conteúdo de outra. No entanto, quando várias mensagens estiverem enfileiradas no canal (pipe), o demon pode ter dificuldade para determinar onde uma termina e onde a outra inicia.

Continuando com este exemplo de um objeto FIFO, o processo *init* do UNIX, que iniciou o programa "demon" para uso da impressora, não sabe de antemão quando um programa começará a imprimir suas saídas. Portanto ele são pode reservar antecipadamente um canal "regular". Assim, quando um programa envia sua saída a um processo lop (line printer) através de um canal regular, o processo *lp* escreve num arquivo especial (FIFO do Sistema V do programa "printer demon").

Usando um objeto classe *fifo,* o código para *lp* fica sendo algo assim:

```
include "fifo.h"

    fifo print_fifo;

    fifomsg fifo_msg(60,lp_cookie);

    print_fifo.attach("/usr/spool/lp/print.fifo");

    fifo_msg.addmsg(filename);
    fifo_msg.endmsg();
    print_fifo.sendmsg(&fifo_msg);
```

Esse código aloca uma FIFO chamada *print_fifo* e uma mensagem, *fifo_message*, que contém um caractere especial (um "magic cookie") para auxiliar o programa "demon spooler" a delimitar as mensagens na fila. O programa liga-se à FIFO, coloca na sua mensagem (da FIFO) o nome do arquivo a ser impresso, declara que a mensagem terminou e envia-a ao "demon" através do "named pipe".

O objeto-classe *fifo* pode também conter uma interface parcial assim:

```
class  fifomsg {
  int size;       // tamanho maximo das mensagens
```

```
  char cookie;  // primeiro caractere e' um "magic cookie"
  char* body;   // corpo da mensagem enviada

public:
  fifomsg(int maxsize, char magcookie)
    { size = maxsize; cookie = magcookie;
      body = new char[maxsize];
      body[0] = NULL; }

  void addmsg(char* newstring)
    { strcat(body, newstring); }
  void endmsg() { strncat(body, "\n", 1); }
};

class fifo {
  int fd;  // arquivo descritor UNIX

public:
  void attach(const char* name)
    { fd = open(name, O_RDONLY); }

  void sendmsg(fifomsg* msg)
    { write(fd, msg, strlen(msg) }
};
```

A mensagens na FIFO têm um comprimento máximo e contêm um sinal especial para simplificar a procura do início das mensagens (o "magic cookie") além do corpo da mensagem. As funções-membro *fifo attach()* e *sendmsg() então simplesmente chamam open()* e *write()* respectivamente.

Naturalmente, essa implementação carece da necessária verificação de erro e verificação de limites de cadeia (string). No entanto, se você acrescentar a verificação de erro, esse objeto FIFO será suficientemente seguro para ser usado em inúmeras aplicações. Assim, a programação objeto-orientada permite-lhe criar classes que tenham interfaces seguras e bem especificadas, as quais você pode facilmente reutilizar em outras aplicações.

De que maneira o sistema UNIX cria a FIFO e a manda ao diretório */usr/spool/lp* na primeira posição? A resposta é que o comando UNIX *mknod* usa a opção *-p* para criar um nó "named pipe", ou um arquivo especial UNIX FIFO, em um diretório. Naturalmente, o *lpdemon* deve saber o nome do nó de sua FIFO e abri-lo para leitura.

Mensagens IPC UNIX

Uma seção anterior discutiu rapidamente como as mensagens poderiam ser usadas para comunicação interprocesso. No System V, as *mensagens IPC* são como caixas de correio mantidas no espaço de endereçamento do núcleo (*kernel*). Elas têm duas vantagens sobre os "named pipes": permitem-lhe alocar um buffer suficientemente grande para conter todas as mensagens esperadas e — aqui a diferença principal — mantêm cada mensagem como uma entidade distinta. Assim, mesmo que um processo errante coloque uma mensagem "maluca" na pilha, ao leitor pode ignorá-la e continuar com a próxima mensagem válida. Se um processo que envia uma mensagem necessitar de uma resposta, então aquele processo pode abrir sua própria fila de mensagens e, em resposta à mensagem que ele envia, incluir a chave que identifica sua mensagem de resposta.

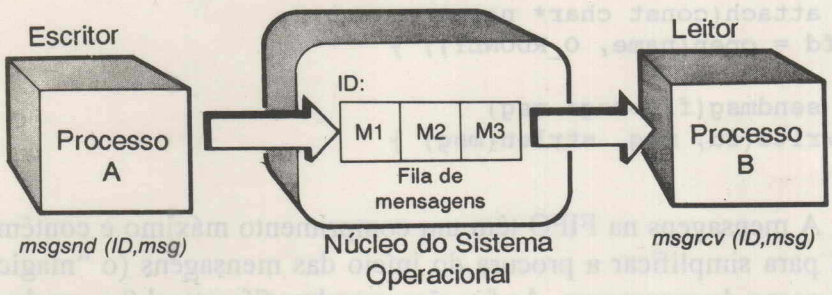


Figura 9.3 Tramitação de mensagem no sistema UNIX V.

Se você reescrever o programa *lp* do exemplo anterior como uma fila de mensagem IPC em vez de uma FIFO, o programa ficará assim:

```
include <sys/tipos.h>
include <sys/ipc.h>
include <sys/msg.h>
include "msgs.h"
    msgq  print_queire;
    ipc_msg ips_msg(60,lp_cookie);
    print_queue.attach(LP_KEY);
    ipc_msg.addmsg(filename);
```

```
    ipc_queire.sendmsg(&ipc_msg);
    print_queue.sendmsg(&ipc_msg);
```

Observe que foram necessárias apenas poucas mudanças — os nomes da variável e da classe. As funções e seus argumentos são quase idênticos àqueles do exemplo da FIFO. Isso sugere que você poderia ter usado realmente métodos objeto-orientados definindo uma classe-base *ipc* e derivando as classes *fifo* e *msgq* a partir dela.

A implementação, no entanto, não é assim tão similar. As diferenças entre os exemplos são importantes, porque elas mostram como você pode mudar uma implementação sem mudar a interface com o usuário. Isso é, em essência, programação objeto-orientada.

```
class ipc_msg {
      struct msgbuf sndbuf;    // em sys/msg.h
  int size;       // tamanho maximo da mensagem
  int sndsize;    // tamanho da mensagem enviada
  char* body;     // corpo da mensagem a ser enviada

public:
  ipc_msg(int maxsize, char magcookie)
    { size = maxsize; sndbuf.mtype = magcookie;
       body = new char[maxsize];
      body[0] = NULL; }

  void addmsg(char* newstring)
    { strcat(body, newstring); sndbuf.mtext = body;
      sndsize = strlen(body); }

  void endmsg()
    { strncat(body, "\n", 1); sndsize = strlen(body); }

  int len() { return sndsize; }
};

class msgp {
  int mqid;    // identificacao da fila de mensagem UNIX

public:
  void attach(const int key)
    { mqid = msgget(key, 0); }

  void sendmsg(const ipc_msg * msg)
```

```
    { msgsnd(mqid, msg, msg.len(), 0); }
};
```

A estrutura msgbuf está definida em *sys/msg.h* e inclui *long mtype* e *char * mtext*. O ponto-chave aqui é que as mensagens IPC proporcionam separação intrínseca de mensagem; assim, não é necessártio o caractere "magic cookie", que não pode ser usado no corpo da mensagem. O buffer de mensagem do System V tem também separadamente um código embutido de tipo de mensagem que permite ao receptor solicitar todas as mensagens de um certo tipo, que estão na fila, antes de ler as mensagens dos outros tipos. A rotina *msgget()* é uma chamada do sistema UNIX System V que retorna o identificador (interno) da fila de mensagens associado com a chave especificada — pense nele como se fosse uma função *open()* (arquivo) para filas de mensagem) — e a chamada de sistema *msgsnd()* envia uma mensagem semelhante a *write()* (arquivo) para aquela fila. A chave deve ser especificada para prováveis clientes (geralmente num arquivo de cabeçalho). Assim, a fila de mensagem ID é um identificador interno do sistema que se assemelha ao descritor de arquivo para uma abertura de arquivo.

Memória Compartilhada

Um dos meios mais eficientes de compartilhar dados é mapear o mesmo bloco de endereçamentos "reais" em dois ou mais espaços virtuais de processo (no mesmo endereço ou em endereços diferentes). Depois, qualquer um dos processos pode ler ou escrever naquela área de memória. Veja a Figura 9.4.

Note que cada processo não precisa pôr os dados no mesmo endereço virtual (privado). No entanto, se você armazenar ponteiros na área compartilhada, deve mapeá-los todos para a mesma localização, e nenhum ponteiro deverá apontar para um endereço "privado" não compartilhado de qualquer processo. Naturalmente, se ambos os processos tentam escrever na mesma localização ao mesmo tempo, pode sobrevir o caos. Mais adiante, um outro exemplo usa memória compartilhada e demonstra como resolver estes problemas.

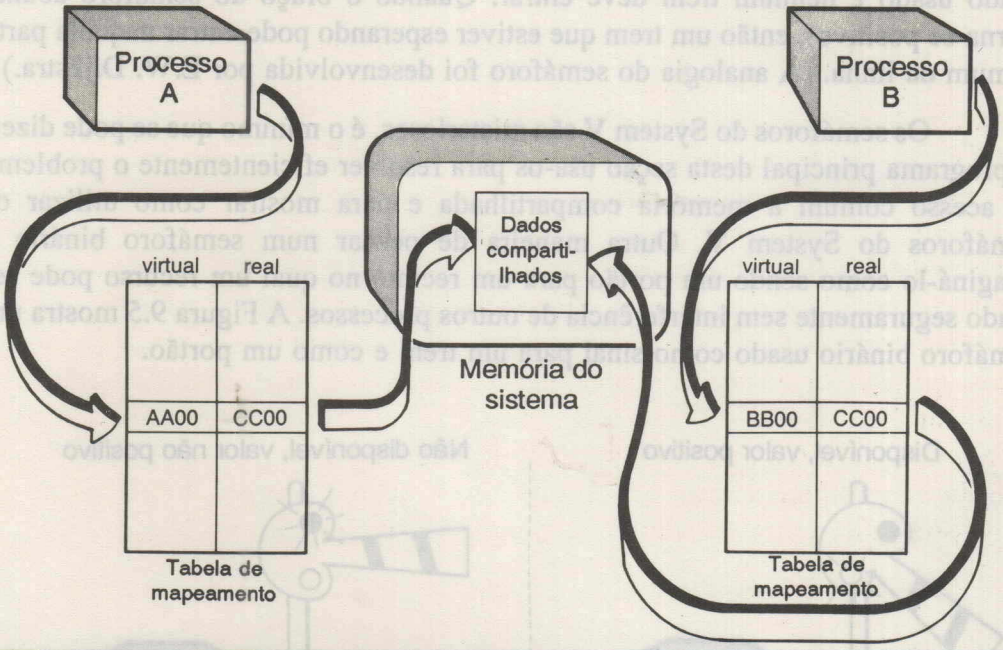


Figura 9.4 Mapeamento de memória compartilhada.

Semáforos

Os *semáforos* constituem um método popular para controlar acesso a um recurso, como uma região de memória compartilhada. Um semáforo pode ser um valor binário, que indica se um recurso está disponível ou não, ou um valor de contagem, que indica quantos recursos estão disponíveis. Se não há nenhum disponível, o semáforo tem um valor não positivo, e o processo deve esperar até que um recurso seja "afixado" ou retornado ao banco. Se um semáforo binário for zero, ele indica que o recurso está indisponível. Para semáforos de contagem, um zero ou um valor negativo indicam o número de processos que necessitam do recurso (além daqueles recursos no banco que ainda estão sendo usados).

O nome *semáforo* veio da idéia de um sistema de controle de tráfego ferroviário desenvolvido no século dezenove para evitar colisões. Se o semáforo

está em uso (não positivo, ou braço levantado), então aquele trecho da linha está sendo usado e nenhum trem deve entrar. Quando o braço do semáforo abaixa (torna-se positivo), então um trem que estiver esperando pode entrar naquela parte comum da linha. (A analogia do semáforo foi desenvolvida por E.W. Dijkstra.)

Os semáforos do System V são misteriosos, é o mínimo que se pode dizer. O programa principal desta seção usa-os para resolver eficientemente o problema de acesso comum à memória compartilhada e para mostrar como utilizar os semáforos do System V. Outra maneira de pensar num semáforo binário é imaginá-lo como sendo um portão para um recinto no qual um recurso pode ser usado seguramente sem interferência de outros processos. A Figura 9.5 mostra um semáforo binário usado como sinal para um trem e como um portão.

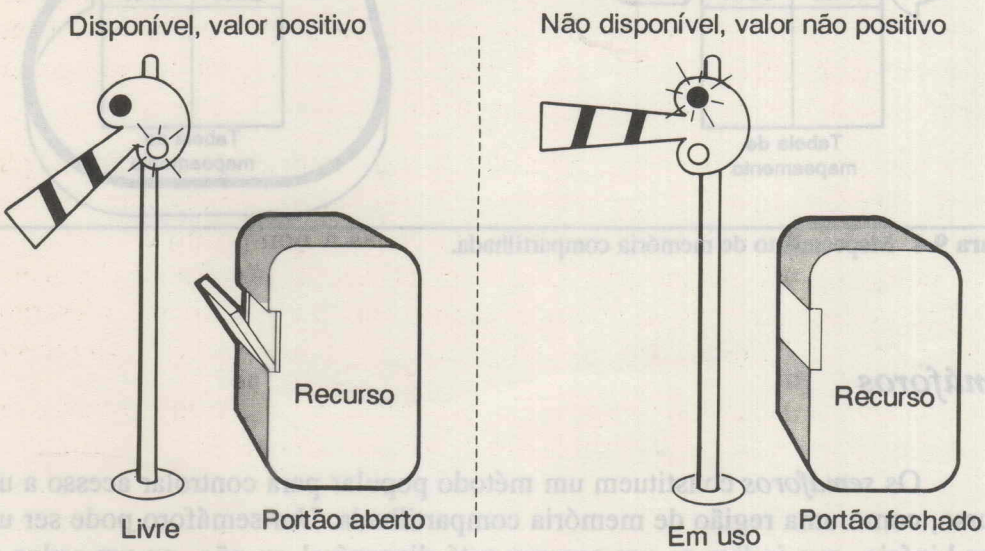


Figura 9.5 Um semáforo binário.

A próxima seção explica como usar a memória compartilhada e os semáforos para proporcionar um acesso compartilhado seguro e um buffer comum.

Monitores e Regiões Críticas com o System V

A seção anterior discutiu vários mecanismos IPC para resolver um problema de leitura/escrita concorrente (o leitor era o *lp* demon). Suponha que você queira implementar um buffer temporário na memória que será compartilhado por muitos "escritores" e um único leitor. Isso seria útil num sistema de controle de automatização de uma fábrica. Os "escritores" retornam valores provenientes dos sensores nas linhas de montagem, e então o leitor pode fazer a correlação das cargas nas linhas para mostrar a situação geral e talvez comparar a atividade nas linhas com vistas a aumentar a eficiência. O exemplo principal deste capítulo não é especificamente este. Ele mostra como os "escritores" podem, com segurança, escrever mensagens completas num buffer comum de registro, do qual um único leitor pode retirar as mensagens. Você poderia usar estes programas para manter um registro da atividade do sistema, ou para conter mensagens que são geradas por processos múltiplos, para fins de depuração.

O exemplo da memória compartilhada para conter as linhas de mensagem, e, assim, ele deve evitar que o processo escreva o mesmo registro ao mesmo tempo e não deixar que o leitor atualize o ponteiro de leitura enquanto o "escritor" o estiver utilizando. O programa precisa também informar como o leitor sabe quando o buffer está vazio e quando ele contém alguma coisa a ser lida, e como o "escritor" sabe quando o buffer está cheio e quando tem espaço para escrever novo registro.

Algumas vezes, processos múltiplos precisam acessar um recurso, mas não se pode permitir que eles utilizem esse recurso ao mesmo tempo. Quando isso ocorre num programa, as seções do programa que acessam o recurso são chamadas de *regiões críticas*. Uma maneira de proteger uma região de registro compartilhada é usar uma estrutura *monitor*. (O termo *monitor* foi criado por C.A.R. Hoare.) Tipicamente, utilizamos um semáforo para montar guarda à entrada (portão) do monitor a fim de permitir que apenas um processo de cada vez esteja ativo dentro dele. Depois podemos utilizar outro conjunto semáforo/portão para sinalizar que o buffer contém linhas de mensagens, e ainda outro para sinalizar que ele não está cheio atualmente, ou que o buffer tem ainda espaço para outra linha de mensagem. A área de memória compartilhada e seus semáforos implementam o monitor para a região crítica do buffer mostrado na Figura 9.6.

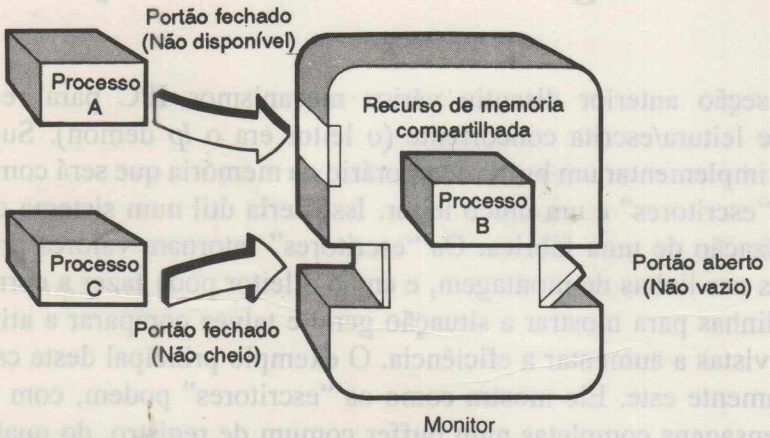


Figura 9.6 Semáforo e memória compartilhada como um monitor.

Na Figura 9.6, o Processo A está esperando para entrar na região monitorada. O Processo B entrou no monitor (ele "fechou o portão") e está pronto para atualizar o semáforo "não-cheio", ou abrir o portão para o Processo C que está esperando. Posteriormente apresentaremos descrições mais completas destas ações, e, numa próxima seção, daremos uma visão geral sobre como implementar este monitor e seus semáforos.

Problemas em Processamento Concorrente

Quando você faz um programa para qualquer sistema de processo concorrente, deve ater-se a vários princípios básicos:

- Evitar condições de "competição"
- Evitar impasse (beco sem saída)
- Evitar "soterramento"
- Tentar ser moderado

Existe uma condição de "competição" quando, por exemplo, dois "escritores" tentam escrever o mesmo registro ao mesmo tempo. Infelizmente, não é

sempre o mais rápido que vence a competição — geralmente ambos perdem, resultando num emaralhado de dados. Evitar esta condição é uma das principais razões para se idealizar um monitor.

Os impasses são mais sutis e portanto mais difíceis de evitar ou detectar. A maior parte da discussão de um livro-texto típico de sistemas operacionais refere-se aos problemas de proporcionar serviços enquanto se evitam impasses. O exemplo mais simples de impasse ocorre quando dois processos têm cada um um recurso e um quer o recurso do outro. Se nenhum dos dois pode ser forçado a desistir de seu recurso, eles permanecerão para sempre bloqueados num "abraço fatal".

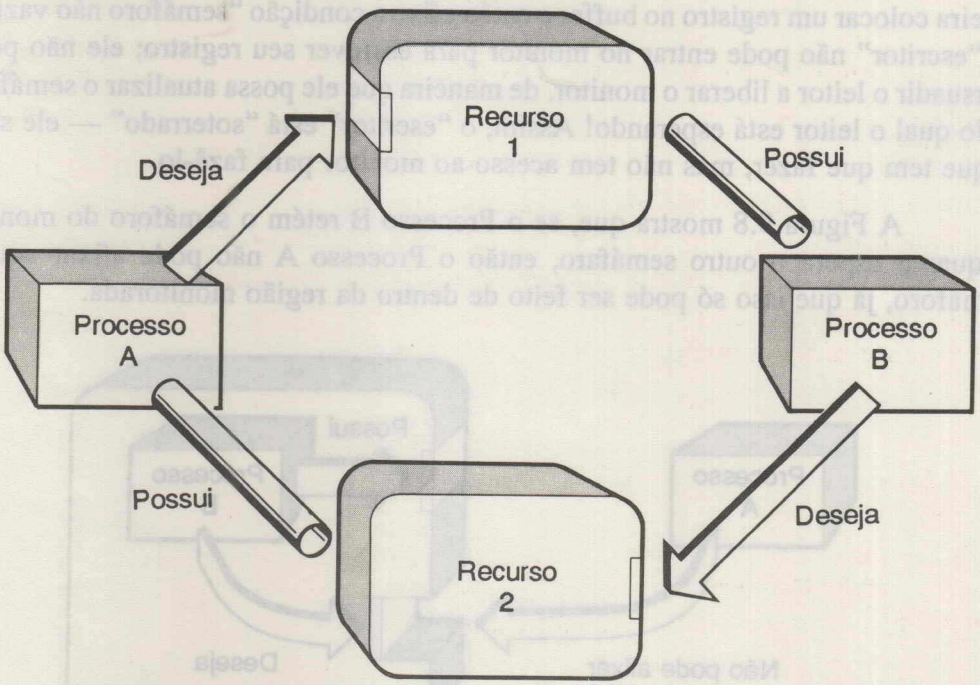


Figura 9.7 Impasse.

Usando o exemplo do buffer, suponha que um processo "escritor" esteja dentro do monitor esperando que o buffer se torne não-cheio. Se o processo leitor alcança o semáforo não-cheio primeiro, então tenta entrar no monitor, e os dois

processos ficarão para sempre esperando um pelo outro. Ou seja, cada um tem o recurso de que o outro precisa, mas nenhum deles cede seu recurso nem desiste de esperar que o outro processo libere o recursos que tem. Esse impasse ocorre quando nenhum dos processos pode continuar; assim os recursos tornam-se inúteis. A solução é não permitir que um processo alcance outro semáforo a menos que ele esteja dentro do monitor primeiro (tenha alcançado o semáforo do monitor). Devido a um processo no monitor estar sempre sozinho, ele não pode estar num impasse.

Agora, considere outro problema que pode aparecer se o leitor tiver o semáforo do monitor (ele está dentro da região monitorada) e estiver esperando que a condição semáforo não-vazio se torne verdadeira. Suponha que um "escritor" queira colocar um registro no buffer e então afixe a condição "semáforo não vazio". O "escritor" não pode entrar no monitor para escrever seu registro; ele não pode persuadir o leitor a liberar o monitor, de maneira que ele possa atualizar o semáforo pelo qual o leitor está esperando! Assim, o "escritor" está "soterrado" — ele sabe o que tem que fazer, mas não tem acesso ao monitor para fazê-lo.

A Figura 9.8 mostra que, se o Processo B retém o semáforo do monitor enquanto espera o outro semáfaro, então o Processo A não pode afixar aquele semáforo, já que isso só pode ser feito de dentro da região monitorada.

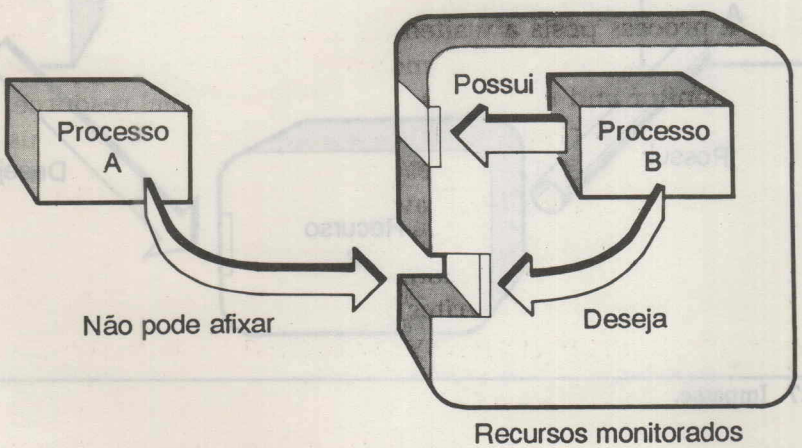


Figura 9.8 Soterramento (Livelock).

A solução clássica para esse problema com monitores é exigir que sempre que um processo dentro do monitor precisar de um recurso, ele deve ceder o monitor até que o recurso esteja disponível. Um processo nunca deverá ficar "dormindo" (esperando indefinidamente) de posse do semáforo do monitor (isto é, dentro do monitor), ou certamente ocorrerá um impasse ou um "soterramento".

Na Figura 9.9, o Processo B tinha o monitor (semáforo G), mas precisa do recurso NF, de modo que ele então cede o monitor (tornando G disponível) e espera fora dele até que NF esteja disponível. Enquanto isso, o Processo A pode ter acesso ao monitor (semáforo G), entrar nele, e afixar (liberar) o semáforo NF. Posteriormente, quando NF é afixado e A deixa o monitor, B estará habilitado a entrar no monitor e usar o recurso NF agora disponível.

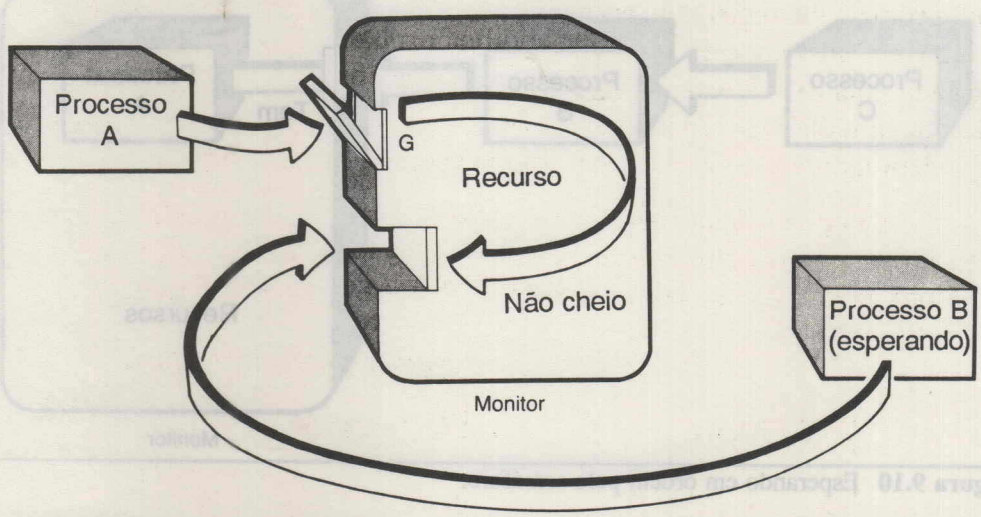


Figura 9.9 "Dormindo" fora do monitor.

Evitando Problemas de Processamento Concorrente

O exemplo do buffer deve tratar o leitor imparcialmente quando este estiver esperando pelo semáforo não-vazio e tratar os múltiplos "escritores" imparcialmente (na ordem de suas solicitações) quando eles estiverem esperando pelo semáforo não-cheio. Para ser imparcial, o sistema operacional UNIX mantém filas do tipo "primeiro que entra, primeiro que sai" para processos que estão

esperando pelo recurso que está sendo protegido pelo semáforo. Algumas implementações de monitor vão ainda mais longe e, quando um processo anuncia que espera por um semáforo, o processo que ocupava o monitor imediatamente sai e permite que o processo que estava esperando ocupe o monitor e use o recurso desejado (agora disponível).

Na Figura 9.10, o Processo A está no monitor e, portanto, tem controle sobre o semáforo G. Os Processo B e C estão em fila esperando pelo semáforo do monitor. Quando A deixa o monitor e libera G, B alcança G e entra no monitor primeiro.

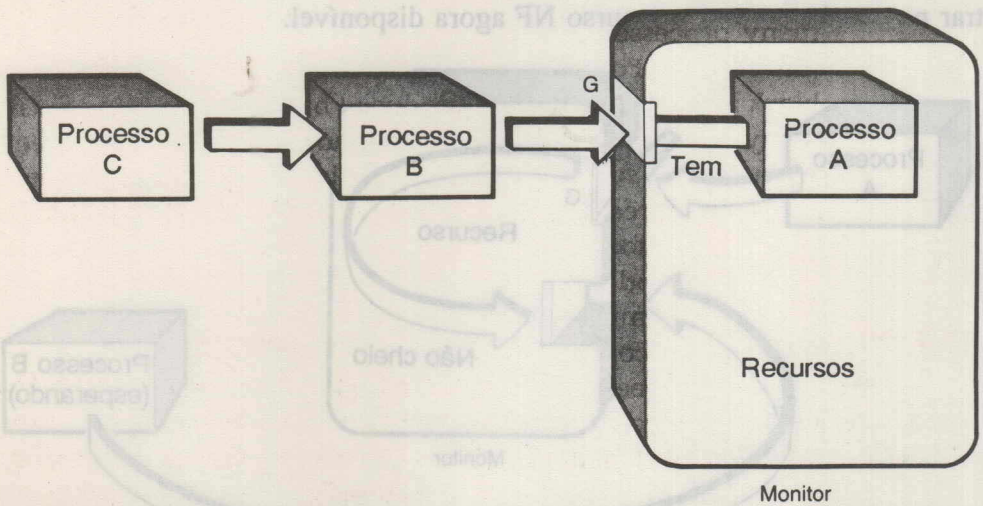


Figura 9.10 Esperando em ordem pelo semáforo.

A implementação desse capítulo permite que um leitor entre quando um escritor termina, e que um escritor entre quando o leitor termina. No entanto, para que o sistema tenha maior eficiência, quando o processo leitor acessa o buffer, ele lê todos os registros disponíveis. Assim, o leitor tem deferência sobre os escritores.

Para revisar, o exemplo de implementação alocará uma região de memória compartilhada que múltiplos escritores e um leitor podem atribuir a seu espaço de endereçamento. O exemplo usará um monitor semáforo que permite que somente um processo de cada vez possa alterar a memória compartilhada; um semáforo para "buffer não-vazio", e um semáforo para um "buffer não-çheio". A próxima

seção mostra como usar as chamadas de sistema em baixo nível do UNIX System V para implementar este projeto.

Quando você examinar este exemplo C++, tenha em mente que o uso adequado das facilidades de abstração de dados de C++ pode garantir que não ocorram condições de competição e impasse. A interface de abstração de dados não permite o acesso aos dados internos, exceto quando permitido especificamente através da interface pública. Assim, devido ao fato de C++ permitir que você argumente confidencialmente sobre as propriedades de um objeto — e até mesmo formalmente comprovar a isenção de um impasse — você pode seguramente usar a classe em outras aplicações. Se você codificasse o mesmo exemplo com estruturas tradicionais C, teria muita dificuldade para reforçar o uso de métodos de acesso cuidadosamente construídos.

Desenvolvendo o Registro de Erro da Memória Compartilhada

Esta seção contém a codificação C++ que implementa um buffer de registro de erro de memória compartilhada protegido por semáforos. O exemplo admite apenas um leitor, mas muitos escritores. As mensagens dos escritores nunca ficarão embaralhadas mesmo que todos os processos, incluindo o leitor, rodem concorrentemente. Esta codificação resolve o problema clássico produtor/consumidor apresentado em muitos livros-textos de sistemas operacionais, em que muitos processos produzem dados (escritores) e um processo os consome (o leitor). Por exemplo, um controlador de disco é um consumidor das solicitações por processos usando dados de arquivos. Entre os muitos exemplos de produtores e consumidores de dados protegidos em sistemas operacionais, este que se apresenta a seguir é simples e instrutivo.

A opção de memória compartilhada para comunicação inter-processo é a melhor solução para muitas aplicações. As FIFOs têm problemas com os delimitadores de mensagem, e as mensagens IPC estão sujeitas a limites internos de sistema, que são tipicamente muito menores do que aqueles das regiões de memória compartilhada. Além disso, as chamadas de sistema copiam mensagem

e dados de FIFO nos buffers de destino, no núcleo, enquanto, com memória compartilhada adequadamente protegida com semáforos, a leitura e escrita ocorre no espaço do usuário (não do sistema), porque os segmentos compartilhados estão mapeados em todos os espaços de dados virtuais dos processos. Em aplicações de tempo real, ou mesmo em aplicações de mutiprocessamento pesadas, a cópia extra de grande quantidade de dados pode causar uma grave perda de desempenho. O uso da memória compartilhada geralmente resulta em menos cópia em algumas situações. Por exemplo, um escritor FIFO cria uma mensagem em seu espaço de endereçamento virtual; depois executa uma chamada de sistema que copia os dados num buffer interno do sistema. Quando o leitor FIFO faz sua chamada de sistema, o dado é copiado novamente no espaço de endereçamento virtual do leitor. Usando memória compartilhada, evitam-se as duas cópias extras para e do espaço do sistema.

O programa-exemplo vem apresentar os seguintes problemas: primeiro, ele tem que obter o monitor e os semáforos de ler/escrever (permissão); depois, as funções primárias ler e escrever para permitir que um processo espere (curlurma) se o recurso não estiver disponível; finalmente, ele deve garantir que os processos cedam o monitor quando estiverem esperando por um semáforo do tipo ler (não-vazio) ou escrever (não cheio). O objetivo da abstração de dados é proporcionar estes serviços de uma maneira segura que não possa ser facilmente lograda.

A solução oferecida pelo programa-amostra demonstra a separação de assuntos, proteção (ocultamento) da implementação, e conceitos estruturais da programação objeto-orientada em C++. Como resultado disso, os programas finais de teste para ler e escrever são bastante curtos e claros. Se você alterar as implementações de classe sem alterar a interface pública, precisará apenas rearticular com as novas rotinas de implementação. A codificação de ler e escrever será exatamente a mesma.

Naturalmente, é a estrutura objeto-orientada de C++ que possibilita tal solução elegante e concisa. Você poderia programar uma solução muito semelhante usando linguagem C, mas o programa não teria a vantagem da proteção e o ocultamento dos dados privados de implementação que C++ oferece.

Iniciando o Processo de Desenvolvimento

Primeiramente, vamos fazer uma formulação precisa do problema. Processos múltiplos podem ser capazes de enviar mensagens de erro para uma região de memória compartilhada, de onde um processo leitor dedicado irá acessá-las e colocá-las num terminal. Neste caso, o leitor simplesmente imprime as mensagens no terminal.

Note que esse exemplo simples de registro de erro é a abstração de uma função de um sistema de manutenção de central telefônica. O sistema de manutenção obtém sua entrada — mensagens de erro e de advertência — proveniente de várias *switches* (chaves). Ele então envia essas mensagens a um "message logger" que procura correlações nas mensagens de erro e tem limiares de mensagem-por-unidade-de-tempo. De acordo com a gravidade das mensagens de erro e a razão com que são recebidos, o "logger" pode disparar uma campainha de alarme ou exibir uma mensagem de advertência num console especial. Mensagens importantes são guardadas em arquivos especiais. Este é um exemplo de um sistema especialista usado em manutenção de central telefônica.

Neste exemplo, os escritores também exibem suas mensagens no terminal, bem como as escrevem na memória compartilhada. As mensagens provenientes dos vários escritores não deverão ser embaralhadas na memória compartilhada e, nem tampouco, no terminal. A exibição das mensagens dos escritores no terminal, para fins de depuração, é também protegida pelo monitor! A própria tela do terminal parece oferecer limites naturais ao tamanho do buffer de registro compartilhado: as mensagens não poderão ter mais do que 80 caracteres de comprimento (80 caracteres por linha) e a memória compartilhada deverá conter até 24 delas de cada vez. Uma solução mais geral trataria a área de memória compartilhada como uma pilha e aceitaria mensagens de tamanhos variáveis; os detalhes de uma implementação assim não estão incluídos aqui.

Antes de você começar realmente a escrever códigos de programas, faça a si próprio as seguintes perguntas: "O que acontece se um escritor tentar escrever uma mensagem e a memória compartilhada estiver cheia?" "O que acontece se o leitor encontrar a área do buffer vazia?" "Como o programa responderá se dois ou mais escritores simultaneamente tentam acessar o registro quando o leitor está tentando ler?"

Outro parâmetro que precisa ser considerado é o número de mensagens escritas ou lidas imediatamente. Neste exemplo, os escritores escreverão somente uma mensagem de cada vez; o leitor lerá tantas mensagens quantas existirem no buffer quando o leitor tem acesso à memória compartilhada. Quando uma mensagem é lida, o programa mostra-a no terminal e depois a descarta. (Na prática, usando esta técnica, você talvez queira guardar as mensagens em vários arquivos ou manter alguns números, de acordo com a gravidade das mensagens.)

As 24 linhas do buffer de registro na memória compartilhada serão usadas de uma maneira circular, com o ponteiro do leitor seguindo o ponteiro do escritor na área do buffer, à medida que as mensagens são lidas e escritas. As linhas do buffer são usadas circularmente; quando a última é preenchida, a primeira linha é a próxima a ser escrita.

Uma Visão Geral dos Arquivos de Programas

O exemplo completo do buffer de registro compartilhado consiste nos arquivos que mencionaremos a seguir. O programa e os arquivos de cabeçalho formam *shlog.h,* que contém chaves de mensagens, tamanhos de linha e buffer, valores de controles de semáforos, e outras definições diversas. O arquivo *logmem.h* contém a definição de classe da memória compartilhada *logmem.* O arquivo de cabeçalho *semaset.h* contém a classe da série de semáforos *semaset.* Os arquivos *semaset.cpp* e *logmem.cpp* contêm as implementações da série de semáforos e as funções-membro de classe *log.* O conteúdo desses arquivos está dividido em várias listagens separadas para mantê-los gerenciáveis. Os arquivos *writer.cpp* e *reader. cpp* contêm o código C++ para os processos escritor e leitor. As listagens incluem também um *makefile* para gerenciar a construção dos módulos e processos. A Figura 9.11 mostra as relações entre esses arquivos.

As rotinas servidoras da série de semáforos e da memória compartilhada solicitadas pelos processos clientes leitor e escritor poderiam ter sido colocadas numa biblioteca System V de código compartilhado. Durante o decorrer de sua própria programação, você usualmente armazenará estas implementações de classe num arquivo de bibliotecas e, quando necessário, irá conectá-las aos programas clientes.

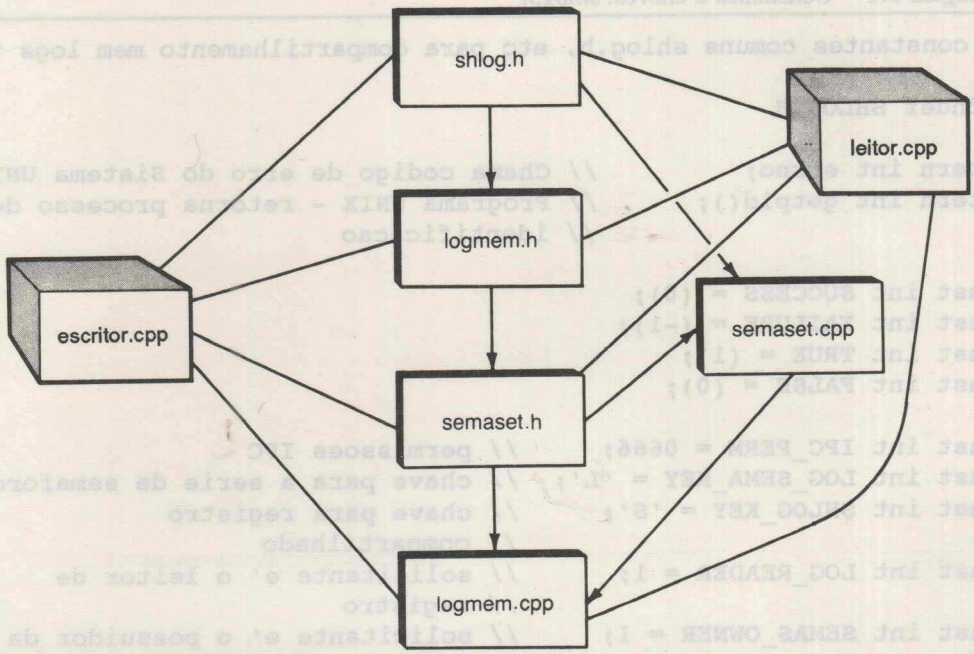


Figura 9.11 Relações de arquivos e processos.

Constantes e Valores-Chave

O arquivo de cabeçalho *shlog.h* (Listagem 9.1) contém constantes comuns, tais como *TRUE* (verdadeiro) e *FALSE* (falso), bem como parâmetros para a memória compartilhada do System V e chamadas do sistema semáforo. (Veja detalhes sobre estas chamadas de sistema no Manual de Referência do Programador UNIX.) Ele inclui também parâmetros para o tamanho do registro compartilhado, *SHBU_LINES* e *SHLINE_SIZE*, de maneira que você pode facilmente alterar seu tamanho num determinado lugar e deixar que a função *make* refaça as implementações e os procedimentos de teste. Note que a listagem usa a declaração de constante *const* de C++ em vez de C *define*. Você pode incluir nomes *const* na tabela de símbolos para uma depuração mais fácil.

Listagem 9.1 Constantes e chaves: *shlog.h*

```
/* constantes comuns shlog.h, etc para compartilhamento mem logs */

#ifndef SHLOG.H

extern int errno;          // Chama codigo de erro do Sistema UNIX
extern int getpid();       // Programa UNIX - retorna processo de
                           // identificacao

const int SUCCESS = (0);
const int FAILURE = (-1);
const int TRUE = (1);
const int FALSE = (0);

const int IPC_PERM = 0666;      // permissoes IPC
const int LOG_SEMA_KEY = 'L';   // chave para a serie de semaforos
const int SHLOG_KEY = 'S';      // chave para registro
                                // compartilhado
const int LOG_READER = 1;       // solicitante e' o leitor de
                                // registro
const int SEMAS_OWNER = 1;      // solicitante e' o possuidor da
                                // serie de semaforos
const int LOG_WRITER = 0;       // solicitante e' o escritor
const int SHBUF_LINES = 24;     // numero de linhas da memoria
                                // compartilhada
const int SHLINE_SIZE = 80;     // tamanho de linhas do buffer
const int SH_SEMAS = 3;         // necessita 3 semaforos para
                                // registro
const int SH_SEMA_OPS = 5;      // numero de opcoes do semaforo
                                // em uma chamada

const int SH_MON_SEMA = 0;      // indice do semaforo monitor
const int SH_NFULL_SEMA = 1;    // indice do semaforo de buffer
                                // nao cheio
const int SH_NEMPTY_SEMA = 2;   // indice do semaforo de buffer
                                // nao vazio
const int SH_SEMA_TRUE = 1;     // valor de inicializacao do
                                // semaforo binario
const int SH_SEMA_FALSE = 0;    // valor do semaforo binario nao
                                // disponivel

#define SHLOG_H
#endif
```

Definindo a Classe de Registro de Memória Compartilhada

Todos os dados do objeto registro de memória compartilhada são privados, e, assim, nenhum processo cliente pode acessar diretamente estes dados. Somente as rotinas reais de implementação podem acessar tais dados e usar as funções-membro internas. Assim, C++ facilita muito a abstação de dados e a proteção de informação, a única coisa a mais é a palavra *public:*. A declaração das funções-membro inclui também os modelos de lista de argumentos (embora a maioria dessas rotinas não usa argumentos). A Listagem 9.2 contém a definição de classe para o registro de memória compartilhada.

Listagem 9.2 A definição de classe do arquivo registro de memória compartilhada: *logmem.h*

```
/* logmem.h classe para arquivo de registro de memoria compartilhada */

#ifndef LOGMEM.H

class logmem {
  struct shmem {
    short readidx;       // le o indice da linha de buffer
    short writeidx;      // escreve o indice na linha de buffer
    char buff[SHBUF_LINES] [SHLINE_SIZE];  // buffer de linhas
                                           // para registro
  } * shmem;             // endereco de memoria ligado a

  semaset * semas;       // serie de semaforos para o monitor
  int id;                // identificacao da area de memoria
                         // compartilhada
  key_t key;             // chave usada para conseguir a area
  int owner;             // proprietario/criador da memoria
                         // compartilhada
  int blines;            // numero total de linhas do buffer
  int bsize;             // numero de bytes das linhas do buffer

  int remove();          // remove regiao de memoria compartilhada
  int markfull();        // afixa buffer cheio (escritor)
  int marknempty();      // afixa buffer nao vazio (escritor)
  int marknfull();       // afixa buffer nao cheio (leitor)
  int markempty();       // afixa buffer vazio (leitor)
  int empty();           // retorna qual buffer esta vazio

public:
  logmem(int flag);      // construtor; flag para o leitor
```

```
  ~logmem();              // destrutor (memoria liberada)

  int detach();           // separa da regiao compartilhada

  int monnfull();         // um momento, nao cheio (escritor)
  int monnempty();        // um momento, nao vazio (leitor)
  int postmon();          // saida do monitor

  int read(char * line);     // le registro; adv le ptr
  int write(char * line);    // escreve registro; adv escreve ptr

  void dump();            // descarrega valores para depuracao

};

#define LOGMEM_H
#endif
```

A estrutura *shmem* tem apenas um ponteiro declarado para ela. Por se tratar de dados compartilhados, esta área deve ser alocada apenas uma vez e compartilhada por todos os usuários. Da mesma maneira, o programa usa um ponteiro para a série de semáforos nos dados *logmem*. A série é alocada separadamente apenas para demonstrar diversidade.

Na teoria de sistemas operacionais, o problema de manter um processo fora de uma parte do código ou de uma região de dados enquanto um outro processo executa uma atualização crítica é chamado de "problema de exclusão mútua". No entanto, embora não possa ser permitido entrar mais de um cliente de cada vez, todos os processos que desejem entrar eventualmente serão servidos. Para resolver o problema da exclusão mútua, trace mentalmente uma parede ao redor da área de memória compartilhada. Você precisa limitar o acesso a essa área especial através de um monitor, que guardará a entrada permitindo que apenas um processo de cada vez possa entrar para alterar a memória compartilhada. Este monitor é representado por um semáforo monitor.

A área de memória compartilhada do exemplo está protegida por um monitor que controla estritamente entrada e saída. A porta do monitor é um semáforo: quando ela está para baixo (tem um valor positivo), um processo pode entrar na região monitorada para executar uma função. Quando ela está para cima (tem um valor zero ou negativo), um único processo está usando o recurso e outros processos têm que esperar numa fila até que o recurso esteja disponível.

O que acontece se o leitor tenta entrar no monitor mas sem haver dados lá? Se ele espera dentro do monitor, então nenhum escritor pode entrar, criando assim um "soterramento". Outro semáforo, o semáforo não-vazio, resolve isso, permitindo que o leitor espere fora do monitor.

O que acontece se um leitor que acessar e o buffer está cheio? Outro semáforo, este agora especificando não-cheio, permite que escritores esperem por espaço de buffer disponível fora da região monitorada.

Finalmente, se um escritor escreve uma linha que torna o buffer não-vazio, ele deve ativar o leitor, afixando ou incrementando o semáforo não-vazio. O leitor deve se comportar de maneira semelhante quando ocorrer a transição não-cheio.

A Figura 9.12 mostra a aparência final da região de memória compartilhada com seu monitor, semáforo e linhas de espera de semáforo.

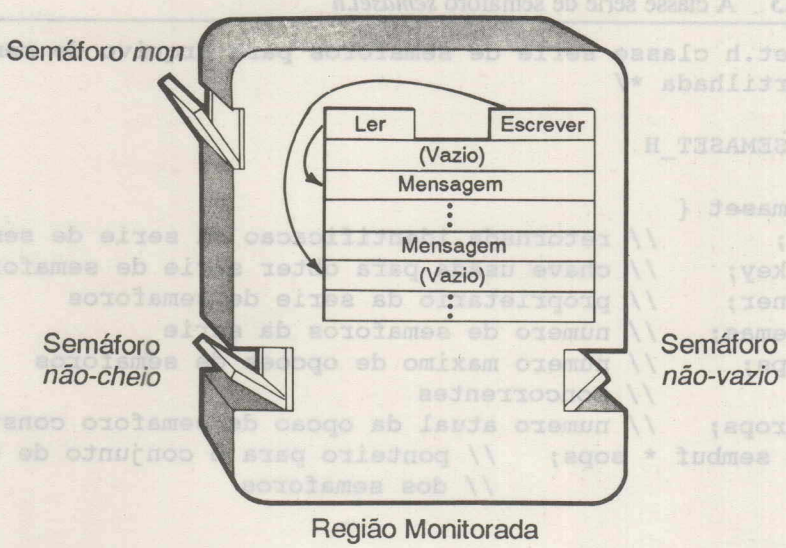


Figura 9.12 Monitores de memória, semáforos e filas.

Agora, usaremos as chamadas de sistema System V para codificar as rotinas de acesso.

Usando os Semáforos: a Classe semaset

A classe *semaset* é definida de uma maneira geral, e, portanto, pode ser usada com aplicações diferentes do registro de memória compartilhada. Por esse motivo, a série de operações está alocada na função construtor, e somente o ponteiro *sops* aparece nos dados. O número de semáforos na série e o número de operações (essencialmente, o número de "coisas do semáforo" feitas numa chamada atômica de sistema, como "toma o monitor, libera o monitor, toma não-vazio, libera não vazio") são argumentos para o construtor, semaset(). Várias funções-membro e operações semáforo usam valores assumidos para os últimos argumentos, de maneira que a série semáforo pode proporcionar semáforos de contagem. (Observe, no entanto, que esse exemplo usa somente versões binárias desses semáforos.) A Listagem 9.3 contém as declarações de semáforo.

Listagem 9.3 A classe série de semáforo *semaset.h*

```
/* semaset.h classe serie de semaforos para arquivo de memoria */
/* compartilhada */

#ifndef SEMASET_H

class semaset {
  int id;         // retornada identificacao da serie de semaforos
  key_t key;      // chave usada para obter serie de semaforos
  int owner;      // proprietario da serie de semaforos
  int nsemas;     // numero de semaforos da serie
  int nops;       // numero maximo de opcoes de semaforos
                  // concorrentes
  int curops;     // numero atual da opcao de semaforo construida
  struct sembuf * sops;    // ponteiro para o conjunto de opcoes
                           // dos semaforos

public:
  semaset(int ckey, int numsems, int maxops, int flag);
                                                  // construtor
  ~semaset();                 // destrutor
  int init(int semnum, int resources); // recursos e' o valor
                                       // inicial
  int remove();                        // remove identificacao do
                                    //  sistema serie de semaforos

  int post(int sema, int val=1);     // indica liberacao de recurso
```

```
  int wait(int sema, int val=1);      // obtem o recurso; espera
                                      // ate estar disponivel
  int wait0(int sema);                // espera que o semaforo se
                                      // torne 0
  int doset0(int sema);               // limpa o semaforo (zera-o)
  int is0(int sema);                  // o valor do semaforo e' 0?
  int isp(int sema);                  // o valor do semaforo e'
                                      // positivo?
  int dosema();                       // executa semaop() do vetor
                                      // de opcoes preenchidas
  int clear();                        // limpa o vetor de opcoes
                                      // (encerra ou cancela ops)

  void dump();          // descarrega os valores de semaforo para
                        // depuracao

};

#define SEMASET_H
#endif
```

As Operações Semáforo

A seguir, estão as declarações de uma série de três semáforos e um grupo de cinco operações semáforo. O conjunto de operações semáforos é o método usado pelo System V para ligar operações semáforo de uma maneira *atômica*, de modo que nenhum outro processo possa acessar um recurso e invalidar uma operação anterior na série se uma operação posterior na série está sendo esperada. Evita-se também o impasse.

Esperando pelas Condições "não-cheio" e "não-vazio"

A codificação que permite a um escritor acessar o buffer pode ser a mais confusa do exemplo das operações semáforo. Para passar pela porta do monitor e

escrever uma linha de registro no buffer da memória compartilhada, devem ser executadas as seguintes operações de *logmem.cpp* (Listagem 9.4).

Listagem 9.4 Entrar no monitor quando o buffer não está cheio: *monnfull()*

```
// usado pelo escritor para ter acesso ao buffer

int logmem :: monnfull()
{
  if (id == 0)
    return FAILURE;

  // primeiro, obtem o monitor de memoria compartilhada
  // aguarda ate que esteja disponivel, entao toma-o
  semas->wait(SH_MON_SEMA);

  // libera o monitor, caso voce tiver que dormir
  // se o buffer estiver cheio!
  // desiste do semaforo do monitor
  semas->post(SH_MON_SEMA);

  // em seguida espera ate que o buffer esteja nao cheio,
  // de forma que voce tenha espaco para escrever
  // espera ate que esteja nao cheio
  semas->wait(SH_NFULL_SEMA);

  /* entao, libera o semaforo nao cheio, de maneira a nao */
  /* trancar o leitor do lado de fora */
  semas->post(SH_NFULL_SEMA);    // espera ate a condicao nao cheio

  /* quando voce puder trabalhar, pegue o monitor novamente para
exclusao */
  semas->wait(SH_NOM_SEMA);      // espera ate estar disponivel

  // agora que esta construido o vetor de operacoes atomicas,
  // executa as operacoes para o semaforo
  int ret;
  if ((ret = semas->dosema()) == FAILURE)
    return FAILURE;

  /* limpa o buffer do vetor de operacoes
    semas->clear();

  return SUCCESS;
}
```

Vamos discutir esse programa em detalhe. Em primeiro lugar, uma discussão desse tipo não aparece nos manuais do System V, e a seção que trata dos semáforos, no Guia do Programador, apenas apresenta um programa para testar todas as opções possíveis de operações semáforo. Embora o programa seja útil, ele não explica por que os semáforos são manipulados daquela maneira em particular. (Note que o Manual de Referência do Programador tem um erro na seqüência de chamada que é repetido no arquivo *sys/sem.h*, o qual é fornecido juntamente com a versão UNIX de C++!)

O uso da série de semáforos é melhor explicado em termos dos princípios de concorrência da seção anterior. Em primeiro lugar, um leitor deve esperar pelo semáforo do monitor; esperar antes por outro semáforo pode causar impasse. Esta é a primeira operação *semas-wait* (*SH_MON_SEMA*) da listagem. Quando o leitor entra no *monitor*, ele libera o semáforo do monitor, de modo que, se tiver que esperar pela condição *buffer não-vazio*, esperará "fora" da região crítica e permitirá que outro processo acesse o monitor. Isso evita o soterramento. Quando é afixado o semáforo *não-vazio*, o leitor retoma a execução e readquire do monitor de modo a poder executar seu trabalho. No entanto, primeiramente ele afixa novamente o semáforo *não-vazio*, porque aquele semáforo é binário — significando "não-vazio é verdadeiro" — e precisa ser incrementado para mostrar seu valor verdadeiro. Assim, se o processo tem que esperar por *mon* (monitor) após ter esperado por *não-vazio*, o semáforo *não-vazio* estará correto enquanto o leitor espera.

Na realidade, as chamadas *wait()* e *post()* simplesmente registram suas operações no conjunto de operações semáforo, de maneira que a chamada de sistema em *semas->dosema()* as executa conforme descrito acima. Após completar a chamada de sistema, a função *semas->clear*() reseta o conjunto de operações zerando a contagem de operações válidas.

Você deve usar o conjunto de semáforos, em vez de operações semáforo simples através de chamadas do sistema, porque a cadeia de chamadas deve ser atômica. Por exemplo, se um escritor não pede o semáforo do monitor enquanto verifica a condição semáforo *não-cheio*, então outro processo escritor poderá tomar o monitor e invalidar a condição (escrevendo a última parte do buffer) antes que o primeiro escritor reclame o monitor. Por outro lado, um escritor deve ceder o semáforo do monitor antes de verificar a condição semáforo *não-cheio*, porque se esta for falsa, o escritor deve esperar até que um leitor o afixe. O conjunto de semáforos e a chamada única de sistema permitem que o escritor renuncie

transitoriamente ao monitor enquanto verifica o semáforo *não-cheio* — para o caso em que ele tenha que esperar. Se o escritor não tiver que esperar, então ele pode retomar o monitor, sabendo que os semáforos que ele acaba de verificar não foram invalidados por outro escritor.

A rotina de acesso "monitor e buffer não-vazio" (Listagem 9.5) é provavelmente também de *logmem-cpp*.

Listagem 9.5 Entrar no monitor quando o buffer estiver não-vazio: *monnempty()*

```
// usado pelo leitor para ter acesso ao buffer

int logmem :: monnempty()
{
  if (id == 0)
    return FAILURE;

  // primeiro, toma o monitor de memoria compartilhada
  // espera ate que esteja disponivel, entao toma-o
  semas->wait(SH_MON_SEMA);

  // libera o monitor, caso voce tiver que dormir
  // enquanto o buffer estiver cheio!
  semas->post(SH_MON_SEMA);     // desiste do semaforo do monitor

  // em seguida espera ate que o buffer esteja nao vazio,
  // de forma que voce tenha alguma coisa para ler
  semas->wait(SH_NEMPTY_SEMA); // espera ate que esteja nao vazio

  /* libera o semaforo nao vazio, de maneira que os */
  /* escritores nao fiquem trancados do lado de fora */
  semas->post(SH_NEMPTY_SEMA);    // libera semaforo nao vazio

  /* quando voce puder trabalhar, pegue o monitor novamente para */
  /* exclusao */
  semas->wait(SH_NOM_SEMA);       // espera ate estar disponivel

  // agora que esta construido o vetor de operacoes atomicas,
  // executa as operacoes para o semaforo
  int ret;
  if ((ret = semas->dosema()) == FAILURE)
    return FAILURE;

  /* limpa o buffer do vetor de operacoes
    semas->clear();
```

```
  return SUCCESS;

}
```

Note que a rotina acima é muito semelhante à rotina *monnfull()* descrita anteriormente.

Funções-Membro Semáforo wait() e post()

Vamos examinar agora a codificação para *wait()* e *post()*, a partir do módulo de implementação da série semáforo, *semaset.cpp* (veja Listagem 9.6).

Listagem 9.6 As funções semáforo *wait()* e *post()*

```
/* val e' o numero de afixacoes (recursos livres); valor assumido
e 1 */
int semaset :: post(int sema, int val=1)
{
  if (id == 0)
    return FAILURE;

  if (val <= 0)    // se zero ou negativo, proibe
    return FAILURE;

  if (curops >= nops)
    return FAILURE;

  sops[curops].sem_num = sema;      // indice do vetor de semaforos
  sops[curops].sem_op = val;        // afixa recursos
  sops[curops++].sem_flg = 0;       // nada para desfazer

  return curops;          // retorna opcoes no vetor, ate aqui
}
int semaset :: wait(int sema, int val=1)
{
  if (id == 0)
    return FAILURE;

  if (val <= 0)  // se zero ou negativo, proibe
```

```
    return FAILURE;

  if (curops >= nops)
    return FAILURE;

  sops[curops].sem_num = sema;        // indice do vetor de semaforos
  sops[curops].sem_op = -1;           // obtem o recurso; senao dorme
  sops[curops++].sem_flg = SEM_UNDO;   // se extinguir-se, afixa-o

  return curops;    // retorna numero do opcoes no vetor, ate aqui
}
```

Note que as rotinas *semaset::wait()* e *semaset:: post()* simplesmente colocam um elemento no conjunto de operações "atômicas" que a chamada de sistema *semop()* executa quando chamada a partir de *semaset::dosema()*. Isso, novamente, garante que uma operação semáforo não possa ser "desfeita" por outro processo antes que a próxima operação semáforo seja completada. A noção de série e conjunto de semáforos funciona porque o UNIX é um núcleo de "linha única sem direito de opção". Naturalmente, se um processo deve esperar que o recurso de um semáforo se torne disponível, o núcleo o desativa até que chegue sua vez novamente na fila de espera do semáforo. É por isso que o programa afixa *mon* antes de experar por *não-vazio* ou *não-cheio.*

O Que Fazer com "UNDO"

O uso do sinalizador (flag) *SEM_UNDO* (desfaz operação semáforo) necessita de alguma explicação. Se um processo cessa (sai) enquanto está esperando por um recurso (ou, por exemplo, é interrompido pelo alarme de um temporizador), interrompendo assim a chamada de sistema *semop*, o UNIX mantém o registro do semáforo pelo qual o processo está esperando. O sistema operacional então afixa o semáforo se o processo é tirado da fila de espera do semáforo. *Não* desfaça (undo) uma operação de afixação, embora seja permitido fazer isso na chamada de sistema, porque o buffer não se torna *não-vazio* se o escritor do último registro que foi escrito sai completamente das filas de espera, do monitor e do núcleo do sistema.

Isso pode não ser óbvio. Minha primeira implementação de membros semáforos tinha um "undo" na afixação de chamadas, mas, quando saiu o último

escritor, saiu também a afixação do semáforo *não-vazio* e o leitor não conseguiu ler os últimos registros escritor no buffer! No entanto, se um semáforo de contagem representa o número de processos "servidores" disponíveis para executar algum serviço, então você precisa desfazer (undo) suas afixações quando aqueles processos cessam ou são interrompidos e indisponíveis. Anule (undo) somente quando a solicitação por um recurso (espera) ou disponibilidade do recurso (afixação) não deve mais permanecer efetiva se desaparece quem espera (waiter) ou quem afixa (poster).

A Chamada de Sistema da Série Semáforo

A rotina *dosema()* que, na realidade, executa a chamada de sistema e a rotina *clear()* são implementadas conforme ilustrado na Listagem 9.7 (por *semaset.cpp*).

Listagem 9.7 Operações semáforo atômicas: *dosema()* e *clear()*

```
int semaset :: dosema()
{
  if (id == 0)
    return FAILURE;
  /* operacoes podem ser montadas; caso retorno bem sucedido, */
  /* entao tudo estava salvo */
  /* verificado que todo o vetor de operacoes esta "atomicamente" */
  /*  em ordem no nucleo */
  if (semop(id, sops, curops) == FAILURE)
    return(FAILURE);

  return SUCCESS;
}

int semaset :: clear()
{
  if (id == 0)
    return FAILURE;

  int ret = curops;
  curops = 0;          // nao ha operacoes correntes; apaga tudo
```

```
  return ret;          // retorna o numero de operacoes emanadas
}
```

Note que a rotina *dosema()* utiliza o seguinte argumento:

```
struct sembuf * sops;
```

Este ponteiro para um conjunto de operações semáforo é o argumento, da maneira como ele é usado no programa de teste de semáforo do System V Programmer's Guide (Guia de Programação do System V). No entanto, a discussão naquela parte e a declaração de argumento no Manual de Referência do Programador do System V pede que seja colocado assim:

```
struct sembuff**sops;
```

É claro que isso está errado. A expressão ** representa "um ponteiro de um ponteiro de um valor", enquanto um único * representa "um ponteiro para um valor", ou um referenciamento simples, e não duplo. Esse erro se estende também ao modelo da função *sys/sem.h*. Presume-se que isso seja corrigido numa próxima versão dos manuais.

A função *clear()* simplesmente zera a contagem de operações armazenada no conjunto de operações semáforo. Essa função é executada antes do retorno da chamada de sistema de semáforo, e o conjunto de operações deve ser zerado para o próximo uso.

Deixando o Monitor

Quando um processo termina sua tarefa dentro da região monitorada, ele afixa o semáforo do monitor para liberar o recurso para outros processos (veja Listagem 9.8, de *semaset.cpp*).

Listagem 9.8 Saída do monitor: *postmon()*

```
int logmem :: postmon()      // libera o monitor de memoria
                             // compartilhada
{
  if (id == 0)
```

```
    return FAILURE;

  /* assume que voce tem o monitor de memoria compartilhada */

  /* afixa o monitor de semaforo para outros processos de */
  /* memoria  compartilhada */
  semas->post(SH_MON_SEMA);     // afixa monitor nao em uso

  /* agora que o vetor de operacao(oes) atomica(s) esta */
  /* construido, */
  /* executa as operacoes de semaforo */
  int ret;
  if ((ret = semas->dosema()) == FAILURE)
    return FAILURE;

  /* limpa o buffer do vetor de operacoes */
  semas->clear();

  return SUCCESS;

}
```

Essa chamada simples coloca apenas uma operação no conjunto; a solicitação para afixar o semáforo do monitor, *sema->(SH_MON_SEMA)*. A chamada *dosema()* faz então apenas uma operação semáforo — a afixada — que não pode esperar.

Escrevendo e Lendo Mensagens

Agora, vamos ver as funções-membros para ler e escrever no registro de memória compartilhada.

Escrevendo Mensagens

O ponteiro do leitor sempre aponta para a próxima linha a ser lida no buffer; o ponteiro do escritor aponta para a próxima localização disponível para escrever uma linha. Se os ponteiros de ler e escrever foram iguais, ou o buffer está cheio, ou está vazio. Os "ponteiros" ler e escrever são, na realidade, índices, porque processos diferentes podem ter endereços virtuais diferentes para o buffer da memória compartilhada. Se um escritor usou um ponteiro efetivo na memória compartilhada com seu endereço (virtual), então um leitor, que poderia atribuir a memória compartilhada a uma localização diferente, não seria capaz de usar aquele ponteiro corretamente. Se o buffer estiver vazio, o programa zera os ponteiros (índices) de ler e escrever.

Se o escritor fizer com que os índices de buffer sejam iguais, então o buffer está cheio, e, como tal, é assinalado. Quando o escritor termina, o buffer deve estar não-vazio, de maneira que o escritor fixa aquele semáforo. A Listagem 9.9 mostra a codificação para a função *write()*.

Listagem 9.9 A função escritor do buffer de registro compartilhado: *write()*

```
// escreve uma linha no buffer
int logmem :: write(char * line)
{
  /* a linha deve apontar para um buffer que seja grande */
  /* o suficiente */
  /* para conte-la */
  if ((id == 0) || (line == (char *)NULL))
    return FAILURE;

  /* nao pode estar cheio, porque voce escreveu um de cada vez */
  /* e voce esperou exatamente pela condicao nao cheio */

  /* ou entao copia a linha do solicitante na linha do buffer */
  /* de memoria compartilhada */

  int to = shmem->writeidx;
  strncpy(shmem->buff[to], line, SHLINE_SIZE);

  if ((shmem->writeidx = ++shmem->writeidx % SHBUF_LINES)
      == shmem->readidx)
    {
```

```
        /* entao buffer esta cheio */
        markfull();          // assinala-o como cheio
    }

    marknempty();            // agora, buffer nao vazio

    return SUCCESS;
}
```

Quando o escritor está executando a função *write()*, o buffer não pode estar cheio porque o escritor esperou pelo semáforo não-cheio, está de posse do semáforo do monitor e, portanto, é o único processo na região monitorada. As operações semáforo para o escritor poderiam ser incluídas na função *write()*, mas elas não são combinadas, de maneira que a estrutura de programa da função escritor se assemelha mais estreitamente à estrutura de programa do leitor. Após ser escrito um registro, o buffer é assinalado como cheio se os índices de ler e escrever forem iguais agora. O buffer é assinalado também como sendo *não-vazio*, atribuindo o valor 1 ao semáforo *não-vazio*. Isso é uma "marca" e não uma "afixação" porque o semáforo é um semáforo binário. Se cada escritor bem-sucedido afixou o semáforo *não-vazio*, seu valor se tornaria maior do que um, e o semáforo se tornaria um semáforo de contagem em vez de um semáforo binário (que só tem os valores zero e um). Se você usar semáforos de contagem (como está sugerido no Exercício 9.3), então você pode usar a função "post" (afixar) para indicar o número de registros escritos. O valor do semáforo representaria, então, o número de recursos (registros) disponíveis (esperando para serem lidos).

Note que o programa utiliza a função *strncpy()*, de maneira que a cópia é "segura" (uma longa linha enviada à rotina de escrever não pode ser escrita em cima de parte da linha seguinte). Sempre que for possível, utilize *strncpy()* e não *strcpy()*. A função *strncpy()* permite que você limite o número de caracteres que podem ser copiados. Se não for encontrado nenhum caractere nulo *end-of-string* (fim-de-cadeia) antes de alcançar o limite, a cópia se encerra. Com a função *strcpy()*, um ponteiro inválido (por exemplo, usando ** em vez de *) pode copiar centenas de caracteres antes de encontrar um caractere nulo para pará-lo. Esse erro é encontrado freqüentemente com argumentos de string inválidos para a função *printf()* em C.

Lendo Mensagens

A rotina *read()* (Listagem 9.10) é diferente da rotina *write()*. Embora somente um registro de cada vez possa ser escrito na memória compartilhada, o leitor pode ler todas as mensagens no buffer. Assim, ele executa uma verificação extra (outra chamada de sistema) para ver se ainda há algum outro registro a ser lido. Note que o leitor não pode esperar pela condição *não-vazio* porque ele está de posse do monitor.

Listagem 9.10 A função leitor do buffer de registro compartilhado: *read()*

```
/* le uma linha do buffer

int logmem :: read(char * line)
{
  /* somente o criador pode ser leitor/consumidor */
  if (owner != getpid())
    return FAILURE;

  /* a linha deve apontar para um buffer suficientemente grande */
  /* para conte-la */
  if ((id == 0) || (line == (char *)NULL))
    return FAILURE;

  /* precisa testar, por causa de multiplas leituras no monitor */
  if (empty())
    return FAILURE;

  /* ou copiar um registro na linha do solicitante */
  int from = shmem->readidx;
  strncpy(line, shmem->buff[from], SHLINE_SIZE);

  if ((shmem->readidx = ++shmem->readidx % SHBUF_LINES)
      == shmem->writeidx)
    {
      /* entao buffer esta vazio */
      markempty();          // assinala-o como vazio
    }

  marknfull();              // agora, buffer nao cheio

  return SUCCESS;
}
```

Note que pode existir somente um leitor, a memória compartilhada e o criador da série de semáforos. Para permitir múltiplos leitores, você teria que alterar os construtores, porque o programa usa a simplificação de que o único leitor faz a alocação real de memória para a memória compartilhada e cria a série de semáforos.

Algumas Propostas de Projeto

Após ser lido o último registro, o programa usa a convenção de zerar ambos os índices (de ler e escrever). Essa providência não é necessária e nenhuma função depende dela. Ela simplesmente torna a depuração mais fácil.

As rotinas ler e escrever são as verdadeiras rotinas "monitor", no sentido de que são as regiões críticas da codificação em que múltiplos processos não devem entrar ao mesmo tempo. O semáforo monitor é o dispositivo do System V que exclui a possibilidade de que mais do que um processo ocupe aquelas regiões críticas. Os semáforos *não-vazio* e *não-cheio* e suas filas de espera representam as condições que podem ser sinalizadas e aguardadas (fora do monitor) no paradigma clássico do monitor. O conjunto das operações semáforo usado com a série de semáforos representa a espera condicional característica de um monitor clássico.

As rotinas de ler e escrever contêm as operações semáforo de entrada e saída meramente para fins de interpretação. Para ser fiel ao modelo do monitor e aos princípios de ocultamento de informação você pode usar as operações *read()* e *write()* somente ao nível de cliente/usuário e ocultar completamente toda espera, afixação e sinalização do semáforo dentro da classe *logmen.*

As rotinas semáforo "mark" (sinalização) meramente definem os semáforos binários em zero ou um (veja Listagem 9.11, de *logmem.cpp*).

Listagem 9.11 Rotinas de "marca" para semáforo cheio ou vazio: *markfull(), markempty(), marknful()* e *marknempty()*

```
// usado pelo escritor para indicar buffer cheio

int logmem :: markfull()
{
  if (id == 0)
    return FAILURE;
```

```
  /* assume que tem o monitor de memoria compartilhada */

  /* afixa buffer cheio */
  /* fazendo "semaforo nao cheio" falso
  semas->doset0(SH_NFULL_SEMA);

  /* set0() nao usa semaop()  */

  return SUCCESS;
}

// usado pelo leitor para indicar buffer vazio

int logmem :: markempty()
{
  if (id == 0)
    return FAILURE;

  /* assume que tem o monitor de memoria compartilhada */

  /* afixa buffer vazio */
  /* fazendo "semaforo nao vazio" falso
  semas->doset0(SH_NEMPTY_SEMA);

  /* set0() nao usa semaop()  */

  return SUCCESS;
}

// usado pelo leitor para indicar buffer nao cheio

int logmem :: marknfull()
{
  if (id == 0)
    return FAILURE;

  /* assume que tem o monitor de memoria compartilhada */

  /* se o semaforo ainda e' 1, entao apenas retorne */
  if (semas->isp(SH_NFULL_SEMA))
    return SUCCESS;

  /* afixa buffer nao cheio */
  semas->post(SH_NFULL_SEMA);
```

```
  /* agora que o vetor de operacao(oes) atomica(s) esta pronto, */
  /* executa as operacoes de semaforo */
  int ret;
  if ((ret = semas->dosema()) == FAILURE)
    return FAILURE;

  /* limpa o buffer do vetor de operacoes */
  semas->clear();

  return SUCCESS;
}

// usado pelo escritor para indicar buffer nao vazio

int logmem :: marknempty()
{
  if (id == 0)
    return FAILURE;

  /* assume que tem o monitor de memoria compartilhada */

  /* se o semaforo ainda e' 1, entao apenas retorne */
  if (semas->isp(SH_NEMPTY_SEMA))
    return SUCCESS;

  /* afixa buffer nao vazio */
  semas->post(SH_NEMPTY_SEMA);

  /* agora que o vetor de operacao(oes) atomica(s) esta pronto, */
  /* executa as operacoes de semaforo */
  int ret;
  if ((ret = semas->dosema()) == FAILURE)
    return FAILURE;

  /* limpa o buffer do vetor de operacoes */
  semas->clear();

  return SUCCESS;
}
```

As rotinas semáforo do System V proporcionam uma função para zerar um semáforo. No entanto, o programa não usa a função normal *wait()* (esperar) porque ele quer *forçar* o semáforo a ser zero, e não *esperar* até que ele seja zero!

A inicialização de um semáforo não é feita através da chamada de sistema *semaop()*. Semáforos são inicializados usando-se a chamada de sistema *semctl()*, como está ilustrado na rotina *semaset.doset0()* discutida mais adiante.

Talvez você queira refletir sobre a implementação de uma versão diferente da classe *semaset*, na qual o semáforo *não-vazio* é um semáforo contador que indica o número de mensagens do buffer que não foram lidas, e o semáforo *não-cheio* é uma contagem do número de linhas vazias. Embora isso possa tornar desnecessária a função *empty()* (vazio) na rotina de verificação do leitor, você deve ter cuidado para que o leitor não fique esperando dentro do monitor quando o semáforo *não-cheio* vai para zero, como poderia acontecer se você simplesmente codificasse uma chamada *wait()* (espera). Se o leitor permanecer esperando enquanto ele está de posse do semáforo do monitor, você terá uma condição "livelock" (soterrado).

Se você alterar *semaset*, não precisa alterar *logmem*, embora possa querer alterá-la para tirar proveito dos semáforos de contagem, talvez para maior eficiência ou melhor verificação de erro. Você também não precisa alterar os processos leitor e escritor. Isso é tudo sobre abstração de dados e programação objeto-orientada!

Construtores e Destrutores

O construtor para a série semáforo é imediato (veja Listagem 9.12). Após algumas verificações simples de segurança (que poderiam ser intensificadas), a chamada de sistema semgt() pega a identificação (ID) para uma série semáforo (veja Listagem 9.12). Devido a todas as rotinas usarem o mesmo valor-chave, elas compartilham a série semáforo. Além disso, cada processo tem seu próprio buffer de operações de conjunto de semáforo, que o construtor aloca usando o operador C++ new. (O destrutor usa o operador delete para limpar esses buffers.) Novamente, o programa aloca a série semáforo dessa maneira para demonstrar mais claramente o uso de construtores e destrutores. Ele poderia ter definido o membro semas sem

o ponteiro e deixar que o compilador automaticamente manipulasse as operações new e delete.

Listagem 9.12 Construtor e Destrutor *semaset*: *semaset()* e *~semaset()*

```
/* semaset.c implementa rotinas para a serie de semaforos */

#include <stream.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "shlog.h"
#include "semaset.h"

semaset :: semaset(int ckey, int numsems, int maxops, int flag)
{
  id = key = owner = nsemas = nops = curops = 0;
  sops = (struct sembuf *)NULL;
  errno = 0;          // apaga o codigo de erro do sistema UNIX

  if ((numsems <= 0) || (maxops <= 0))
    return;

  // obtem o vetor sembuf de armazenamento livre
  sops = new sembuf[nops];
}

semaset :: ~semaset()
{
  if (id == 0)
    return;

  delete[nops] sops;    // retorna o vetor sembuf para
                        // armazenamento livre
}
```

A chamada de sistema *semget()* retorna uma série de semáforos (do núcleo) ID, de modo que todos os usuários da série usam os mesmos semáforos. No entanto, cada usuário da série semáforo tem que ter seu próprio conjunto de operações semáforo, porque vários processos podem simultaneamente estar esperando, afixando ou "marcando" diferentes semáforos.

A Listagem 9.13 mostra como *init()* (a função leitor/criador) inicializa os semáforos. Quando todos os processos terminam de usar os semáforos, a função *remove()* os remove do sistema.

Listagem 9.13 Inicializando e removendo as funções semáforo: *init()* e *remove()*

```
int semaset :: init(int sema, int resources)
{
  errno = 0;      // inicializa codigo de erros do sistema

  if (id == 0)
    return FAILURE;

  if (owner != getpid())
    return FAILURE;

  if ((sema < 0) || (sema > (nsemas - 1)))
    return FAILURE;      // certifica-se de que o numero do
                         // semaforo esta na serie

  if (resources < 0)     // se negativo, proibe
    return FAILURE;

  union senum arg; // argumento val inicial de semaset nao e usual
  arg.val = resources;

  // inicializa o semaforo na serie com valor inicial desejado
  if (semctl(id, sema, SETVAL, arg) == FAILURE)
    return FAILURE;      // errno e' estabelecido pelo sistema

  return SUCCESS;
}


int semaset :: remove()
{
  errno = 0;      // apaga o codigo de erro de chamada do sistema
UNIX
  if (id == 0)
    return FAILURE;

  if (owner != getpid())
    return SUCCESS;
```

```
  union senum arg;          // argumento para remocao da serie
                            // semaforo e' 0
  arg.val = 0;

  /* remove a serie semaforo do sistema */
  if (semctl(id, nsemas, IPC_RMID, arg) == FAILURE)
    return FAILURE;         // errno e' estabelecido pelo sistema

  id = key = owner = 0;

  return SUCCESS;
}
```

A chamada de sistema *semctl()* do System V inicializa os valores semáforo. Outra chamada *semctl()* também remove a série semáforo ID, que efetivamente o remove do núcleo. Somente um processo deverá executar estas operações — o leitor. (Você precisa alterar isso para permitir múltiplos leitores.)

A função construtor *logmem* (Listagem 9.14) é mais complicada, porque contém o ponteiro da região de memória compartilhada e a série de semáforos. Observe que *logmem* contém também uma instância da classe *semaset* , em vez de ser derivada daquela classe. Portanto, esse exemplo segue a prática da "decomposição de cima para baixo" em vez do paradigma hereditário totalmente objeto-orientado. No entanto, você poderia derivar regiões de memória compartilhada de finalidade especial ou séries de semáforos, a partir destes exemplos, seguindo o modelo da hereditariedade. Estas são mais simples de apresentar.

Listagem 9.14 Construtor e destrutor *logmem*: *logmem()* e *~logmem()*

```
/* logmem.c implementa rotinas para arquivo de memoria
compartilhada */

#include <stream.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shlog.h"
#include "semaset.h"
#include "logmem.h"

logmem :: logmem(int flag)
{
```

```
  id = key = owner = blines = bsize = 0;
  shmem = (struct shmem *)NULL;
  semas = (semaset *)NULL;

  errno = 0;
  int bytes = (SHBUF_LINES * SHLINE_SIZE) + (2*sizeof(int));

  /* toma a regiao de memoria compartilhada */
  /* somente o leitor pode criar memoria compartilhada */
  if ((flag == LOG_READER)
      && ((id = shmget(SHLOG_KEY, bytes,
      (IPC_CREAT    IPC_PERM))) == FAILURE)
    return;          // errno e' estabelecido pelo sistema
  else if ((id = shmget(SHLOG_KEY, bytes, IPC_PERM)) == FAILURE)
    return;          // errno e' estabelecido pelo sistema

  shmem = new shmem;

  if ((shmem = (shmem *)shmat(id, 0, 0)) == (shmem *)(-1))
    {
    shmem = (shmem *)NULL;  id = 0;
    return;
    }
  if (flag == LOG_READER)
    shmem->readidx = shmem->writeidx = 0;

  semas = new semaset(LOG_SEMA_KEY, SH_SEMAS,
    SH_SEMA_OPS, flag);

  if (flag == LOG_READER)     // se leitor, inicializa semaforos
    {
    semas->init(SH_MON_SEMA, SH_SEMA_TRUE);
    semas->init(SH_NFULL_SEMA, SH_SEMA_TRUE);
    semas->init(SH_NEMPTY_SEMA, SH_SEMA_FALSE);
    }

  key = SHLOG_KEY;
  blines = SHBUF_LINES;  bsize = SHLINE_SIZE;

  if (flag == LOG_READER)
    owner = getpid();
}
```

```
logmem :: ~logmem()
{
  if (id == 0)
    return;

  if (owner == getpid())
    remove();          // apaga a regiao de memoria compartilhada

  delete semas;
}
```

A chamada de sistema *semget()* aloca a região de memória compartilhada e dá a ela uma identificação (ID) única baseada na chave dada. Note que a função construtor *logmem* liga a memória compartilhada com a chamada de sistema *shmat()* e também cria e inicializa os semáforos, de maneira que todos os usuários os utilizam. A função destrutor remove a área de memória compartilhada do sistema; a função "criador" também remove a série semáforo.

A operação *remove()*, chamada somente pelo leitor, remove os semáforos e depois remove do núcleo o segmento de memória compartilhada. Os escritores simplesmente chamam *detach()*, que usa a chamada de sistema UNIX *shmdt()* para remover o espaço compartilhado de seus espaços de endereçamento. (Veja Listagem 9.15.)

Listagem 9.15 Removendo memória compartilhada: *remove()* e *detach()*

```
int logmem :: remove()
{
  errno = 0;      // inicializa codigo de erros do sistema

  if ((owner != getpid()) || (shmem == (shmem *)NULL))
    return FAILURE;

  if (id == 0)
    return FAILURE;

  if (owner == getpid())
    if (semas->remove() == FAILURE)
    return FAILURE;

  /* se possui memoria compartilha, entao remove do sistema */
  if (owner == getpid())
    if (shmctl(id, IPC_RMID, (shmid_ds *)NULL) == FAILURE)
      return FAILURE;
```

```
  return SUCCESS;
}

int logmem :: detach()
{
  errno = 0;        // inicializa codigo de erros do sistema

  /* desconecta do segmento de memoria compartilhada */
  /* (sem remover do sistema) */
  if ((id == 0) || (shmdt((char *)shmem) == FAILURE))
    return FAILURE;

  return SUCCESS;
}
```

Outras Funções

Finalmente, vamos examinar o restante das rotinas do programa, incluindo as rotinas "dump" (descarregar) usadas na depuração das implementações. A operação *read()* usa a operação *logmem::empty()* para verificar se são possíveis múltipalas leituras continuamente.

Listagem 9.16 Saindo da memória compartilhada: *remove()* e *detach()*

```
int logmem :: empty()
{
  if (semas->is0(SH_NEMPTY_SEMA))
    return TRUE;
  else
    return FALSE;
}

void logmem :: dump()
{
  if (shmem != (shmem *)NULL)
    cerr << "le idx: " << shmem->readidx
      << "escreve idx: " << shmem->writeidx << "\n";

  return;
}
```

Os semáforos têm outras funções, das quais nem todas são usadas nesta aplicação. Elas são usadas para descobrir se um semáforo é zero ou se é positivo (não zero), e também para zerar um semáforo [note que é *semctl()* que faz isso, e não *semop()*].

Listagem 9.17 Outras funções *semaset(): wait0(), is0(), isp(), doset0()* e *dump()*

```
int semaset :: wait0(int sema)
{
  if (id == 0)
    return FAILURE;

  if (curops >= nops)
    return FAILURE;

  sops[curops].sem_num = sema;    // indice do vetor de semaforo
  sops[curops].sem_op = 0;        // retorna se semaforo = 0;
                                  // senao espera
  sops[curops++].sem_flg = 0;     // se extinto, nada a fazer

  return curops;     // retorna numero de opcoes do vetor, ate aqui
}

int semaset :: is0(int sema)
{
  errno = 0;      // inicializa codigo de erros do sistema

  if (id == 0)
    return FAILURE;

  if ((sema < 0) || (sema > (nsemas - 1)))
    return FAILURE;       // certifica-se que numero do semaforo
                          // esta' na serie

  union semum arg; // argumento inicial val para semctl nao e usual
  arg.val = 0;

  int ret;
  // determina o valor corrente do semaforo
  ret = semctl(id, sema, GETVAL, arg);

  if (ret == FAILURE)
    return FAILURE;         // errno e' estabelecido pelo sistema

  if (ret == 0)
```

```
    return TRUE;
  else
    return FALSE;
}

int semaset :: isp(int sema)
{
  errno = 0;      // inicializa codigo de erros do sistema

  if (id == 0)
    return FAILURE;

  if ((sema < 0) || (sema > (nsemas - 1)))
    return FAILURE;       // certifica-se que numero de semaforo
                          // esta' na serie
  union semum arg; // argumento inicial val para semctl nao e usual
  arg.val = 0;

  int ret;
  // determina o valor corrente do semaforo
  ret = semctl(id, sema, GETVAL, arg);

  if (ret == FAILURE)
    return FAILURE;         // errno e estabelecido pelo sistema

  if (ret > 0)
    return TRUE;
  else
    return FALSE;
}

int semaset :: doset0(int sema)
{
  errno = 0;      // inicializa codigo de erros do sistema

  if (id == 0)
    return FAILURE;

  if ((sema < 0) || (sema > (nsemas - 1)))
    return FAILURE;       // certifica-se que numero de semaforo
                          //esta' na serie

  union semum arg; // argumento inicial val para semctl nao e usual
```

```
  arg.val = 0;      // zera semaforo

  // inicializa o semaforo na serie com um valor inicial desejado
  if (semctl(id, sema, GETVAL, arg) == FAILURE)
    return FAILURE;          // errno e' estabelecido pelo sistema

  return SUCCESS;
}

int semaset :: dump()
{
  cerr << "num ops: " << nops << " cur ops: "
    << curops << "\n";

  for (int i=0; i<curops; i++)
    cerr << "sembuf[" << i << "]: "
      << form("n: %d op: %d flg: %d",
      sops[i].sem_num, sops[i].sem_op,
      sops[i].sem_flg) << "\n";

  return;
}
```

Processos de Testes do Leitor e do Escritor

Agora que estão completas as implementações dos semáforos e a região da memória compartilhada, os usuários podem colher os benefícios desses esforços, tornados mais seguros e fáceis com C++. As rotinas de prova são quase embaraçosamente simples, empregando a maior parte do seu tempo usando os objetos fornecidos, em vez de estarem envolvidas com os detalhes das implementações de semáforo e registro compartilhado. E, o que é mais importante, os usuários de *logmem* e *semaset* não têm acesso às funções-membro privadas daquelas classes.

Agora, vamos examinar a codificação para o programa do leitor:

Listagem 9.18 O programa leitor do buffer de registro de memória compartilhada: *reader.c*

```
/* reader.c rotina de teste do leitor de memoria compartilhada */

#include <stream.h>
```

```
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>
#include "shlog.h"
#include "semaset.h"
#include "logmem.h"

extern int sleep(int naptime);
extern int sigcatch(int sig);

int gotsig = FALSE;     // definido pela rotina captadora do sinal

logmem logmem(LOG_READER);

main()
{
  signal(SIGTERM, sigcatch);

  char myline[SHLINE_SIZE+1];
  nyline[SHLINE_SIZE] = '\0'; // certifica-se de que termina com \0

  while (!gotsig)
    {
    sleep(5);       // espera por tempo arbitrario

    // obtem o monitor e espera ate nao vazio
    (void)logmem.monnempty();

    /* le tudo de novo que pode ser colocado no buffer */
    while (logmem.read(myline) != FAILURE)
      cout << "\nleitor " << "\"" << myline << "\"\n";

  (void)logmem.postmon();       // cede o monitor
    }
}

int sigcatch(int sig)
{
  cerr << "\npega sinal " << sig << "; saindo\n";
  gotsig = TRUE;          // tem o sinal; sai do loop

  exit(0);
}
```

Note que o leitor continua rodando até que receba um sinal *terminate* (termina). Com o processo *exit()* (saída) na rotina de manuseio de interrupção, a função destrutor para *logmem* será chamada corretamente? (Sim, porque o tradutor C++ insere construtores para os objetos globais na introdução do programa, e insere destrutores para aqueles objetos antes da rotina de saída do sistema.) Observe que o leitor lê todas as linhas no buffer compartilhado, quando ele assume o controle.

O programa escritor é mais direto:

Listagem 9.19 Programa escritor do buffer de memória compartilhada: *writer.c*

```
/* writer.c rotina de teste do escritor de memoria compartilhada */

#include <stream.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "shlog.h"
#include "semaset.h"
#include "logmem.h"

extern int sleep(int naptime);

main(int argc, char * argv[])
{
  sleep(2);          // espera que o leitor inicialize

  logmem logmem(0);

  char myline[SHLINE_SIZE+1];
  int mypid = getpid();      // designa processo escritor

  for (int i=0; i<10; i++)
    {
      sleep(2);              // espera por tempo arbitrario

      logmem.monnfull();     // toma o monitor, espera ate nao cheio

      /* escreve uma linha no buffer */
      strcpy(myline, form("%d cadeia %d para buffer",mypid, i);
      logmem.write(myline);
```

```
        cout << "\nescrito " << myline << "\n";
    logmem.postmon();       // desiste do monitor
  }
  cerr << "\ndesconectado de shmem log " << logmem.detach()
    << " erro no. " << errno << "\n";
}
```

As chamadas de sistema *sleep()* são usadas para forçar os escritores a ceder a CPU para permitir a execução do leitor e outros escritores, possibilitando assim uma melhor mistura de mensagens de diferentes programas na tela do terminal (e proporcionando um melhor teste das rotinas). [A rotina da biblioteca UNIX *sleep()* atrasa um processo durante um certo número de segundos.] Os processos mostram suas saídas no terminal (saída-padrão), mas você pode facilmente desviar a saída para um arquivo no sistema UNIX. Se você digita:

```
leitor &
escritor &
escritor &
escritor &
escritor &
```

no terminal do seu sistema UNIX, verá mensagens dos escritores (numeradas de um a dez) "interfolheadas" com o leitor na tela do seu terminal. Note que, quando o leitor obtém o monitor, ele lê todas as mensagens afixadas pelos diferentes escritores àquele ponto. (A notação & instrui o UNIX para rodar os programas em "background", ou, mais corretamente, devolve o controle para o "shell" do usuário após ele iniciar os processos.)

Uma observação final — usando o recurso *make* do sistema UNIX, você pode reformar todos os programas alterados digitando:

```
make
```

com o seguinte arquivo *makefile*:

Listagem 9.20 Teste leitor/escritor *makefile*

```
LIBS = logmem.o semaset.o
HEADERS = shlog.h semaset.h logmem.h
DEBUG = +i
CFLAGS = $(DEBUG) -g
```

```
all:    reader writer
      @echo "leitor e escritor estao atualizados"

reader:  reader.o $(LIBS)
      CC -o reader $(CFLAGS) reader.o $(LIBS)

reader.o: reader.c $(HEADERS)
      CC -c $(CFLAGS) reader.c

writer:  writer.o $(LIBS)
      CC -o writer $(CFLAGS) writer.o $(LIBS)

writer.o: writer.c $(HEADERS)
      CC -c $(CFLAGS) writer.c

logmem.o: logmem.c $(HEADERS)
      CC -c $(FLAGS) logmem.c

semaset.o: semaset.c $(HEADERS)
      CC -c $(FLAGS) semaset.c
```

Existem também utilitários *make* semelhantes disponíveis no MS-DOS.

Resumo

Este capítulo discutiu o uso da linguagem C++ com o UNIX, destacando sua abstração de dados e suas propriedades objeto-orientadas para especificar claramente interfaces entre os objetos. Após examinar os problemas de programação multitarefa correta, sistemas de processos cooperativos, você viu como definir classes C++ paras encapsular elegantemente soluções seguras para aqueles problemas difíceis.

O exemplo principal demonstrou como um leitor e vários escritores podem compartilhar o buffer de memória que pode conter muitas mensagens. A solução utilizou a memória compartilhada e os semáforos do System V e ocultou os detalhes complexos da implementação em classes de dados abstratos, de maneira que os processos clientes leitor e escritor pudessem usar uma interface simples e segura.

Projetos de Programação

Os exercícios a seguir, cujas respostas não estão neste livro, sugerem outras variações para esse exemplo.

1. Implemente o exemplo usando mensagens em vez de memória compartilhada e semáforos; use o exemplo inicial como base para o seu trabalho. A sua implementação UNIX é boa com referência aos múltiplos escritores bloqueados? Qual implementação é mais eficiente?

2. Use "named pipes" em lugar de memória compartilhada e semáforos; baseie sua solução no exemplo parcial inicial. Como você manipulou marcas inter-registro na "pipeline"? Como você garantiu as fronteiras dos registros? Se o leitor uma vez perder o registro do início de uma mensagem devido a estouro no buffer por um escritor, como você pode conseguir a sincronização do leitor novamente?

3. Use semáforos contadores, não semáforos binários. Com isso talvez você precise de menos chamadas de sistema, mas seja cuidadoso com as situações "livelock" (soterramento) e com o sinalizador da operação semáforo "undo".

4. Faça com que as rotinas de "dump" (descarga) usem *stream I/0*; use funções sobrecarregadas *ostream& operator<<() (logmem&)*, de modo que *cout<<logmem* ou *cout<<semas* funcionem.

5. Reescreva o programa para leitores múltiplos. Você precisou de um "criador" de segmento de memória compartilhada "demon"? Ou você usou o primeiro leitor/escritor? E se o "criador" não for o último que está ativo (vivo)? Você pode iniciar o "demon" ou primeiro processa */etc/inittab* do UNIX?

6. Use a herança quando estiver codificando semáforos, memória compartilhada, ou ambos. Você pode usar *amigas* em vez do refinamento "de cima para baixo"?

7. Use a sobrecarga de operador em lugar das funções denominadas para as operações semáforo. Isso é uma boa idéia? (Alguns programadores só vêem aritmética quando vêem sinais +, não quando vêem "semáforo afixado".)

8. Mapeie a seção de memória compartilhada para a tela do terminal. Como você pode alterá-la quando a memória muda?

9. Use a idéia de memória compartilhada para desenvolver um controlador de tela para múltiplos processos. Use *curses*. Neste exemplo, o leitor deverá ser passivo. Como você mudou as rotinas de acesso (funções-membro-classe) para permitir múltiplos leitores concorrentes?

10. Expanda o exercício anterior para proporcionar a cada escritor uma janela num terminal não-inteligente.

Makron
Books

Índice Analítico

D

E

R

S

T

U

V

Impressão e acabamento
(com filmes fornecidos):
EDITORA SANTUÁRIO
Fone (0125) 36-2140
APARECIDA - SP

CADASTRO PARA MALA DIRETA

Favor preencher todos os campos

NOME (não abreviar):

END. RESIDENCIAL:

CIDADE: UF: CEP: FONE:

EMPRESA:

SEXO: F ☐ M ☐ NASCIMENTO ____/____ (MÊS / ANO)

1. Cargo: ☐1.Digitador ☐2.Programador ☐3.Analista ☐4.Supervisor ☐5.Gerente ☐6.Diretor ☐7.Outros (especificar): _____

2. Escolaridade: ☐1.1º grau ☐2.2º grau ☐3.3º grau

3. Eu geralmente compro livros: ☐1.Em livrarias ☐2.Por telefone ☐3.Em feiras ☐4.Por Reembolso Postal ☐5.Outros (especificar): _____

4.Costumo comprar, em média, a seguinte quantidade de livros por ano: ☐1.01 ☐2.03 ☐3.05 ☐4.mais

5.Estou interessado em livros das seguintes áreas: ☐1.Informática ☐2.Negócios ☐3.Controle Qualidade ☐4.Informática na Educação ☐5.Ensino Inglês ☐6.Psicologia ☐7.Sociologia ☐8.Metodologia ☐9.Medicina ☐10.Eng.Civil ☐11.Eng.Elétrica/ Eletrônica ☐12.Eng.Mecânica ☐13.Eng.Química ☐14.Física ☐15.Matemática ☐16.Outros (especificar): _____

6.Utiliza microcomputador: ☐sim ☐não

7.Qual? ☐1.PC ☐2.Apple ☐3.MSX ☐4.Outros (especificar): _____

8.Software mais utilizados: ☐1.Banco de Dados ☐2.Planilha ☐3.Proc. Texto ☐4.Outros (especificar): _____

9.Linguagens: ☐1.C ☐2.Cobol ☐3.Clipper/dBase ☐4.Assembler ☐5.Outros (especificar): _____

Obra: Programando em C++
Autor: John Berry

MAKRON *Books* Editora Ltda.
Editora McGRAW-HILL Ltda.
"Padrão em Livros de Qualidade"
Rua Tabapuã, 1105 – Itaim Bibi – São Paulo
04533 – Tels.: (011) 881-8604 / 820-8528

CADASTRO PARA MALA DIRETA

Favor preencher todos os campos

NOME

ENDEREÇO

BAIRRO

CIDADE

ESTADO

CEP

NASCIMENTO ____/____/____ SEXO ☐ M ☐ F

PROFISSÃO

EMPRESA

ENDEREÇO

BAIRRO

CIDADE

ESTADO

CEP

TELEFONE

Como tomou conhecimento da Digital Books?

☐ Anúncio em revista ☐ Anúncio em jornal ☐ Indicação de amigo

☐ Mala direta ☐ Internet ☐ Outros: _____

Quais títulos de seu interesse?

☐ Visual Basic 6 ☐ Visual C++ 6 ☐ Delphi 5

☐ Java ☐ Linux ☐ Windows NT

☐ Windows 2000 ☐ Redes ☐ Hardware

☐ Banco de Dados ☐ Internet ☐ Outros: _____

Qual seu nível de conhecimento em informática?

☐ Básico ☐ Intermediário ☐ Avançado

Qual seu cargo?

☐ Programador ☐ Analista ☐ Gerente

☐ Diretor ☐ Estudante ☐ Outros: _____

Qual seu interesse em livros técnicos?

☐ Uso profissional ☐ Uso pessoal ☐ Estudo

Quantos livros técnicos você compra por ano?

☐ 1 a 3 ☐ 4 a 6 ☐ 7 a 9 ☐ mais de 10

Quantos livros técnicos você lê por ano?

☐ 1 a 3 ☐ 4 a 6 ☐ 7 a 9 ☐ mais de 10

Você pretende comprar livros técnicos nos próximos 12 meses?

☐ Sim, mais do que comprei nos últimos 12 meses

☐ Sim, a mesma quantidade que comprei nos últimos 12 meses

☐ Sim, menos do que comprei nos últimos 12 meses

☐ Não

The Waite Group's

PROGRAMANDO EM C++

OUTROS LIVROS NA ÁREA

Hergert	*O ABC do Turbo C*
Jamsa	*C Library – Bibliotecas*
Mayer	*Linguagem C ANSI – Um Curso Introdutório para Programadores*
Mayer	*Integrando Clipper com Linguagem C*
Mizrahi	*Treinamento em Linguagem C – Módulos 1 e 2*
Plauger	*Standard C – Guia de Referência Básica*
Pugh	*Programando em Linguagem C*
Quintela	*Clipper com Linguagem C*
Schildt	*Linguagem C – Guia do Usuário*
Schildt	*C Avançado – Guia do Usuário – 2ª edição*
Schildt	*Linguagem C – Guia Prático e Interativo*
Schildt	*Turbo C Avançado – Guia do Usuário*
Schildt	*C – Guia de Referência Básica*
Schildt	*Turbo C – Guia do Usuário – 2ª edição revisada*
Silveira	*Linguagem C – Comandos Básicos – Guia do Operador*

MAKRON *Books*

0-07-460547-X