8

Ponteiros e strings baseadas em ponteiro

Os endereços nos são dados para ocultar nosso paradeiro.
Saki (H. H. Munro)

Usando de cautela e circunlóquios, chegamos ao caminho por desvios.
William Shakespeare

Muitas coisas, quando tudo parece apontar para o consenso, podem funcionar de maneira adversa.
William Shakespeare

Você descobrirá que é uma prática muito boa sempre verificar suas referências, senhor!
Dr. Routh

OBJETIVOS

Neste capítulo, você aprenderá:

- O que são ponteiros.
- As semelhanças e diferenças entre ponteiros e referências e quando utilizar cada um.
- Como utilizar ponteiros para passar argumentos a funções por referência.
- Como utilizar strings no estilo C baseadas em ponteiro.
- Os estreitos relacionamentos entre ponteiros, arrays e strings no estilo C.
- Como utilizar ponteiros para funções.
- A declarar e utilizar arrays de strings no estilo C.

Sumário

8.1 Introdução

Este capítulo discute um dos recursos mais poderosos da linguagem de programação C++, o ponteiro. No Capítulo 6, vimos que as referências podem ser utilizadas para realizar a passagem por referência. Os ponteiros também permitem a passagem por referência e podem ser utilizados para criar e manipular estruturas de dados dinâmicas (isto é, estruturas de dados que podem crescer e encolher), como listas vinculadas, filas, pilhas e árvores. Este capítulo explica conceitos básicos do ponteiro e reforça o relacionamento íntimo entre arrays e ponteiros. A visão de arrays como ponteiros deriva da linguagem de programação C. Como vimos no Capítulo 7, a classe C++ Standard Library `vector` fornece uma implementação de arrays como objetos completos.

De modo semelhante, o C++ realmente oferece dois tipos de strings — objetos da classe `string` (que utilizamos desde o Capítulo 3) e strings `char *` baseadas em ponteiro no estilo C. Este capítulo sobre ponteiros discute as strings `char *` para aprofundar seu conhecimento de ponteiros. De fato, as strings terminadas por caractere nulo introduzidas na Seção 7.4 e utilizadas na Figura 7.12 são strings `char *` baseadas em ponteiro. Este capítulo também inclui uma coleção considerável de exercícios de processamento de strings que utilizam strings `char *`. As strings `char *` baseadas em ponteiro no estilo C são amplamente utilizadas em sistemas C e C++ legados. Então, se trabalha com sistemas C ou C++ legados, você pode ser solicitado a manipular essas strings `char *` baseadas em ponteiro.

Examinaremos o uso de ponteiros com classes no Capítulo 13, "Programação orientada a objetos: polimorfismo", em que veremos que o chamado 'processamento polimórfico' de programação orientada a objetos é realizado com ponteiros e referências. O Capítulo 21, "Estruturas de dados", apresenta exemplos de como criar e utilizar estruturas de dados dinâmicas que são implementadas com ponteiros.

8.2 Declarações de variável ponteiro e inicialização

As variáveis ponteiro contêm endereços de memória como seus valores. Normalmente, uma variável contém diretamente um valor específico. Mas um ponteiro contém o endereço de memória de uma variável que, por sua vez, contém um valor específico. Nesse sentido, o nome de uma variável **referencia um valor diretamente** e um ponteiro **referencia um valor indiretamente** (Figura 8.1). Referenciar um valor por um ponteiro é freqüentemente chamado de **indireção**. Observe que os diagramas normalmente representam um ponteiro como uma seta da variável que contém um endereço para a variável localizada nesse endereço na memória.

Os ponteiros, como qualquer outra variável, devem ser declarados antes de ser utilizados. Por exemplo, para o ponteiro na Figura 8.1, a declaração

```
int *countPtr, count;
```

declara a variável `countPtr` como do tipo `int *` (isto é, um ponteiro para um valor `int`) e exibe '`countPtr` é um ponteiro para `int`' ou '`countPtr` aponta para um objeto tipo `int`'. Além disso, a variável `count` na declaração anterior é declarada como um `int`, não como

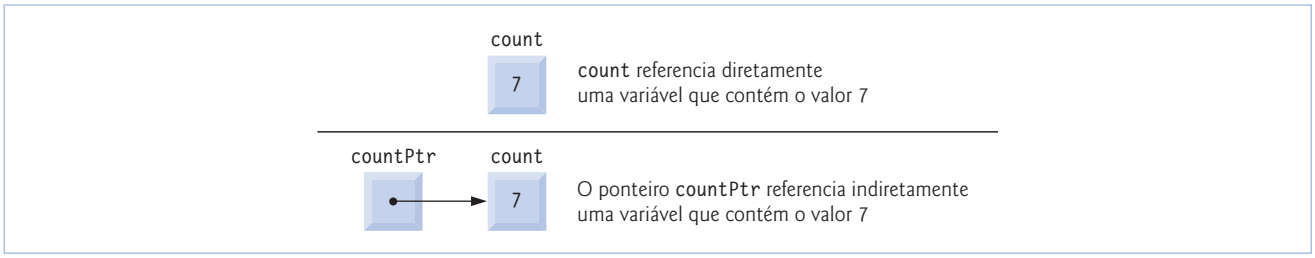

Figura 8.1 Referenciando direta e indiretamente uma variável.

um ponteiro para um `int`. O `*` na declaração se aplica apenas a `countPtr`. Toda variável sendo declarada como um ponteiro deve ser precedida por um asterisco (`*`). Por exemplo, a declaração

```
double *xPtr, *yPtr;
```

indica que tanto `xPtr` como `yPtr` são ponteiros para valores `double`. Quando `*` aparece em uma declaração, ele não é um operador; em vez disso, indica que a variável sendo declarada é um ponteiro. Os ponteiros podem ser declarados para apontar para objetos de qualquer tipo de dados.

Erro comum de programação 8.1

*Supor que o * utilizado para declarar um ponteiro distribui-se por todos os nomes de variável na lista de variáveis separadas por vírgulas de uma declaração pode levar a erros. Cada ponteiro deve ser declarado com o * prefixado ao nome (com ou sem espaço entre eles — o compilador ignora o espaço). Declarar apenas uma variável por declaração ajuda a evitar esse tipo de erro e melhora a legibilidade do programa.*

Boa prática de programação 8.1

Embora não seja um requisito, incluir as letras `Ptr` nos nomes de variáveis ponteiro torna claro que essas variáveis são ponteiros e devem ser tratadas apropriadamente.

Os ponteiros devem ser inicializados quando forem declarados ou em uma atribuição. Um ponteiro pode ser inicializado como `0`, **`NULL`** ou como um endereço. Um ponteiro com o valor `0` ou `NULL` não aponta para nada e é conhecido como um **ponteiro nulo**. A constante simbólica `NULL` é definida no arquivo de cabeçalho `<iostream>` (e em vários outros arquivos de cabeçalho de biblioteca-padrão) para representar o valor `0`. Inicializar um ponteiro como `NULL` é equivalente a inicializar um ponteiro como `0`, mas em C++, o `0` é utilizado por convenção. Quando `0` é atribuído, ele é convertido em um ponteiro do tipo apropriado. O valor `0` é o único valor do tipo inteiro que pode ser atribuído diretamente a uma variável ponteiro sem primeiro fazer coerção do inteiro em um tipo ponteiro. A atribuição de endereço numérico de uma variável a um ponteiro é discutida na Seção 8.3.

Dica de prevenção de erro 8.1

Inicialize ponteiros para impedir apontar para áreas da memória desconhecidas ou não inicializadas.

8.3 Operadores de ponteiro

O **operador de endereço (`&`)** é um operador unário que retorna o endereço de memória de seu operando. Por exemplo, considerando as declarações

```
int y = 5; // declara variável y
int *yPtr; // declara variável ponteiro yPtr
```

a instrução

```
yPtr = &y; // atribui o endereço de y a yPtr
```

atribui o endereço da variável `y` à variável ponteiro `yPtr`. Então dizemos que a variável `yPtr` 'aponta para' `y`. Agora, `yPtr` referencia indiretamente o valor da variável `y`. Observe que o uso do `&` na instrução de atribuição anterior não é o mesmo uso do `&` em uma declaração de variável de referência, que sempre é precedida por um nome de tipo de dados.

A Figura 8.2 mostra uma representação esquemática da memória depois da atribuição precedente. O 'relacionamento de apontar' é indicado desenhando uma seta a partir da caixa que representa o ponteiro `yPtr` na memória até a caixa que representa a variável `y` na memória.

A Figura 8.3 mostra outra representação do ponteiro na memória, supondo que a variável do tipo inteiro `y` está armazenada na posição da memória `600000` e que a variável ponteiro `yPtr` é armazenada na posição da memória `500000`. O operando do operador de

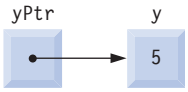

Figura 8.2 Representação gráfica de um ponteiro que aponta para uma variável na memória.

Figura 8.3 Representação de `y` e `yPtr` na memória.

endereço deve ser um *lvalue* (isto é, algo a que um valor pode ser atribuído, como um nome de variável ou uma referência); o operador de endereço não pode ser aplicado a constantes ou expressões que não resultam em referências.

O **operador *** , comumente referido como **operador de indireção** ou **operador de desreferência**, retorna um sinônimo (isto é, um alias ou um apelido) ao objeto para o qual seu operando de ponteiro aponta. Por exemplo (referindo-se novamente à Figura 8.2), a instrução

```
cout << *yPtr << endl;
```

imprime o valor de variável `y`, isto é, `5`, assim como faria a instrução

```
cout << y << endl;
```

Utilizar o * dessa maneira é chamado de **desreferenciar um ponteiro**. Observe que um ponteiro desreferenciado também pode ser utilizado no lado esquerdo de uma instrução de atribuição, como em

```
*yPtr = 9;
```

que atribuiria `9` a `y` na Figura 8.3. O ponteiro desreferenciado também pode ser utilizado para receber um valor de entrada como em

```
cin >> *yPtr;
```

o que coloca o valor de entrada em `y`. O ponteiro desreferenciado é um *lvalue*.

Erro comum de programação 8.2

Desreferenciar um ponteiro que não foi inicializado de modo adequado ou que não foi atribuído para apontar a uma posição específica na memória poderia produzir um erro fatal em tempo de execução, ou poderia modificar acidentalmente dados importantes e permitir que o programa executasse até a conclusão, possivelmente com resultados incorretos.

Erro comum de programação 8.3

Uma tentativa de desreferenciar uma variável que não é um ponteiro é um erro de compilação.

Erro comum de programação 8.4

Desreferenciar um ponteiro nulo é normalmente um erro fatal em tempo de execução.

O programa na Figura 8.4 demonstra os operadores de ponteiro & e *. Neste exemplo, as posições da memória são enviadas para a saída por << como inteiros hexadecimais (isto é, base 16). (Consulte o Apêndice D, "Sistemas de numeração", para obter informações adicionais sobre inteiros hexadecimais.) Observe que os endereços de memória hexadecimal enviados para a saída por esse programa dependem do compilador e do sistema operacional; portanto, você pode obter diferentes resultados ao executar o programa.

Dica de portabilidade 8.1

O formato em que um ponteiro é enviado para a saída depende do compilador. Alguns sistemas geram saída de valores de ponteiro como inteiros hexadecimais, enquanto outros sistemas utilizam inteiros decimais.

Note que o endereço de `a` (linha 15) e o valor de `aPtr` (linha 16) são idênticos na saída, confirmando que o endereço de `a` é, de fato, atribuído à variável ponteiro `aPtr`. Os operadores & e * são o oposto um do outro — quando os dois são aplicados consecutivamente a `aPtr` em qualquer ordem, eles se 'cancelam' e o mesmo resultado (o valor em `aPtr`) é impresso.

```
1  // Figura 8.4: fig08_04.cpp
2  // Utilizando os operadores & e *.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int a; // a é um inteiro
10    int *aPtr; // aPtr é um ponteiro int * -- para um inteiro
11
12    a = 7; // atribuiu 7 a a
13    aPtr = &a; // atribui o endereço de a ao aPtr
14
15    cout << "The address of a is " << &a
16       << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18       << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20       << "each other.\n&*aPtr = " << &*aPtr
21       << "\n*&aPtr = " << *&aPtr << endl;
22    return 0; // indica terminação bem-sucedida
23 } // fim de main
```

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

Figura 8.4 Operadores de ponteiro & e *.

A Figura 8.5 lista a precedência e a associatividade dos operadores introduzidos até agora. Observe que o operador de endereço (&) e o de desreferência (*) são operadores unários no terceiro nível de precedência no gráfico.

8.4 Passando argumentos para funções por referência com ponteiros

Há três maneiras em C++ de passar argumentos para uma função — passagem por valor, **passagem por referência com argumentos de referência** e **passagem por referência com argumentos de ponteiro**. O Capítulo 6 comparou e contrastou a passagem por valor e a passagem por referência com argumentos de referência. Nesta seção, explicamos como passar por referência com argumentos de ponteiro.

Como vimos no Capítulo 6, `return` pode ser utilizado para retornar um valor de uma função chamada para um chamador (ou retornar o controle de uma função chamada sem passar um valor de volta). Vimos também que os argumentos podem ser passados para uma função usando argumentos de referência. Esses argumentos permitem que a função chamada modifique os valores originais dos argumentos no chamador. Os argumentos de referência também permitem aos programas passar grandes objetos de dados para uma função e evitam o overhead de passar os objetos por valor (o que, naturalmente, exige a produção de uma cópia do objeto). Os ponteiros, como as referências, também podem ser utilizados para modificar uma ou mais variáveis no chamador ou passar ponteiros para objetos grandes de dados a fim de evitar o overhead de passar os objetos por valor.

Em C++, os programadores podem utilizar ponteiros e o operador de indireção (*) para realizar a passagem por referência (exatamente como a passagem por referência é feita em programas C, porque o C não tem referências). Ao chamar uma função com um argumento que deve ser modificado, o endereço do argumento é passado. Isso normalmente é realizado aplicando o operador de endereço (&) ao nome da variável cujo valor será modificado.

Operadores	Associatividade	Tipo
`()` `[]`	da esquerda para a direita	mais alto
`++` `--` `static_cast< tipo >( operando )`	da esquerda para a direita	unário (pós-fixo)
`++` `--` `+` `-` `!` `&` `*`	da direita para a esquerda	unário (prefixo)
`*` `/` `%`	da esquerda para a direita	multiplicativo
`+` `-`	da esquerda para a direita	aditivo
`<<` `>>`	da esquerda para a direita	inserção/extração
`<` `<=` `>` `>=`	da esquerda para a direita	relacional
`==` `!=`	da esquerda para a direita	igualdade
`&&`	da esquerda para a direita	E lógico
`||`	da esquerda para a direita	OU lógico
`?:`	da direita para a esquerda	ternário condicional
`=` `+=` `-=` `*=` `/=` `%=`	da direita para a esquerda	atribuição
`,`	da esquerda para a direita	vírgula

Figura 8.5 Precedência e associatividade de operadores.

Como vimos no Capítulo 7, os arrays não são passados com operador `&`, porque o nome do array é a posição inicial na memória do array (isto é, um nome de array já é um ponteiro). O nome de um array, `arrayName`, é equivalente a `&arrayName[ 0 ]`. Quando o endereço de uma variável é passado para uma função, o operador de indireção (`*`) pode ser utilizado na função para formar um sinônimo para o nome da variável — isso por sua vez pode ser utilizado para modificar o valor da variável naquela localização na memória do chamador.

As figuras 8.6 e 8.7 possuem duas versões presentes de uma função que eleva um inteiro ao cubo — `cubeByValue` e `cubeByReference`. A Figura 8.6 passa a variável `number` por valor para a função `cubeByValue` (linha 15). A função `cubeByValue` (linhas 21–24) eleva seu argumento ao cubo e passa o novo valor de volta para `main` utilizando uma instrução `return` (linha 23). O novo valor é atribuído a `number` (linha 15) em `main`. Observe que a função chamadora tem a oportunidade de examinar o resultado da chamada de função antes de modificar o valor da variável `number`. Por exemplo, nesse programa, poderíamos ter armazenado o resultado de `cubeByValue` em outra variável, examinado seu valor e atribuído o resultado a `number` somente depois de determinar que o valor retornado era razoável.

A Figura 8.7 passa a variável `number` para a função `cubeByReference` utilizando a passagem por referência com um argumento de ponteiro (linha 15) — o endereço de `number` é passado para a função. A função `cubeByReference` (linhas 22–25) especifica o parâmetro `nPtr` (um ponteiro para `int`) para receber seu argumento. A função desreferencia o ponteiro e eleva ao cubo o valor para o qual `nPtr` aponta (linha 24). Isso altera diretamente o valor de `number` em `main`.

```
1  // Figura 8.6: fig08_06.cpp
2  // Eleva uma variável ao cubo utilizando passagem por valor.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int cubeByValue( int ); // protótipo
8
9  int main()
10 {
11    int number = 5;
12
```

Figura 8.6 Passagem por valor utilizada para elevar o valor de uma variável ao cubo. (continua)

```
13     cout << "The original value of number is " << number;
14
15     number = cubeByValue( number ); // passa number por valor ao cubeByValue
16     cout << "\nThe new value of number is " << number << endl;
17     return 0; // indica terminação bem-sucedida
18  } // fim de main
19
20  // calcula e retorna o cubo do argumento inteiro
21  int cubeByValue( int n )
22  {
23     return n * n * n; // eleva a variável local n ao cubo e retorna o resultado
24  } // fim da função cubeByValue
```

```
The original value of number is 5
The new value of number is 125
```

Figura 8.6 Passagem por valor utilizada para elevar o valor de uma variável ao cubo. (continuação)

```
1   // Figura 8.7: fig08_07.cpp
2   // Eleva uma variável ao cubo usando passagem por referência com um argumento nPtr.
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6
7   void cubeByReference( int * ); // protótipo
8
9   int main()
10  {
11     int number = 5;
12
13     cout << "The original value of number is " << number;
14
15     cubeByReference( &number ); // passa endereço de number para cubeByReference
16
17     cout << "\nThe new value of number is " << number << endl;
18     return 0; // indica terminação bem-sucedida
19  } // fim de main
20
21  // calcula o cubo de *nPtr; modifica a variável number em main
22  void cubeByReference( int *nPtr )
23  {
24     *nPtr = *nPtr * *nPtr * *nPtr; // eleva *nPtr ao cubo
25  } // fim da função cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

Figura 8.7 Passagem por referência com um argumento de ponteiro utilizado para elevar ao cubo o valor de uma variável.

Erro comum de programação 8.5

Não desreferenciar um ponteiro quando é necessário fazer isso para obter o valor para o qual o ponteiro aponta é um erro.

Uma função que recebe um endereço como um argumento deve definir um parâmetro de ponteiro para receber o endereço. Por exemplo, o cabeçalho para função `cubeByReference` (linha 22) especifica que `cubeByReference` recebe o endereço de uma variável `int` (isto é, um ponteiro para um `int`) como um argumento, armazena o endereço localmente em `nPtr` e não retorna um valor.

O protótipo de função para `cubeByReference` (linha 7) contém `int *` entre parênteses. Como com outros tipos de variável, não é necessário incluir nomes de parâmetros de ponteiro em protótipos de função. Os nomes de parâmetro incluídos para propósitos de documentação são ignorados pelo compilador.

As figuras 8.8 e 8.9 analisam graficamente a execução dos programas das figuras 8.6 e. 8.7, respectivamente.

Observação de engenharia de software 8.1

Utilize a passagem por valor para passar argumentos para uma função a menos que o chamador requeira explicitamente que a função chamada modifique diretamente o valor da variável do argumento no chamador. Esse é outro exemplo do princípio do menor privilégio.

Passo 1: Antes de `main` chamar `cubeByValue`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: indefinido

Passo 2: Depois de `cubeByValue` receber a chamada:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Passo 3: Depois de `cubeByValue` elevar o parâmetro n ao cubo e antes de `cubeByValue` retornar a `main`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

125

n: 5

Passo 4: Depois de `cubeByValue` retornar a `main` e antes de atribuir o resultado a `number`:

```
int main()
{
   int number = 5;

   number = cubeByValue( number );
}
```

number: 5

125

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: indefinido

Passo 5: Depois de `main` completar a atribuição a `number`:

```
int main()
{
   int number = 5;
   125
   number = cubeByValue( number );
}
```

number: 125

125

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: indefinido

Figura 8.8 Análise da passagem por valor do programa da Figura 8.6.

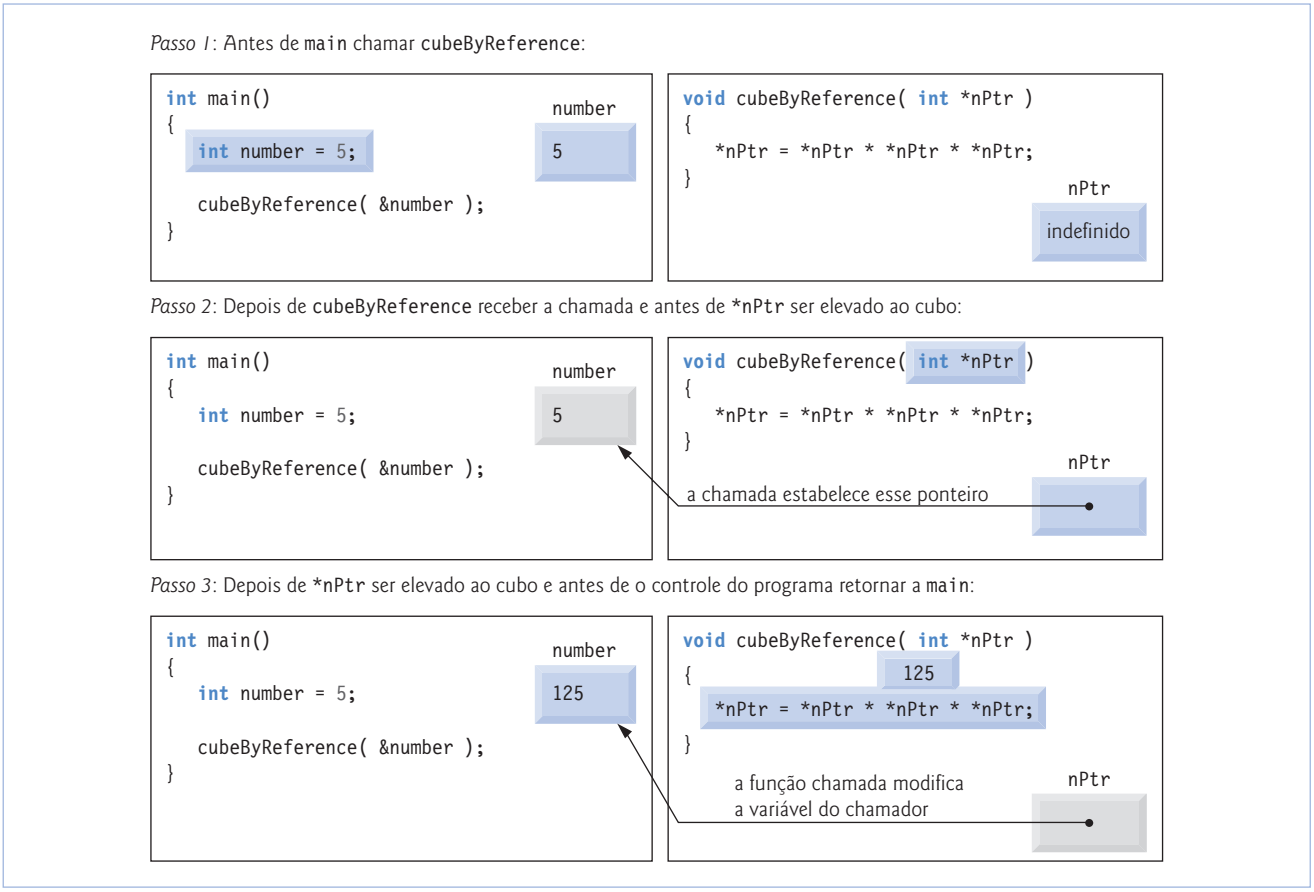

Figura 8.9 Análise da passagem por referência (com um argumento de ponteiro) do programa da Figura 8.7.

No cabeçalho e no protótipo de uma função que espera um array unidimensional como um argumento, pode-ser utilizar a notação de ponteiro na lista de parâmetros de `cubeByReference`. O compilador não diferencia entre uma função que recebe um ponteiro e uma que recebe um array unidimensional. Isso, naturalmente, quer dizer que a função deve 'saber' quando está recebendo um array ou simplesmente uma única variável pela qual realizar a passagem por referência. Quando o compilador encontra um parâmetro de função para um array unidimensional na forma `int b[]`, o compilador converte o parâmetro em notação de ponteiro `int *b` (pronuncia-se '`b` é um ponteiro para um inteiro'). Ambas as formas de declarar um parâmetro de função como um array dimensional são intercambiáveis.

8.5 Utilizando const com ponteiros

Lembre-se de que o qualificador `const` permite ao programador informar ao compilador que o valor de uma variável particular não deve ser modificado.

Dica de portabilidade 8.2

Embora `const` seja bem definido em ANSI C e C++, alguns compiladores não o impõem adequadamente. Portanto, uma boa regra é: 'Conheça seu compilador'.

Ao longo dos anos, nas primeiras versões do C, foi escrita uma grande base de código legado que não utilizava `const`, porque este qualificador não estava disponível. Por essa razão, há excelentes oportunidades de aprimorar a engenharia de software de código C antigo (também chamado de 'legado'). Além disso, muitos programadores atualmente utilizando ANSI C e C++ não utilizam `const` em seus programas, porque começaram a programar nas primeiras versões de C. Esses programadores estão perdendo muitas oportunidades de aplicar boa engenharia de software.

Há muitas possibilidades para utilizar (ou não utilizar) `const` com parâmetros de função. Como escolher a mais apropriada dessas possibilidades? Deixe o princípio do menor privilégio ser seu guia. Sempre dê a uma função acesso suficiente aos dados em seus parâmetros para realizar sua tarefa especificada, porém não mais. Esta seção discute como combinar `const` com declarações de ponteiro para impor o princípio do menor privilégio.

O Capítulo 6 explicou que, quando uma função é chamada utilizando passagem por valor, uma cópia do argumento (ou argumentos) na chamada de função é feita e passada à função. Se a cópia é modificada na função, o valor original é mantido no chamador sem alteração.

Em muitos casos, um valor passado a uma função é modificado para que a função possa realizar sua tarefa. Entretanto, em alguns casos, o valor não deve ser alterado na função chamada, mesmo que a função chamada manipule apenas uma cópia do valor original.

Por exemplo, considere uma função que aceita um array unidimensional e seu tamanho como argumentos e, subseqüentemente, imprime o array. Tal função deve fazer loop pelo array e gerar saída de cada elemento de array individualmente. O tamanho do array é utilizado no corpo da função para determinar o subscrito mais alto do array a fim de que o loop possa terminar quando a impressão for concluída. O tamanho do array não muda no corpo de função, então ele deve ser declarado como `const`. Naturalmente, como o array está apenas sendo impresso, ele também deve ser declarado como `const`. Isso é especialmente importante porque um array inteiro é *sempre* passado por referência e poderia ser facilmente alterado na função chamada.

Observação de engenharia de software 8.2

Se um valor não muda (ou não deve mudar) no corpo de uma função para o qual ele é passado, o parâmetro deve ser declarado como `const` para assegurar que ele não seja modificado acidentalmente.

Se houver uma tentativa de modificar um valor `const`, um aviso ou um erro é emitido, dependendo do compilador particular.

Dica de prevenção de erro 8.2

Antes de utilizar uma função, verifique seu protótipo de função para determinar os parâmetros que ele pode modificar.

Há quatro maneiras de passar um ponteiro para uma função: um ponteiro não constante para dados não constantes (Figura 8.10), um ponteiro não constante para dados constantes (figuras 8.11 e 8.12), um ponteiro constante para dados não constantes (Figura 8.13) e um ponteiro constante para dados constantes (Figura 8.14). Cada combinação fornece um nível diferente de privilégios de acesso.

Ponteiro não constante para dados não constantes

O acesso mais alto é concedido por um **ponteiro não constante para dados não constantes** — os dados podem ser modificados pelo ponteiro desreferenciado e o ponteiro pode ser modificado para apontar para outros dados. A declaração de um ponteiro não constante para dados não constantes não inclui `const`. Esse ponteiro pode ser utilizado para receber uma string terminada por caractere nulo em uma função que altera o valor de ponteiro para processar (e, possivelmente, modificar) cada caractere na string. A partir da Seção 7.4, lembre-se de que uma string terminada com caractere nulo pode ser colocada em um array de caracteres que contém os caracteres da string e um caractere nulo que indica onde a string termina.

```
1  // Figura 8.10: fig08_10.cpp
2  // Convertendo minúsculas em maiúsculas
3  // utilizando um ponteiro não constante para dados não constantes.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <cctype> // protótipos para islower e toupper
9  using std::islower;
10 using std::toupper;
11
12 void convertToUppercase( char * );
13
14 int main()
15 {
16    char phrase[] = "characters and $32.98";
17
18    cout << "The phrase before conversion is: " << phrase;
19    convertToUppercase( phrase );
20    cout << "\nThe phrase after conversion is:  " << phrase << endl;
21    return 0; // indica terminação bem-sucedida
22 } // fim de main
23
24 // converte string em letras maiúsculas
25 void convertToUppercase( char *sPtr )
```

Figura 8.10 Convertendo uma string em letras maiúsculas. (continua)

```
26  {
27     while ( *sPtr != '\0' ) // faz loop enquanto caractere atual não é '\0'
28     {
29        if ( islower( *sPtr ) ) // se o caractere estiver em minúsculas,
30           *sPtr = toupper( *sPtr ); // converte em maiúsculas
31
32        sPtr++; // move sPtr para o próximo caractere na string
33     } // fim do while
34  } // fim da função convertToUppercase
```

```
The phrase before conversion is: characters and $32.98
The phrase after conversion is:  CHARACTERS AND $32.98
```

Figura 8.10 Convertendo uma string em letras maiúsculas. (continuação)

Na Figura 8.10, a função `convertToUppercase` (linhas 25–34) declara o parâmetro `sPtr` (linha 25) como sendo um ponteiro não constante para dados não constantes (novamente, `const` não é utilizado). A função processa um caractere por vez a partir da string terminada por caractere nulo armazenada no array de caracteres `phrase` (linhas 27–33). Tenha em mente que o nome de um array de caracteres na realidade equivale a um ponteiro para o primeiro caractere do array, então é possível passar `phrase` como um argumento para `convertToUppercase`. A função **islower** (linha 29) aceita um argumento de caractere e retorna verdadeiro se o caractere for uma letra minúscula e falso, caso contrário. Os caracteres no intervalo de `'a'` a `'z'` são convertidos em suas letras maiúsculas correspondentes pela função `toupper` (linha 30); outros permanecem inalterados — a função `toupper` aceita um caractere como um argumento. Se o caractere for uma letra minúscula, a letra maiúscula correspondente será retornada; caso contrário, o caractere original será retornado. A função `toupper` e a `islower` fazem parte da biblioteca de tratamento de caracteres `<cctype>` (ver Capítulo 22, "Bits, caracteres, strings C e `structs`"). Depois de processar um caractere, a linha 32 incrementa `sPtr` por 1 (isso não seria possível se `sPtr` fosse declarado como `const`). Quando o operador `++` é aplicado a um ponteiro que aponta para um array, o endereço de memória armazenado no ponteiro é modificado para apontar para o próximo elemento do array (nesse caso, o próximo caractere na string). Adicionar um a um ponteiro é uma operação válida na **aritmética de ponteiros**, que é discutida em detalhes nas seções 8.8 e 8.9.

Ponteiro não constante para dados constantes

Um **ponteiro não constante para dados constantes** é um ponteiro que pode ser modificado para apontar para qualquer item de dados do tipo apropriado, mas os dados para os quais ele aponta não podem ser modificados por esse ponteiro. Esse ponteiro poderia ser utilizado para receber um argumento de array para uma função que irá processar cada elemento do array, mas não deve ter permissão de modificar os dados. Por exemplo, a função `printCharacters` (linhas 22–26 da Figura 8.11) declara o parâmetro `sPtr` (linha 22) para ser do tipo `const char *`, para que ele possa receber uma string baseada em ponteiro terminada por caractere nulo. A declaração é lida da direita para a esquerda como '`sPtr` é um ponteiro para uma constante do tipo caractere'. O corpo da função utiliza uma instrução `for` (linhas 24–25) para gerar saída de cada caractere na string até que o caractere nulo seja encontrado. Depois que cada caractere é impresso, o ponteiro `sPtr` é incrementado para apontar para o próximo caractere na string (isso funciona porque o ponteiro não é `const`). A função `main` cria o array `char phrase` para ser passado para `printCharacters`. Novamente, podemos passar o array `phrase` para `printCharacters` porque o nome do array é, na realidade, um ponteiro para o primeiro caractere no array.

```
1  // Figura 8.11: fig08_11.cpp
2  // Imprimindo uma string um caractere por vez utilizando
3  // um ponteiro não constante para dados constantes.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  void printCharacters( const char * ); // imprime usando ponteiro para dados const
9
10 int main()
11 {
12    const char phrase[] = "print characters of a string";
```

Figura 8.11 Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes. (continua)

```
13
14    cout << "The string is:\n";
15    printCharacters( phrase ); // imprime caracteres em phrase
16    cout << endl;
17    return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // sPtr pode ser modificado, mas não pode modificar o caractere para o qual
21 // ele aponta, isto é, sPtr é um ponteiro 'de leitura'
22 void printCharacters( const char *sPtr )
23 {
24    for ( ; *sPtr != '\0'; sPtr++ ) // nenhuma inicialização
25       cout << *sPtr; // exibe caractere sem modificação
26 } // fim da função printCharacters
```

```
The string is:
print characters of a string
```

Figura 8.11 Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes. (continuação)

A Figura 8.12 demonstra as mensagens de erro de compilação produzidas ao tentar compilar uma função que recebe um ponteiro não-constante para dados constantes e, então, tenta utilizar esse ponteiro para modificar os dados. [*Nota:* Lembre-se de que as mensagens de erro de compilador variam entre compiladores.]

Como sabemos, os arrays são tipos de dados agregados que armazenam itens de dados relacionados do mesmo tipo sob um nome. Quando uma função é chamada com um array como um argumento, o array é passado à função por referência. Entretanto, os objetos são sempre passados por valor — uma cópia do objeto inteiro é passada. Isso requer o overhead em tempo de execução de produzir uma cópia de cada item de dados no objeto e armazená-la na pilha de chamada de função. Quando um objeto deve ser passado para uma função, podemos utilizar um ponteiro para dados constantes (ou uma referência para dados constantes) para obter o desempenho da passagem por referência e a proteção da passagem por valor. Quando um ponteiro for passado para um objeto, deve-se fazer apenas uma cópia do endereço do objeto; o objeto em si não é copiado. Em uma máquina com endereços de quatro bytes, uma cópia de quatro bytes de memória é feita em vez de uma cópia de um objeto possivelmente grande.

```
1  // Figura 8.12: fig08_12.cpp
2  // Tentando modificar dados por meio de um
3  // ponteiro não-constante para dados constantes.
4
5  void f( const int * ); // protótipo
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f tenta modificação não-válida
12    return 0; // indica terminação bem-sucedida
13 } // fim de main
14
15 // xPtr não pode modificar o valor da variável constante para a qual ele aponta
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // erro: não é possível modificar objeto const
19 } // fim da função f
```

Figura 8.12 Tentando modificar dados por meio de um ponteiro não-constante para dados constantes. (continua)

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_12.cpp 18:
   Cannot modify a const object in function f(const int *)
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphtp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
   error C2166: l-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

Figura 8.12 Tentando modificar dados por meio de um ponteiro não constante para dados constantes. (continuação)

Dica de desempenho 8.1

Se eles não precisarem ser modificados pela função chamada, passe os objetos grandes utilizando ponteiros para dados constantes ou referências para dados constantes, para obter os benefícios de desempenho da passagem por referência.

Observação de engenharia de software 8.3

Passe objetos grandes utilizando ponteiros para dados constantes, ou referências para dados constantes, para obter a segurança da passagem por valor.

Ponteiro constante para dados não constantes

Um **ponteiro constante para dados não constantes** é um ponteiro que sempre aponta para a mesma posição da memória; os dados nessa posição podem ser modificados pelo ponteiro. Esse é o padrão de um nome de array. Um nome de array é um ponteiro constante para o começo do array. Todos os dados no array podem ser acessados e alterados utilizando o nome do array e o subscrito do array. Um ponteiro constante para dados não constantes pode ser utilizado para receber um array como um argumento para uma função que acessa elementos do array utilizando a notação de subscrito do array. Os ponteiros que são declarados `const` devem ser inicializados quando são declarados. (Se o ponteiro for um parâmetro de função, ele será inicializado com um ponteiro que é passado para a função.) O programa da Figura 8.13 tenta modificar um ponteiro constante. A linha 11 declara o ponteiro `ptr` para ser do tipo `int * const`. A declaração na figura é lida da direita para a esquerda como '`ptr` é um ponteiro constante para um inteiro não constante'. O ponteiro é inicializado

```
1  // Figura 8.13: fig08_13.cpp
2  // Tentar modificar um ponteiro constante para dados não constantes.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr é um ponteiro constante para um inteiro que pode
9     // ser modificado por ptr, mas ptr sempre aponta para a
10    // mesma posição da memória.
11    int * const ptr = &x; // ponteiro const deve ser inicializado
12
13    *ptr = 7; // permitido: *ptr não é const
14    ptr = &y; // erro: ptr é constante; não é possível atribuí-lo a um novo endereço
15    return 0; // indica terminação bem-sucedida
16 } // fim de main
```

Figura 8.13 Tentando modificar um ponteiro constante para dados não constantes. (continua)

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()s
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphtp5e_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
   l-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

Figura 8.13 Tentando modificar um ponteiro constante para dados não constantes. (continuação)

com o endereço de variável de inteiro `x`. A linha 14 tenta atribuir o endereço de `y` a `ptr`, mas o compilador gera uma mensagem de erro. Observe que nenhum erro ocorre quando a linha 13 atribui o valor `7` a `*ptr` — o valor não-constante para o qual `ptr` aponta pode ser modificado utilizando o `ptr` desreferenciado, mesmo que o próprio `ptr` tenha sido declarado como `const`.

Erro comum de programação 8.6

Não inicializar um ponteiro que é declarado como `const` é um erro de compilação.

Ponteiro constante para dados constantes

A menor quantidade de privilégio de acesso é concedida por um **ponteiro constante para dados constantes**. Esse ponteiro sempre aponta para a mesma posição da memória, e os dados nessa posição da memória não podem ser modificados utilizando o ponteiro. Essa é a maneira como um array deve ser passado para uma função que somente lê o array, usando a notação de subscrito de array, e que não modifica o array. O programa da Figura 8.14 declara variável ponteiro `ptr` para ser do tipo `const int * const` (linha 14). Essa declaração é lida da direita para a esquerda como '`ptr` é um ponteiro constante para um inteiro constante'. A figura mostra as mensagens de erro geradas quando se tenta modificar os dados para os quais `ptr` aponta (linha 18) e quando se tenta modificar o endereço armazenado na variável ponteiro (linha 19). Observe que não ocorre nenhum erro quando o programa tenta desreferenciar `ptr` ou gerar saída do valor para o qual `ptr` aponta (linha 16), porque nem o ponteiro nem os dados para os quais ele aponta estão sendo modificados nessa instrução.

8.6 Classificação por seleção utilizando passagem por referência

Nesta seção, definimos um programa de classificação para demonstrar a passagem de arrays e elementos de array individuais por referência. Utilizamos o algoritmo de **classificação por seleção**, que é um algoritmo de classificação fácil de programar, mas, infelizmente, ineficiente. A primeira iteração do algoritmo seleciona o menor elemento no array e o troca pelo primeiro elemento. A segunda iteração seleciona o segundo menor elemento (que é o menor dos elementos restantes) e o troca pelo segundo elemento. O algoritmo continua até que a última iteração selecione o segundo maior elemento e permute-o pelo penúltimo índice, deixando o maior elemento no último índice. Depois da *i*-ésima iteração, os *i* menores itens do array serão classificados pela ordem crescente nos primeiros *i* elementos do array.

Como um exemplo, considere o array

34 56 4 10 77 51 93 30 5 52

Um programa que implementa classificação por seleção primeiro determina o menor elemento (4) desse array, que está contido no elemento 2. O programa troca o 4 pelo elemento 0 (34), resultando em

4 56 **34** 10 77 51 93 30 5 52

[*Nota:* Utilizamos negrito para destacar os valores que foram trocados.] O programa então determina o menor valor dos elementos restantes (todos os elementos exceto 4), que é 5, contido no elemento 8. O programa troca o 5 pelo elemento 1 (56), resultando em

4 **5** 34 10 77 51 93 30 **56** 52

Na terceira iteração, o programa determina o próximo menor valor (10) e o troca pelo elemento 2 (34).

4 5 **10** **34** 77 51 93 30 56 52

O processo continua até que o array seja completamente classificado.

4 5 10 30 34 51 52 56 77 93

```
1  // Figura 8.14: fig08_14.cpp
2  // Tentando modificar um ponteiro constante para dados constantes.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 5, y;
10
11    // ptr é um ponteiro constante para um inteiro constante.
12    // ptr sempre aponta para a mesma posição; o inteiro
13    // nessa posição não pode ser modificado.
14    const int *const ptr = &x;
15
16    cout << *ptr << endl;
17
18    *ptr = 7; // erro: *ptr é const; não é possível atribuir novo valor
19    ptr = &y; // error: ptr é const; não é possível atribuir endereço
20    return 0; // indica terminação bem-sucedida
21 } // fim de main
```

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphtp5e_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C2166:
   l-value specifies const object
c:\cpphtp5e_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C2166:
   l-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'
```

Figura 8.14 Tentando modificar um ponteiro constante para dados constantes.

Observe que, depois da primeira iteração, o menor elemento estará na primeira posição. Depois da segunda iteração, os dois menores elementos estarão na ordem nas duas primeiras posições. Depois da terceira iteração, os três menores elementos estarão na ordem nas três primeiras posições.

A Figura 8.15 implementa a classificação por seleção utilizando duas funções — `selectionSort` e `swap`. A função `selectionSort` (linhas 36–53) classifica o array. A linha 38 declara a variável `smallest`, que armazenará o índice do menor elemento no array restante. As linhas 41–52 iteram `size - 1` vezes. A linha 43 configura o índice do menor elemento para o índice atual. As linhas 46–49 iteram pelos elementos restantes no array. Para cada um desses elementos, a linha 48 compara seu valor com o valor do menor elemento. Se o elemento atual for menor que o menor elemento, a linha 49 atribui o índice do elemento atual a `smallest`. Quando esse loop termina, `smallest` conterá o índice do menor elemento no array restante. A linha 51 chama a função `swap` (linhas 57–62) para colocar o menor elemento restante no próximo local no array (isto é, troca os elementos do array `array[ i ]` e `array[ smallest ]`).

Façamos agora um exame mais minucioso na função `swap`. Lembre-se de que o C++ impõe o ocultamento de informações entre funções, por isso `swap` não tem acesso aos elementos de array individuais em `selectionSort`. Como `selectionSort` quer que `swap` tenha acesso aos elementos do array a serem trocados, `selectionSort` passa cada um desses elementos para `swap` por referência — o endereço de cada elemento de array é passado explicitamente. Embora os arrays inteiros sejam passados por referência, os elementos do array individuais são escalares e comumente passados por valor. Portanto, `selectionSort` utiliza o operador de endereço (`&`) em cada elemento

do array na chamada a `swap` (linha 51) para produzir a passagem por referência. A função `swap` (linhas 57–62) recebe `&array[ i ]` na variável ponteiro `element1Ptr`. O ocultamento de informações impede que `swap` 'saiba' o nome `array[ i ]`, mas `swap` pode utilizar `*element1Ptr` como um sinônimo de `array[ i ]`. Portanto, quando `swap` referencia `*element1Ptr`, ele na realidade está referenciando `array[ i ]` em `selectionSort`. De maneira semelhante, quando `swap` referencia `*element2Ptr`, na realidade está referenciando `array[ smallest ]` em `selectionSort`.

Ainda que `swap` não tenha permissão de utilizar as instruções

```
hold = array[ i ];
array[ i ] = array[ smallest ];
array[ smallest ] = hold;
```

precisamente o mesmo efeito é alcançado por

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

na função `swap` da Figura 8.15.

```
1  // Figura 8.15: fig08_15.cpp
2  // Este programa coloca valores em um array, classifica os valores em
3  // ordem crescente e imprime o array resultante.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 void selectionSort( int * const, const int ); // protótipo
12 void swap( int * const, int * const ); // protótipo
13
14 int main()
15 {
16    const int arraySize = 10;
17    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    cout << "Data items in original order\n";
20
21    for ( int i = 0; i < arraySize; i++ )
22       cout << setw( 4 ) << a[ i ];
23
24    selectionSort( a, arraySize ); // classifica o array
25
26    cout << "\nData items in ascending order\n";
27
28    for ( int j = 0; j < arraySize; j++ )
29       cout << setw( 4 ) << a[ j ];
30
31    cout << endl;
32    return 0; // indica terminação bem-sucedida
33 } // fim de main
34
35 // função para classificar um array
36 void selectionSort( int * const array, const int size )
37 {
38    int smallest; // índice do menor elemento
39
```

Figura 8.15 Classificação por seleção com passagem por referência. (continua)

```
40     // itera sobre size - 1 elementos
41     for ( int i = 0; i < size - 1; i++ )
42     {
43        smallest = i; // primeiro índice do array remanescente
44
45        // faz um loop para localizar o índice do menor elemento
46        for ( int index = i + 1; index < size; index++ )
47
48           if ( array[ index ] < array[ smallest ] )
49              smallest = index;
50
51        swap( &array[ i ], &array[ smallest ] );
52     } // fim do for
53  } // fim da função selectionSort
54
55  // troca os valores nas posições da memória para as quais
56  // element1Ptr e element2Ptr apontem
57  void swap( int * const element1Ptr, int * const element2Ptr )
58  {
59     int hold = *element1Ptr;
60     *element1Ptr = *element2Ptr;
61     *element2Ptr = hold;
62  } // fim da função swap
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Figura 8.15 Classificação por seleção com passagem por referência. (continuação)

Vários recursos da função `selectionSort` devem ser observados. O cabeçalho de função (linha 36) declara `array` como `int * const array`, em vez de `int array[]`, para indicar que a função `selectionSort` recebe um array unidimensional como um argumento. Tanto o ponteiro do parâmetro `array` como o parâmetro `size` são declarados como `const` para impor o princípio do menor privilégio. Embora o parâmetro `size` receba uma cópia de um valor em `main` e modificar a cópia não possa alterar o valor em `main`, `selectionSort` não precisa alterar `size` para realizar sua tarefa — o tamanho de array permanece fixo durante a execução de `selectionSort`. Portanto, `size` é declarado como `const` para assegurar que ele não seja modificado. Se o tamanho do array fosse modificado durante o processo de classificação, o algoritmo de classificação não teria executado corretamente.

Observe que a função `selectionSort` recebe o tamanho do array como um parâmetro, porque a função deve ter essas informações para classificar o array. Quando um array baseado em ponteiro é passado para uma função, somente o endereço de memória do primeiro elemento do array é recebido pela função; o tamanho do array deve ser passado separadamente para a função.

Definindo a função `selectionSort` para receber o tamanho de array como um parâmetro, permitimos que a função seja utilizada por qualquer programa que classifique arrays `int` unidimensionais de tamanho arbitrário. O tamanho do array poderia ter sido programado diretamente na função, mas isso restringiria a função a processar um array de um tamanho específico e reduziria a reusabilidade da função — somente os programas que processam arrays unidimensionais `int` de um tamanho específico codificados diretamente [‘*hard coded*’] na função poderiam utilizar a função.

Observação de engenharia de software 8.4

Ao passar um array para uma função, passe também o tamanho do array (em vez de construir na função o conhecimento sobre o tamanho de array). Isso torna a função mais reutilizável.

8.7 Operadores sizeof

O C++ fornece o operador unário **`sizeof`** para determinar o tamanho de um array (ou de qualquer outro tipo de dados, variável ou constante) em bytes durante a compilação de programa. Quando aplicado ao nome de um array, como na Figura 8.16 (linha 14), o operador `sizeof` retorna o número total de bytes no array como um valor de tipo `size_t` (um alias para `unsigned int` na maioria dos

compiladores). Observe que esse é diferente do `size` de um `vector< int >`, por exemplo, que é o número de elementos do tipo inteiro no `vector`. O computador que utilizamos para compilar esse programa armazena variáveis do tipo `double` em 8 bytes de memória, e `array` é declarado para ter 20 elementos (linha 12), portanto `array` utiliza 160 bytes na memória. Quando aplicado a um parâmetro de ponteiro (linha 24) em uma função que recebe um array como um argumento, o operador `sizeof` retorna o tamanho do ponteiro em bytes (4), não o tamanho do array.

Erro comum de programação 8.7

Utilizar o operador `sizeof` em uma função para localizar o tamanho em bytes de um parâmetro de array resulta no tamanho em bytes de um ponteiro, não no tamanho em bytes do array.

[*Nota:* Quando o compilador Borland C++ é utilizado para compilar a Figura 8.16, o compilador gera a mensagem de advertência `"Parameter 'ptr' is never used in function getSize(double *)"`. Esse aviso ocorre porque `sizeof` é na realidade um operador em tempo de compilação; portanto, a variável `ptr` não é utilizada no corpo da função em tempo de execução. Muitos compiladores emitem avisos como esse para que se saiba que uma variável não está sendo utilizada de modo que você possa removê-la de seu código ou modificar seu código para utilizar a variável adequadamente. Mensagens semelhantes ocorrem na Figura 8.17 com vários compiladores.]

O número de elementos em um array também pode ser determinado utilizando os resultados de duas operações `sizeof`. Por exemplo, considere a seguinte declaração de array:

```
double realArray[ 22 ];
```

Se variáveis do tipo de dados `double` forem armazenadas em oito bytes de memória, o array `realArray` conterá um total de 176 bytes. Para determinar o número de elementos no array, a seguinte expressão pode ser utilizada:

```
sizeof realArray / sizeof( double ) // calcula o número de elementos
```

A expressão determina o número de bytes no array `realArray` (176) e divide esse valor pelo número de bytes utilizados na memória para armazenar um valor `double` (8); o resultado é o número de elementos em `realArray` (22).

```
1  // Figura 8.16: fig08_16.cpp
2  // Operador Sizeof quando utilizado em um nome de array
3  // retorna o número de bytes no array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  size_t getSize( double * ); // protótipo
9
10 int main()
11 {
12    double array[ 20 ]; // 20 doubles; o que ocupa 160 bytes em nosso sistema
13
14    cout << "The number of bytes in the array is " << sizeof( array );
15
16    cout << "\nThe number of bytes returned by getSize is "
17       << getSize( array ) << endl;
18    return 0; // indica terminação bem-sucedida
19 } // fim de main
20
21 // retorna o tamanho de ptr
22 size_t getSize( double *ptr )
23 {
24    return sizeof( ptr );
25 } // fim da função getSize
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Figura 8.16 O operador `sizeof` quando aplicado a um nome de array retorna o número de bytes no array.

Determinando os tamanhos dos tipos fundamentais, um array e um ponteiro

O programa da Figura 8.17 utiliza o operador `sizeof` para calcular o número de bytes utilizados para armazenar a maioria dos tipos de dados padrão. Note que, na saída, os tipos `double` e `long double` têm o mesmo tamanho. Os tipos podem ter tamanhos diferentes com base no sistema em que o programa é executado. Em outro sistema, por exemplo, `double` e `long double` podem ser definidos com tamanhos diferentes.

```
1  // Figura 8.17: fig08_17.cpp
2  // Demonstrando o operador sizeof.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     char c; // variável de tipo char
10    short s; // variável de tipo short
11    int i; // variável de tipo int
12    long l; // variável de tipo long
13    float f; // variável de tipo float
14    double d; // variável de tipo double
15    long double ld; // variável de tipo long double
16    int array[ 20 ]; // array de int
17    int *ptr = array; // variável de tipo int *
18
19    cout << "sizeof c = " << sizeof c
20       << "\tsizeof(char) = " << sizeof( char )
21       << "\nsizeof s = " << sizeof s
22       << "\tsizeof(short) = " << sizeof( short )
23       << "\nsizeof i = " << sizeof i
24       << "\tsizeof(int) = " << sizeof( int )
25       << "\nsizeof l = " << sizeof l
26       << "\tsizeof(long) = " << sizeof( long )
27       << "\nsizeof f = " << sizeof f
28       << "\tsizeof(float) = " << sizeof( float )
29       << "\nsizeof d = " << sizeof d
30       << "\tsizeof(double) = " << sizeof( double )
31       << "\nsizeof ld = " << sizeof ld
32       << "\tsizeof(long double) = " << sizeof( long double )
33       << "\nsizeof array = " << sizeof array
34       << "\nsizeof ptr = " << sizeof ptr << endl;
35    return 0; // indica terminação bem-sucedida
36 } // fim de main
```

```
sizeof c = 1     sizeof(char) = 1
sizeof s = 2     sizeof(short) = 2
sizeof i = 4     sizeof(int) = 4
sizeof l = 4     sizeof(long) = 4
sizeof f = 4     sizeof(float) = 4
sizeof d = 8     sizeof(double) = 8
sizeof ld = 8    sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Figura 8.17 Operador `sizeof` utilizado para determinar tamanhos de tipo de dados padrão.

Dica de portabilidade 8.3

O número de bytes utilizado para armazenar um tipo de dados particular pode variar entre sistemas. Ao escrever programas que dependem de tamanhos de tipo de dados e que executarão em vários sistemas de computador, utilize `sizeof` *para determinar o número de bytes utilizados para armazenar os tipos de dados.*

O operador `sizeof` pode ser aplicado a qualquer nome de variável, nome de tipo ou valor constante. Quando `sizeof` é aplicado a um nome de variável (que não seja um nome de array) ou um valor constante, o número de bytes utilizados para armazenar o tipo de variável específica ou constante é retornado. Observe que os parênteses utilizados com `sizeof` só são requeridos se o nome de um tipo (por exemplo, `int`) for fornecido como seu operando. Os parênteses utilizados com `sizeof` não são requeridos quando o operando de `sizeof` for um nome de variável ou constante. Lembre-se de que `sizeof` é um operador, não uma função e que ele tem seu efeito em tempo de compilação, não em tempo de execução.

Erro comum de programação 8.8

Omitir os parênteses em uma operação `sizeof` *quando o operando é um nome de tipo é um erro de compilação.*

Dica de desempenho 8.2

Como `sizeof` *é um operador unário em tempo de compilação, não um operador em tempo de execução, utilizar* `sizeof` *não influencia negativamente o desempenho da execução.*

Dica de prevenção de erro 8.3

Para evitar erros associados com a omissão dos parênteses em torno do operando do operador `sizeof`*, muitos programadores colocam cada operando* `sizeof` *entre parênteses.*

8.8 Expressões e aritmética de ponteiro

Os ponteiros são operandos válidos em expressões aritméticas, expressões de atribuição e expressões de comparação. Entretanto, nem todos os operadores normalmente utilizados nessas expressões são válidos com variáveis ponteiro. Esta seção descreve os operadores que podem ter ponteiros como operandos e como esses operadores são utilizados com ponteiros.

Várias operações aritméticas podem ser realizadas em ponteiros. Um ponteiro pode ser incrementado (`++`) ou decrementado (`--`), um inteiro pode ser adicionado a um ponteiro (`+` ou `+=`), um inteiro pode ser subtraído de um ponteiro (`-` ou `-=`) ou um ponteiro pode ser subtraído de outro.

Suponha que o array `int v[ 5 ]` tenha sido declarado e que seu primeiro elemento esteja na posição da memória 3000. Suponha que o ponteiro `vPtr` tenha sido inicializado para apontar para `v[ 0 ]` (isto é, o valor de `vPtr` é 3000). A Figura 8.18 diagrama essa situação para uma máquina com inteiros de quatro bytes. Observe que `vPtr` pode ser inicializado para apontar para o array `v` com qualquer uma das seguintes instruções (porque o nome de um array é equivalente ao endereço de seu primeiro elemento):

```
int *vPtr = v;
int *vPtr = &v[ 0 ];
```

Dica de portabilidade 8.4

A maioria dos computadores hoje tem inteiros de dois bytes ou quatro bytes. Algumas máquinas mais novas utilizam inteiros de oito bytes. Como os resultados da aritmética de ponteiros dependem do tamanho dos objetos para os quais um ponteiro aponta, a aritmética de ponteiros é dependente da máquina.

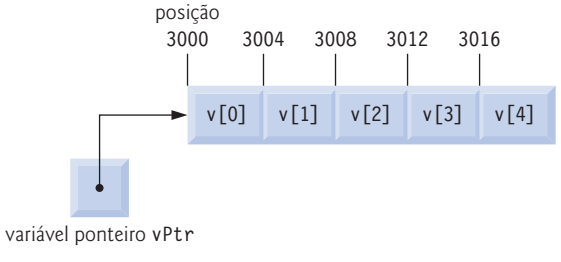

Figura 8.18 O array `v` e uma variável ponteiro `vPtr` que aponta para `v`.

Na aritmética convencional, a adição `3000 + 2` resulta no valor `3002`. Normalmente não é esse o caso com a aritmética de ponteiros. Quando um inteiro é adicionado a, ou subtraído de, um ponteiro, o ponteiro simplesmente não é incrementado ou decrementado por esse inteiro, mas por esse inteiro vezes o tamanho do objeto que o ponteiro referencia. O número de bytes depende do tipo de dados do objeto. Por exemplo, a instrução

```
vPtr += 2;
```

produziria `3008 (3000 + 2 * 4)`, supondo que um `int` é armazenado em quatro bytes de memória. No array `v`, `vPtr` agora apontaria para `v[ 2 ]` (Figura 8.19). Se um inteiro fosse armazenado em dois bytes de memória, então o cálculo precedente resultaria na posição da memória `3004 (3000 + 2 * 2)`. Se o array fosse de um tipo de dados diferente, a instrução precedente incrementaria o ponteiro por duas vezes o número de bytes que ele aceita para armazenar um objeto desse tipo de dados. Ao realizar a aritmética de ponteiros em um array de caracteres, os resultados serão consistentes com a aritmética regular, porque cada caractere tem um byte de comprimento.

Se `vPtr` tiver sido incrementado por `3016`, que aponta para `v[4]`, a instrução

```
vPtr -= 4;
```

configuraria `vPtr` de volta para `3000` — o começo do array. Se um ponteiro estiver sendo incrementado ou decrementado por um, os operadores de incremento (`++`) e decremento (`--`) podem ser utilizados. Cada uma das instruções

```
++vPtr;
vPtr++;
```

incrementa o ponteiro para apontar para o próximo elemento do array. Cada uma das instruções

```
--vPtr;
vPtr--;
```

decrementa o ponteiro para apontar para o elemento anterior do array.

As variáveis ponteiro que apontam para o mesmo array podem ser subtraídas uma da outra. Por exemplo, se `vPtr` contiver a posição `3000` e `v2Ptr` contiver o endereço `3008`, a instrução

```
x = v2Ptr - vPtr;
```

atribuiria a `x` o número de elementos do array de `vPtr` a `v2Ptr` — nesse caso, `2`. A aritmética de ponteiros não faz sentido a menos que realizada em um ponteiro que aponta para um array. Não podemos pressupor que duas variáveis do mesmo tipo estejam armazenadas contiguamente na memória a menos que elas sejam elementos adjacentes de um array.

Erro comum de programação 8.9

Utilizar aritmética de ponteiros em um ponteiro que não referencia um array de valores é um erro de lógica.

Erro comum de programação 8.10

Subtrair ou comparar dois ponteiros que não referenciam elementos do mesmo array é um erro de lógica.

Erro comum de programação 8.11

Utilizar aritmética de ponteiros para incrementar ou decrementar um ponteiro de modo que esse ponteiro referencie um elemento depois do fim, ou antes do começo do array, é normalmente um erro de lógica.

Um ponteiro pode ser atribuído a outro ponteiro se ambos forem do mesmo tipo. Caso contrário, um operador de coerção deve ser utilizado para converter o valor do ponteiro à direita da atribuição no tipo de ponteiro à esquerda da atribuição. A exceção a essa regra é o ponteiro para `void` (isto é, `void *`), que é um ponteiro genérico capaz de representar qualquer tipo de ponteiro. Um ponteiro de

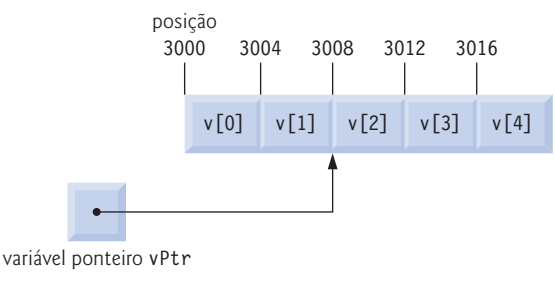

Figura 8.19 Ponteiro `vPtr` depois da aritmética de ponteiros.

tipo `void *` sem coerção pode ser atribuído a todos os tipos de ponteiro. Entretanto, um ponteiro do tipo `void *` não pode ser atribuído diretamente a um ponteiro de outro tipo — o ponteiro do tipo `void *` deve primeiro sofrer coerção para o tipo de ponteiro adequado.

Observação de engenharia de software 8.5

Os argumentos de ponteiro não-constantes podem ser passados para parâmetros de ponteiro constantes. Isso é útil quando o corpo de um programa utiliza um ponteiro não-constante para acessar dados, mas não quer que os dados sejam modificados por uma chamada de função no corpo do programa.

Um ponteiro `void *` não pode ser desreferenciado. Por exemplo, o compilador 'sabe' que um ponteiro para `int` referencia quatro bytes de memória em uma máquina com inteiros de quatro bytes, mas um ponteiro para `void` contém simplesmente um endereço de memória para um tipo de dados desconhecido — o número preciso de bytes que o ponteiro referencia e o tipo dos dados não são conhecidos pelo compilador. O compilador deve conhecer o tipo de dados para determinar o número de bytes a ser desreferenciado por um ponteiro particular — para um ponteiro para `void`, esse número de bytes não pode ser determinado a partir do tipo.

Erro comum de programação 8.12

Atribuir um ponteiro de um tipo a um ponteiro de outro (diferente de `void *`) sem fazer coerção do primeiro ponteiro para o tipo do segundo ponteiro é um erro de compilação.

Erro comum de programação 8.13

Todas as operações em um ponteiro `void *` são erros de compilação, exceto comparar ponteiros `void *` com outros ponteiros, fazendo coerção dos ponteiros `void *` para tipos de ponteiros válidos e atribuindo endereços a ponteiros `void *`.

Os ponteiros podem ser comparados utilizando operadores de igualdade e operadores relacionais. As comparações que utilizam operadores relacionais não têm sentido a menos que os ponteiros apontem para membros do mesmo array. As comparações de ponteiro comparam os endereços armazenados nos ponteiros. Uma comparação de dois ponteiros que apontam para o mesmo array poderia mostrar, por exemplo, que um ponteiro aponta para um elemento de número mais alto no array do que o outro ponteiro. Uma utilização comum da comparação de ponteiro é determinar se um ponteiro é 0 (isto é, o ponteiro é um ponteiro nulo — ele não aponta para nada).

8.9 Relacionamento entre ponteiros e arrays

Os arrays e os ponteiros estão intimamente relacionados em C++ e podem ser utilizados quase intercambiavelmente. É possível pensar em um nome de array como um ponteiro constante. Os ponteiros podem ser utilizados para fazer qualquer operação que envolva subscrito de array.

Suponha as seguintes declarações:

```
int b[ 5 ]; // cria array int b de 5 elementos
int *bPtr; // cria ponteiro int bPtr
```

Como o nome de array (sem subscrito) é um ponteiro (constante) para o primeiro elemento do array, podemos configurar `bPtr` como o endereço do primeiro elemento no array `b` com a instrução

```
bPtr = b; // atribui o endereço de array b ao bPtr
```

Isso é equivalente a aceitar o endereço do primeiro elemento do array como mostrado a seguir:

```
bPtr = &b[ 0 ]; // também atribui o endereço do array b ao bPtr
```

O elemento do array `b[ 3 ]` pode ser alternativamente referenciado com a expressão de ponteiro

```
*( bPtr + 3 )
```

O `3` na expressão precedente é o **deslocamento** para o ponteiro. Quando o ponteiro aponta para o começo de um array, o deslocamento indica que elemento do array deve ser referenciado e o valor de deslocamento é idêntico ao subscrito de array. A notação anterior é referida como **notação de ponteiro/deslocamento**. Os parênteses são necessários, porque a precedência de `*` é mais alta que a precedência de `+`. Sem os parênteses, a expressão acima adicionaria `3` ao valor de `*bPtr` (isto é, `3` seriam adicionados a `b[ 0 ]`, supondo que `bPtr` aponta para o início do array). Assim como o elemento do array pode ser referenciado com uma expressão de ponteiro, o endereço

```
&b[ 3 ]
```

pode ser escrito com a expressão de ponteiro

```
bPtr + 3
```

O nome do array pode ser tratado como um ponteiro e utilizado na aritmética de ponteiros. Por exemplo, a expressão

```
*( b + 3 )
```

também referencia o elemento `b[ 3 ]` do array. Em geral, todas as expressões de array subscritas podem ser escritas com um ponteiro e um deslocamento. Nesse caso, a notação de ponteiro/deslocamento foi utilizada com o nome do array como um ponteiro. Observe que a expressão precedente não modifica o nome do array de nenhuma maneira; `b` ainda aponta para o primeiro elemento no array.

Os ponteiros podem ser indexados com subscritos exatamente como arrays. Por exemplo, a expressão

```
bPtr[ 1 ]
```

referencia o elemento do array `b[ 1 ]`; essa expressão utiliza **notação de ponteiro/subscrito**.

Lembre-se de que um nome do array é um ponteiro constante; ele sempre aponta para o começo do array. Portanto, a expressão

```
b += 3
```

causa um erro de compilação, porque tenta modificar o valor do nome do array (uma constante) com a aritmética de ponteiros.

Erro comum de programação 8.14

Embora os nomes de array sejam ponteiros para o começo do array e os ponteiros possam ser modificados em expressões aritméticas, os nomes de array não podem ser modificados em expressões aritméticas, porque são ponteiros constantes.

Boa prática de programação 8.2

Por questão de clareza, use notação de array em vez de notação de ponteiro ao manipular arrays.

A Figura 8.20 utiliza as quatro notações discutidas nesta seção para referenciar elementos de um array — a notação de subscrito de array, a notação de ponteiro/deslocamento com o nome de array como um ponteiro, a notação de subscrito de ponteiro e a notação de ponteiro/deslocamento com um ponteiro — para realizar a mesma tarefa, isto é, imprimir os quatro elementos do array de inteiro `b`.

Para ilustrar ainda mais a intercambialidade de arrays e ponteiros, vejamos as duas funções de copiar strings — `copy1` e `copy2` — no programa da Figura 8.21. Ambas as funções copiam uma string em um array de caracteres. Depois de uma comparação dos protótipos de função para `copy1` e `copy2`, as funções parecem idênticas (por causa da intercambialidade de arrays e ponteiros). Essas funções realizam a mesma tarefa, mas são implementadas de modo diferente.

```
1  // Figura 8.20: fig08_20.cpp
2  // Utilizando notações de subscrito e de ponteiro com arrays.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int b[] = { 10, 20, 30, 40 }; // cria o array b de 4 elementos
10    int *bPtr = b; // configura bPtr para apontar para o array b
11
12    // gera saída do array b utilizando notação de subscrito de array
13    cout << "Array b printed with:\n\nArray subscript notation\n";
14
15    for ( int i = 0; i < 4; i++ )
16       cout << "b[" << i << "] = " << b[ i ] << '\n';
17
18    // gera saída do array b utilizando a notação de nome de array e a de ponteiro/deslocamento
19    cout << "\nPointer/offset notation where "
20       << "the pointer is the array name\n";
21
22    for ( int offset1 = 0; offset1 < 4; offset1++ )
23       cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
24
25    // gera saída do array b utilizando bPtr e notação de subscrito de array
26    cout << "\nPointer subscript notation\n";
27
```

Figura 8.20 Referenciando elementos do array com o nome do array e com ponteiros. (continua)

```
28     for ( int j = 0; j < 4; j++ )
29        cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
30
31     cout << "\nPointer/offset notation\n";
32
33     // gera saída do array b utilizando bPtr e notação de ponteiro/deslocamento
34     for ( int offset2 = 0; offset2 < 4; offset2++ )
35        cout << "*(bPtr + " << offset2 << ") = "
36           << *( bPtr + offset2 ) << '\n';
37
38     return 0; // indica terminação bem-sucedida
39  } // fim de main
```

```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) + 40
```

Figura 8.20 Referenciando elementos do array com o nome do array e com ponteiros. (continuação)

A função `copy1` (linhas 26–31) utiliza a notação de subscrito de array para copiar a string em `s2` para o array de caracteres `s1`. A função declara uma variável contadora de inteiros `i` para utilizar como o subscrito de array. O cabeçalho da instrução `for` (linha 29) realiza toda a operação de cópia — seu corpo é a instrução vazia. O cabeçalho especifica que `i` é inicializado como zero e incrementado por um a cada iteração do loop. A condição no `for`, `( s1[ i ] = s2[ i ] ) != '\0'`, realiza a operação de cópia de caractere por caractere de `s2` para `s1`. Quando o caractere nulo é encontrado em `s2`, ele é atribuído a `s1` e o loop termina, porque o caractere nulo é igual a `'\0'`. Lembre-se de que o valor de uma instrução de atribuição é o valor atribuído a seu operando esquerdo.

A função `copy2` (linhas 34–39) utiliza ponteiros e aritmética de ponteiros para copiar a string em `s2` para o array de caracteres `s1`. Novamente, o cabeçalho da instrução `for` (linha 37) realiza a operação inteira de cópia. O cabeçalho não inclui nenhuma variável de inicialização. Como na função `copy1`, a condição `( *s1 = *s2 ) != '\0'` realiza a operação de cópia. O ponteiro `s2` é desreferenciado e o caractere resultante é atribuído ao ponteiro desreferenciado `s1`. Depois da atribuição na condição, o loop incrementa ambos os ponteiros, então eles apontam para o próximo elemento de array `s1` e o próximo caractere de string `s2`, respectivamente. Quando o loop encontra o caractere nulo em `s2`, o caractere nulo é atribuído ao ponteiro desreferenciado `s1` e o loop termina. Observe que a 'parte de incremento' dessa instrução `for` tem duas expressões de incremento separadas por um operador vírgula.

O primeiro argumento para `copy1` e `copy2` deve ser um array grande o bastante para armazenar a string no segundo argumento. Caso contrário, pode ocorrer um erro quando for feita uma tentativa de escrever em uma posição da memória além dos limites do array (lembre-se de que ao utilizar arrays baseados em ponteiro, não há nenhuma verificação de limite 'integrada'). Além disso, observe que

```
1  // Figura 8.21: fig08_21.cpp
2  // Copiando uma string utilizando a notação de array e a notação de ponteiro.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void copy1( char *, const char * ); // protótipo
8  void copy2( char *, const char * ); // protótipo
9
10 int main()
11 {
12    char string1[ 10 ];
13    char *string2 = "Hello";
14    char string3[ 10 ];
15    char string4[] = "Good Bye";
16
17    copy1( string1, string2 ); // copia string2 para string1
18    cout << "string1 = " << string1 << endl;
19
20    copy2( string3, string4 ); // copia string4 para string3
21    cout << "string3 = " << string3 << endl;
22    return 0; // indica terminação bem-sucedida
23 } // fim de main
24
25 // copia s2 para s1 utilizando notação de array
26 void copy1( char * s1, const char * s2 )
27 {
28    // a cópia ocorre no cabeçalho do for
29    for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
30       ; // não faz nada no corpo
31 } // fim da função copy1
32
33 // copia s2 para s1 utilizando notação de ponteiro
34 void copy2( char *s1, const char *s2 )
35 {
36    // a cópia ocorre no cabeçalho do for
37    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
38       ; // não faz nada no corpo
39 } // fim da função copy2
```

```
string1 = Hello
string3 = Good Bye
```

Figura 8.21 Cópia de strings utilizando a notação de array e a notação de ponteiro.

o segundo parâmetro de cada função é declarado como `const char *` (um ponteiro para um caractere constante — isto é, uma string constante). Em ambas as funções, o segundo argumento é copiado para o primeiro argumento — os caracteres são copiados um a um a partir do segundo argumento, mas os caracteres nunca são modificados. Portanto, o segundo parâmetro é declarado para apontar para um valor constante para impor o princípio do menor privilégio —nenhuma função precisa modificar o segundo argumento, então nenhuma função recebe a permissão de modificar o segundo argumento.

8.10 Arrays de ponteiros

Os arrays podem conter ponteiros. Uma utilização comum dessa estrutura de dados é formar um array de strings baseadas em ponteiro, referido simplesmente como um **array de string**. Toda entrada no array é uma string, mas em C++ uma string é essencialmente um ponteiro para seu primeiro caractere, então cada entrada em um array de strings é simplesmente um ponteiro para o primeiro caractere de uma string. Considere a declaração de array de string `suit` que poderia ser útil na representação de um baralho:

```
const char *suit[ 4 ] =
   { "Hearts", "Diamonds", "Clubs", "Spades" };
```

A parte `suit[4]` da declaração indica um array de quatro elementos. A parte `const char *` da declaração indica que cada elemento de array `suit` é do tipo 'ponteiro para dados `char` constantes'. Os quatro valores a ser colocados no array são `"Hearts"`, `"Diamonds"`, `"Clubs"` e `"Spades"`. Cada um é armazenado na memória como uma string de caracteres terminada por caractere nulo, que é um caractere mais longo que o número de caracteres entre aspas. As quatro strings são os caracteres de tamanho sete, nove, seis e sete (incluindo seus caracteres nulos de terminação), respectivamente. Embora pareça que essas strings estão sendo colocadas no array `suit`, somente os ponteiros são realmente armazenados no array, como mostrado na Figura 8.22. Cada ponteiro aponta para o primeiro caractere de sua string correspondente. Portanto, mesmo que o array `suit` tenha tamanho fixo, ele fornece acesso a strings de caractere de qualquer comprimento. Essa flexibilidade é um exemplo das poderosas capacidades da estrutura de dados do C++.

As strings de naipe poderiam ser colocadas em um array bidimensional, em que cada linha representa um naipe e cada coluna representa uma das letras de um nome de naipe. Essa estrutura de dados deve ter um número fixo de colunas por linha, e esse número deve ser tão grande quanto a maior string. Portanto, uma quantidade considerável de memória é desperdiçada quando armazenamos um grande número de strings, das quais a maioria é mais curta que a string mais longa. Utilizamos arrays de strings para ajudar a representar um baralho na próxima seção.

Os arrays de string são comumente utilizados com **argumentos de linha de comando** que são passados para a função `main` quando um programa inicia a execução. Esses argumentos seguem o nome de programa quando um programa é executado da linha de comando. Uma utilização típica de argumentos de linha de comando é passar opções para um programa. Por exemplo, a partir da linha de comando em um computador Windows, o usuário pode digitar

```
dir /P
```

para listar o conteúdo do diretório atual e pausar depois de cada tela das informações. Quando o comando `dir` executa, a opção `/P` é passada para `dir` como um argumento de linha de comando. Esses argumentos são colocados em um array de string que `main` recebe como um argumento. Discutimos os argumentos de linha de comando no Apêndice E, "Tópicos sobre o código C legado".

8.11 Estudo de caso: simulação de embaralhamento e distribuição de cartas

Esta seção utiliza a geração de números aleatórios para desenvolver um programa de simulação de embaralhamento e distribuição de cartas. Esse programa pode então ser utilizado como uma base para implementar programas que reproduzem jogos de cartas específicos. Para revelar alguns problemas de desempenho sutis, utilizamos intencionalmente algoritmos de embaralhamento e distribuição subótimos. Nos exercícios, desenvolvemos algoritmos mais eficientes.

Utilizando a abordagem de refinamento passo a passo de cima para baixo, desenvolvemos um programa que irá embaralhar um baralho de 52 cartas e, em seguida, negociar cada uma delas. A abordagem de cima para baixo é particularmente útil para atacar problemas maiores e mais complexos que vimos nos capítulos anteriores.

Utilizamos um array bidimensional 4 por 13 `deck` para representar o baralho (Figura 8.23). As linhas correspondem aos naipes — a linha 0 corresponde ao naipe de copas, a linha 1 ao de ouros, a linha 2 ao de paus e a linha 3 ao de espadas. As colunas correspondem aos valores de face das cartas — as colunas de 0 a 9 correspondem às faces de ás a 10, respectivamente, e as colunas de 10 a 12 correspondem ao valete, dama e rei, respectivamente. Carregaremos o array de strings `suit` com strings de caracteres que representam os quatro naipes (como na Figura 8.22) e o array de strings `face` com strings de caracteres que representam os 13 valores de face.

Esse baralho simulado pode ser embaralhado como mostrado a seguir. Primeiro o array `deck` é inicializado como zeros. Em seguida, uma `row` (0–3) e uma `column` (0–12) são escolhidas aleatoriamente. O número 1 é inserido no elemento do array `deck[ row ][ column ]` para indicar que essa será a primeira carta distribuída do conjunto de cartas embaralhadas. Esse processo continua com os números 2, 3, ..., 52 sendo inseridos aleatoriamente no array `deck` para indicar as cartas que devem ser colocadas em segundo lugar, terceiro lugar, ... e 52º lugar no baralho embaralhado. À medida que o array `deck` começa a ser preenchido com os números de carta, é possível que uma carta seja selecionada duas vezes (isto é, `deck[ row ][ column ]` será não-zero quando for selecionada). Essa seleção simplesmente é ignorada, e outras `row`s e `column`s são repetidamente escolhidas de forma aleatória até que uma carta não selecionada seja localizada. Por fim, os números 1 a 52 ocuparão os 52 slots do array `deck`. Nesse ponto, o baralho está completamente embaralhado.

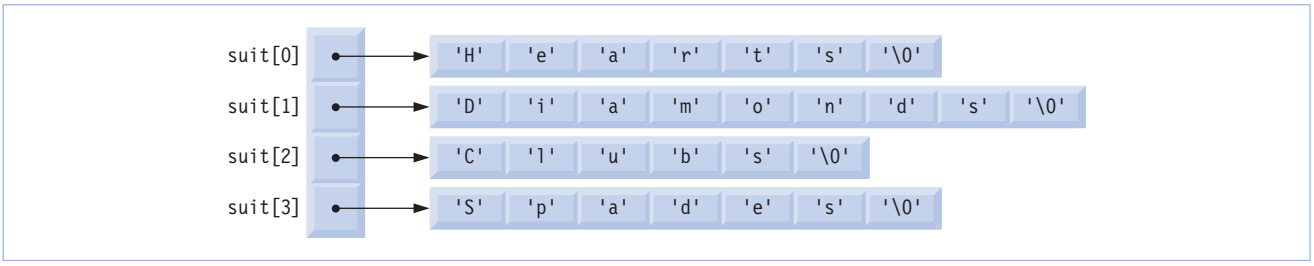

Figura 8.22 Representação gráfica do array `suit`.

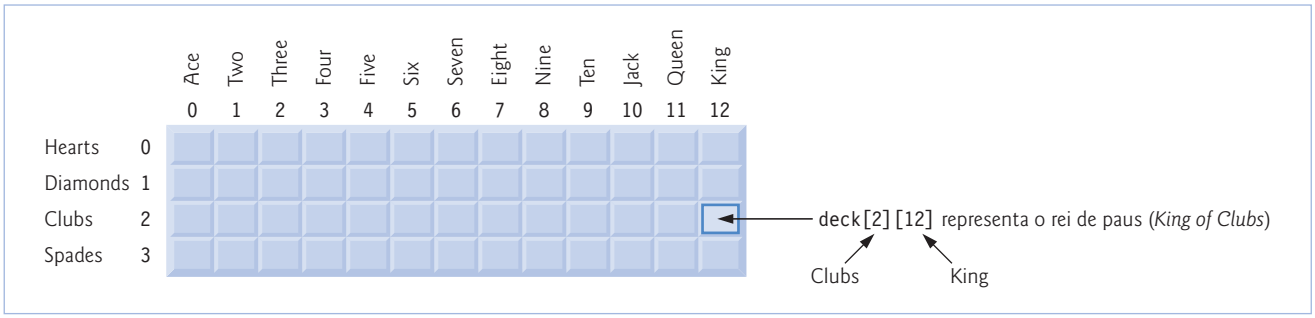

Figura 8.23 A representação de array bidimensional de um baralho.

Esse algoritmo de embaralhamento poderia executar por um período indefinidamente longo se as cartas que já foram embaralhadas fossem repetidamente selecionadas a esmo. Esse fenômeno é conhecido como **adiamento indefinido** (também chamado **inanição**). Nos exercícios, discutimos um algoritmo de embaralhamento melhor que elimina a possibilidade de adiamento indefinido.

Dica de desempenho 8.3

Às vezes os algoritmos que emergem de modo 'natural' podem conter problemas de desempenho sutis como o adiamento indefinido. Busque algoritmos que evitem adiamento indefinido.

Para distribuir a primeira carta, procuramos no array o elemento `deck[ row ][ column ]` que corresponda a `1`. Isso é realizado com uma instrução aninhada `for` que varia `row` de 0 a 3 e `column` de 0 a 12. A que carta esse slot de array corresponde? O array `suit` foi pré-carregado com os quatro naipes, portanto, para obter o naipe, imprimimos a string de caracteres `suit[ row ]`. De modo semelhante, para obter o valor de face da carta, imprimimos a string de caracteres `face[ column ]`. Imprimimos também a string de caracteres `" of "`. Imprimir essas informações na ordem adequada permite imprimir cada carta na forma `"King of Clubs"` [Rei de paus], `"Ace of Diamonds"` [Ás de ouros] e assim por diante.

Vamos prosseguir com o processo de refinamento passo a passo de cima para baixo. A parte superior é simplesmente

Embaralhe e distribua as 52 cartas

Nosso primeiro refinamento resulta em

Inicialize o array suit
Inicialize o array face
Inicialize o array deck
Embaralhe as cartas
Distribua as 52 cartas

'Embaralhe as cartas' pode ser expandido como mostrado a seguir:

Para cada uma das 52 cartas
    Coloque o número da carta no slot de baralho vazio selecionado aleatoriamente

'Distribua as 52 cartas' pode ser expandido como mostrado a seguir:

Para cada uma das 52 cartas
    Localize o número no array deck e imprima a face e o naipe da carta

Incorporar essas expansões produz nosso segundo refinamento completo:

Inicialize o array suit
Inicialize o array face
Inicialize o array deck
Para cada uma das 52 cartas
    Coloque o número da carta no slot vazio selecionado aleatoriamente
Para cada uma das 52 cartas
    Localize o número no array deck e imprima a face e o naipe da carta

'Coloque o número de carta no slot vazio selecionado aleatoriamente' pode ser expandido como mostrado a seguir:

Escolha o slot aleatoriamente
Enquanto o slot escolhido tiver sido previamente escolhido
    Escolha o slot aleatoriamente
Coloque o número de carta no slot escolhido do baralho

'Localize o número de carta no array deck e imprima a face e o naipe da carta' pode ser expandido como mostrado a seguir:

Para cada slot do array deck
Se o slot contiver o número da carta
Imprima a face e o naipe da carta

Incorporar essas expansões produz nosso terceiro refinamento (Figura 8.24).

Isso completa o processo de refinamento. As figuras 8.25–8.27 contêm o programa de embaralhamento e distribuição de cartas e uma execução de exemplo. As linhas 61–67 da função `deal` (Figura 8.26) implementam as linhas 1–2 da Figura 8.24. O construtor (linhas 22–35 da Figura 8.26) implementa as linhas 1–3 da Figura 8.24. A função `shuffle` (linhas 38–55 da Figura 8.26) implementa as linhas 5–11 da Figura 8.24. A função `deal` (linhas 58–88 da Figura 8.26) implementa as linhas 13–16 da Figura 8.24. Observe a formatação de saída utilizada na função `deal` (linhas 81–83 da Figura 8.26). A instrução de saída gera saída da face alinhada à direita em um campo de cinco caracteres e gera saída do naipe alinhado à esquerda em um campo de oito caracteres (Figura 8.27). A saída é impressa em formato de duas colunas — se a carta cuja saída está sendo gerada estiver na primeira coluna, uma tabulação é enviada para a saída depois da carta a ser movida para a segunda coluna (linha 83); caso contrário, uma nova linha é enviada para a saída.

Há também uma fraqueza no algoritmo de distribuição de cartas. Uma vez que uma correspondência é encontrada, mesmo que seja encontrada na primeira tentativa, as duas instruções `for` internas continuam procurando uma correspondência nos elementos restantes de `deck`. Nos exercícios, corrigimos essa deficiência.

```
1  Inicialize o array suit
2  Inicialize o array face
3  Inicialize o array deck
4
5  Para cada uma das 52 cartas
6      Escolha o slot aleatoriamente
7
8      Enquanto o slot tiver sido previamente escolhido
9          Escolha o slot aleatoriamente
10
11     Coloque o número de carta no slot escolhido do baralho
12
13 Para cada uma das 52 cartas
14     Para cada slot do array deck
15         Se o slot contiver o número de carta desejado
16             Imprima a face e o naipe da carta
```

Figura 8.24 O algoritmo em pseudocódigo para o programa de embaralhamento e distribuição de cartas.

```
1  // Figura 8.25: DeckOfCards.h
2  // Definição da classe DeckOfCards que
3  // representa um baralho.
4
5  // Definição da classe DeckOfCards
6  class DeckOfCards
7  {
8  public:
9     DeckOfCards(); // construtor inicializa deck
10    void shuffle(); // embaralha as cartas do baralho
11    void deal(); // distribui as cartas do baralho
12 private:
13    int deck[ 4 ][ 13 ]; // representa o baralho de cartas
14 }; // fim da classe DeckOfCards
```

Figura 8.25 Arquivo de cabeçalho `DeckOfCards`.

```
1  // Figura 8.26: DeckOfCards.cpp
2  // Definições de função-membro para a classe DeckOfCards que simula
3  // o embaralhamento e distribuição de um baralho.
4  #include <iostream>
5  using std::cout;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // protótipos para rand e srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // protótipo para time
17 using std::time;
18
19 #include "DeckOfCards.h" // definição da classe DeckOfCards
20
21 // construtor-padrão DeckOfCards inicializa deck
22 DeckOfCards::DeckOfCards()
23 {
24    // itera pelas linhas do baralho
25    for ( int row = 0; row <= 3; row++ )
26    {
27       // itera pelas colunas do baralho para linha atual
28       for ( int column = 0; column <= 12; column++ )
29       {
30          deck[ row ][ column ] = 0; // inicializa slot de deck como 0
31       } // fim do for interno
32    } // fim do for externo
33
34    srand( time( 0 ) ); // semeia o gerador de número aleatório
35 } // fim do construtor-padrão DeckOfCards
36
37 // embaralha as cartas do baralho
38 void DeckOfCards::shuffle()
39 {
40    int row; // representa o valor do naipe da carta
41    int column; // representa o valor da face da carta
42
43    // para cada uma das 52 cartas, escolhe um slot aleatoriamente
44    for ( int card = 1; card <= 52; card++ )
45    {
46       do // escolhe uma nova localização aleatória até um slot vazio ser encontrado
47       {
48          row = rand() % 4; // seleciona a linha aleatoriamente
49          column = rand() % 13; // seleciona a coluna aleatoriamente
50       } while( deck[ row ][ column ] != 0 ); // fim da instrução do...while
51
52       // coloca o número de carta no slot escolhido
53       deck[ row ][ column ] = card;
54    } // fim do for
55 } // fim da função shuffle
56
57 // distribui as cartas do baralho
```

Figura 8.26 Definições de funções-membro para embaralhamento e distribuição. (continua)

```
58 void DeckOfCards::deal()
59 {
60    // inicializa o array suit
61    static const char *suit[ 4 ] =
62       { "Hearts", "Diamonds", "Clubs", "Spades" };
63
64    // inicializa o array face
65    static const char *face[ 13 ] =
66       { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
67       "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
68
69    // para cada uma das 52 cartas
70    for ( int card = 1; card <= 52; card++ )
71    {
72       // itera pelas linhas do baralho
73       for ( int row = 0; row <= 3; row++ )
74       {
75          // itera pelas colunas de baralho para linha atual
76          for ( int column = 0; column <= 12; column++ )
77          {
78             // se o slot contiver a carta atual, exibe a carta
79             if ( deck[ row ][ column ] == card )
80             {
81                cout << setw( 5 ) << right << face[ column ]
82                   << " of " << setw( 8 ) << left << suit[ row ]
83                   << ( card % 2 == 0 ? '\n' : '\t' );
84             } // fim do if
85          } // fim do for mais interno
86       } // fim do for interno
87    } // fim do for externo
88 } // fim da função deal
```

Figura 8.26 Definições de funções-membro para embaralhamento e distribuição. (continuação)

```
1  // Figura 8.27: fig08_27.cpp
2  // Programa de embaralhamento e distribuição de cartas.
3  #include "DeckOfCards.h" // Definição da classe DeckOfCards
4
5  int main()
6  {
7     DeckOfCards deckOfCards; // cria objeto DeckOfCards
8
9     deckOfCards.shuffle(); // embaralha as cartas
10    deckOfCards.deal(); // distribui as cartas
11    return 0; // indica terminação bem-sucedida
12 } // fim de main
```

```
 Nine of Spades         Seven of Clubs
 Five of Spades         Eight of Clubs
Queen of Diamonds       Three of Hearts
 Jack of Spades          Five of Diamonds
 Jack of Diamonds       Three of Diamonds
Three of Clubs            Six of Clubs
  Ten of Clubs           Nine of Diamonds
```

Figura 8.27 Programa de embaralhamento e distribuição de cartas. (continua)

```
  Ace of Hearts         Queen of Hearts
Seven of Spades         Deuce of Spades
  Six of Hearts         Deuce of Clubs
  Ace of Clubs          Deuce of Diamonds
 Nine of Hearts         Seven of Diamonds
  Six of Spades         Eight of Diamonds
  Ten of Spades          King of Hearts
 Four of Clubs            Ace of Spades
  Ten of Hearts         Four of Spades
Eight of Hearts         Eight of Spades
 Jack of Hearts           Ten of Diamonds
 Four of Diamonds       King of Diamonds
Seven of Hearts         King of Spades
Queen of Spades         Four of Hearts
 Nine of Clubs           Six of Diamonds
Deuce of Hearts         Jack of Clubs
 King of Clubs         Three of Spades
Queen of Clubs          Five of Clubs
 Five of Hearts           Ace of Diamonds
```

Figura 8.27 Programa de embaralhamento e distribuição de cartas. (continuação)

8.12 Ponteiros de função

Um ponteiro para uma função contém o endereço da função na memória. No Capítulo 7, vimos que o nome de um array é, na realidade, o endereço na memória do primeiro elemento do array. De maneira semelhante, o nome de uma função é o endereço inicial na memória do código que realiza a tarefa da função. Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros de função.

Classificação por seleção para múltiplos propósitos utilizando ponteiros de função

Para ilustrar o uso de ponteiros para funções, a Figura 8.28 modifica o programa de classificação por seleção da Figura 8.15. A Figura 8.28 consiste em `main` (linhas 17–55) e nas funções `selectionSort` (linhas 59–76), `swap` (linhas 80–85), `ascending` (linhas 89–92) e `descending` (linhas 96–99). A função `selectionSort` recebe um ponteiro para uma função — uma função `ascending` ou `descending` — como um argumento além do array de inteiros a classificar e o tamanho do array. As funções `ascending` e `descending` determinam a ordem da classificação. O programa pede para o usuário escolher se o array deve ser classificado em ordem crescente ou decrescente (linhas 24–26). Se a entrada do usuário for 1, um ponteiro para a função `ascending` é passado para a função `selectionSort` (linha 37), fazendo com que o array seja classificado na ordem crescente. Se a entrada do usuário for 2, um ponteiro para a função `descending` é passado para a função `selectionSort` (linha 45), fazendo com que o array seja classificado em ordem decrescente.

O parâmetro a seguir aparece na linha 60 do cabeçalho de função de `selectionSort`:

```
bool ( *compare )( int, int )
```

Esse parâmetro especifica um ponteiro para uma função. A palavra-chave `bool` indica que a função sendo apontada retorna um valor `bool`. O texto `( *compare )` indica o nome do ponteiro para a função (o `*` indica que o parâmetro `compare` é um ponteiro). O texto `( int, int )` indica que a função apontada por comparação aceita dois argumentos do tipo inteiro. Os parênteses são necessários em torno de `*compare` para indicar que `compare` é um ponteiro para uma função. Se não tivéssemos incluído os parênteses, a declaração teria sido

```
bool *compare( int, int )
```

que declara uma função que recebe dois inteiros como parâmetros e retorna um ponteiro para um valor `bool`.

O parâmetro correspondente no protótipo de função de `selectionSort` é

```
bool (*)( int, int )
```

Observe que só foram incluídos tipos. Como sempre, para propósitos de documentação, o programador pode incluir nomes que o compilador irá ignorar.

A função passada para `selectionSort` é chamada na linha 71 como mostrado a seguir:

```
( *compare )( work[ smallestOrLargest ], work[ index ] )
```

Assim como um ponteiro para uma variável é desreferenciado para acessar o valor da variável, um ponteiro para uma função é desreferenciado para executar a função. Os parênteses em torno de `*compare` são novamente necessários — se não fossem incluídos, o operador `*` tentaria desreferenciar o valor retornado a partir da chamada de função. A chamada à função poderia ter sido feita sem desreferenciar o ponteiro, como em

```
compare( work[ smallestOrLargest ], work[ index ] )
```

que utiliza o ponteiro diretamente como o nome de função. Preferimos o primeiro método de chamar uma função por um ponteiro, porque ilustra explicitamente que `compare` é um ponteiro para uma função que é desreferenciado para chamar a função. O segundo método de chamar uma função por um ponteiro faz parecer que `compare` seria o nome de uma função real no programa. Isso pode ser confuso para um usuário do programa que gostaria de ver a definição de função `compare` e descobre que ela não está definida no arquivo.

```
1  // Figura 8.28: fig08_28.cpp
2  // Programa de classificação para múltiplos propósitos usando ponteiros de função.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 // protótipos
12 void selectionSort( int [], const int, bool (*)( int, int ) );
13 void swap( int * const, int * const );
14 bool ascending( int, int ); // implementa ordem crescente
15 bool descending( int, int ); // implementa ordem decrescente
16
17 int main()
18 {
19    const int arraySize = 10;
20    int order; // 1 = crescente, 2 = decrescente
21    int counter; // índice do array
22    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24    cout << "Enter 1 to sort in ascending order,\n"
25       << "Enter 2 to sort in descending order: ";
26    cin >> order;
27    cout << "\nData items in original order\n";
28
29    // gera saída do array original
30    for ( counter = 0; counter < arraySize; counter++ )
31       cout << setw( 4 ) << a[ counter ];
32
33    // classifica o array em ordem crescente; passa a função ascending
34    // como um argumento para especificar a ordem de classificação ascendente
35    if ( order == 1 )
36    {
37       selectionSort( a, arraySize, ascending );
38       cout << "\nData items in ascending order\n";
39    } // fim do if
40
41    // classifica o array em ordem decrescente; passa a função descending
42    // como um argumento para especificar a ordem de classificação descendente
43    else
44    {
45       selectionSort( a, arraySize, descending );
46       cout << "\nData items in descending order\n";
47    } // fim da parte else do if...else
48
```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função. (continua)

```
49    // gera a saída do array classificado
50    for ( counter = 0; counter < arraySize; counter++ )
51       cout << setw( 4 ) << a[ counter ];
52
53    cout << endl;
54    return 0; // indica terminação bem-sucedida
55 } // fim de main
56
57 // classificação por seleção para múltiplos propósitos; o parâmetro compare é um ponteiro para
58 // a função compare que determina a ordem de classificação
59 void selectionSort( int work[], const int size,
60                     bool (*compare)( int, int ) )
61 {
62    int smallestOrLargest; // índice do menor (ou maior) elemento
63
64    // itera sobre size - 1 elementos
65    for ( int i = 0; i < size - 1; i++ )
66    {
67       smallestOrLargest = i; // primeiro índice do vetor restante
68
69       // itera para localizar o índice do menor (ou maior) elemento
70       for ( int index = i + 1; index < size; index++ )
71          if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
72             smallestOrLargest = index;
73
74       swap( &work[ smallestOrLargest ], &work[ i ] );
75    } // fim do if
76 } // fim da função selectionSort
77
78 // troca os valores nas posições da memória para as quais
79 // element1Ptr e element2Ptr apontem
80 void swap( int * const element1Ptr, int * const element2Ptr )
81 {
82    int hold = *element1Ptr;
83    *element1Ptr = *element2Ptr;
84    *element2Ptr = hold;
85 } // fim da função swap
86
87 // determina se o elemento a é menor que o
88 // elemento b para uma classificação em ordem crescente
89 bool ascending( int a, int b )
90 {
91    return a < b; // retorna true se a for menor que b
92 } // fim da função ascending
93
94 // determina se o elemento a é maior que o
95 // elemento b para uma classificação em ordem decrescente
96 bool descending( int a, int b )
97 {
98    return a > b; // retorna true se a for maior que b
99 } // fim da função descending
```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função. (continua)

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função. (continuação)

Arrays de ponteiros para funções

Uma utilização de ponteiros de função é em sistemas baseados em menus. Por exemplo, um programa poderia pedir para um usuário selecionar uma opção de um menu inserindo valores de um inteiro. A escolha do usuário pode ser utilizada como um subscrito para um array de ponteiros de função e o ponteiro no array pode ser utilizado para chamar a função.

A Figura 8.29 fornece um exemplo mecânico que demonstra como declarar e utilizar um array de ponteiros para funções. O programa define três funções — `function0`, `function1` e `function2` — cada uma aceita um argumento de inteiro e não retorna um valor. A linha 17 armazena ponteiros para essas três funções no array `f`. Nesse caso, todas as funções para as quais o array aponta devem ter o mesmo tipo de retorno e os mesmos tipos de parâmetro. A declaração na linha 17 é lida começando no conjunto de parênteses mais à esquerda como, '`f` é um array de três ponteiros para funções que aceitam um `int` como um argumento e retornam `void`'. O array é inicializado com os nomes das três funções (que, novamente, são ponteiros). O programa pede para o usuário inserir um número entre 0 e 2 ou 3 para terminar. Quando o usuário insere um valor entre 0 e 2, o valor é utilizado como o subscrito no array de ponteiros para funções. A linha 29 invoca uma das funções no array `f`. Na chamada, `f[ choice ]` seleciona o ponteiro na localização `choice` do array. O ponteiro é desreferenciado para chamar a função e `choice` é passado como o argumento à função. Cada função imprime o valor do seu argumento e seu nome de função para indicar que a função foi chamada corretamente. Nos exercícios, você desenvolverá um sistema baseado em menus. Veremos no Capítulo 13, "Programação orientada a objetos: polimorfismo", que os arrays de ponteiros para funções são utilizados por desenvolvedores de compiladores para implementar os mecanismos que suportam as funções `virtual` — a tecnologia-chave por trás do polimorfismo.

```
 1  // Figura 8.29: fig08_29.cpp
 2  // Demonstrando um array de ponteiros para funções.
 3  #include <iostream>
 4  using std::cout;
 5  using std::cin;
 6  using std::endl;
 7
 8  // protótipos de função -- cada função realiza ações semelhantes
 9  void function0( int );
10  void function1( int );
11  void function2( int );
12
13  int main()
14  {
15     // inicializa array de 3 ponteiros para funções que
16     // aceitam um argumento int e retornam void
```

Figura 8.29 Array de ponteiros para funções. (continua)

```
   void (*f[ 3 ])( int ) = { function0, function1, function2 };

   int choice;

   cout << "Enter a number between 0 and 2, 3 to end: ";
   cin >> choice;

   // processa escolha do usuário
   while ( ( choice >= 0 ) && ( choice < 3 ) )
   {
      // invoca a função na localização choice no
      // array f e passa choice como um argumento
      (*f[ choice ])( choice );

      cout << "Enter a number between 0 and 2, 3 to end: ";
      cin >> choice;
   } // fim do while

   cout << "Program execution completed." << endl;
   return 0; // indica terminação bem-sucedida
} // fim de main

void function0( int a )
{
   cout << "You entered " << a << " so function0 was called\n\n";
} // fim da função function0

void function1( int b )
{
   cout << "You entered " << b << " so function1 was called\n\n";
} // fim da função function1

void function2( int c )
{
   cout << "You entered " << c << " so function2 was called\n\n";
} // fim da função function2
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function0 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function1 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function2 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Figura 8.29 Array de ponteiros para funções. (continuação)

8.13 Introdução ao processamento de string baseada em ponteiro

Nesta seção, introduzimos algumas funções comuns da C++ Standard Library que facilitam o processamento de string. As técnicas aqui discutidas são apropriadas para desenvolver editores de texto, processadores de texto, software de layout de página, sistemas computadorizados de composição e outros tipos de software de processamento de textos. Já utilizamos a classe `string` da C++ Standard Library em vários exemplos para representar strings como objetos completos com todos os recursos. Por exemplo, o estudo de caso da classe

`GradeBook` nos capítulos 3–7 representa um nome de curso que usa um objeto `string`. No Capítulo 18 apresentamos a classe `string` em detalhes. Embora utilizar os objetos `string` seja normalmente simples e direto, nesta seção, utilizamos strings baseadas em ponteiro terminadas por caractere nulo. Muitas funções da C++ Standard Library operam apenas em strings baseadas em ponteiro terminadas por caractere nulo, que são mais complicadas de utilizar do que os objetos `string`. Além disso, se você trabalha com programas C++ legados, pode ser solicitado a manipular essas strings baseadas em ponteiro.

8.13.1 Fundamentos de caracteres e strings baseadas em ponteiro

Os caracteres são os blocos de construção fundamentais do código-fonte dos programas C++. Cada programa é composto de uma seqüência de caracteres que — quando agrupados entre si significativamente — é interpretada pelo compilador como uma série de instruções utilizadas para realizar uma tarefa. Um programa pode conter **caracteres constantes**. Um caractere constante é um valor de inteiro representado como caractere entre aspas simples. O valor de uma constante de caractere é o valor inteiro do caractere no conjunto de caracteres da máquina. Por exemplo, `'z'` representa o valor inteiro de `z` (122 no conjunto de caracteres ASCII; ver Apêndice B) e `'\n'` representa o valor inteiro de nova linha (10 no conjunto de caracteres ASCII).

Uma string é uma série de caracteres tratada como uma unidade única. Uma string pode incluir letras, dígitos e vários **caracteres especiais** como `+`, `-`, `*`, `/` e `$`. Os **literais string**, ou **constantes string**, em C++ são escritos em aspas duplas como mostrado a seguir:

```
"John Q. Doe"              (um nome)
"9999 Main Street"         (um endereço)
"Maynard, Massachusetts"   (uma cidade e um estado)
"(201) 555-1212"           (um número de telefone)
```

Uma string baseada em ponteiro em C++ é um array de caracteres que acabam no caractere nulo (`'\0'`), que marca onde a string termina na memória. Uma string é acessada via um ponteiro para seu primeiro caractere. O valor de uma string é o endereço de seu primeiro caractere. Portanto, em C++, é apropriado dizer que *uma string é um ponteiro constante* — de fato, um ponteiro para o primeiro caractere da string. Nesse sentido, as strings são como arrays, porque um nome de array é também um ponteiro para seu primeiro elemento.

Um literal string pode ser utilizado como um inicializador na declaração de um array de caracteres ou de uma variável de tipo `char *`. Cada uma das declarações

```
char color[] = "blue";
const char *colorPtr = "blue";
```

inicializa uma variável como a string `"blue"`. A primeira declaração cria um array `color` de cinco elementos contendo o caractere `'b'`, `'l'`, `'u'`, `'e'` e `'\0'`. A segunda declaração cria o ponteiro de variável `colorPtr` que aponta para a letra `b` na string `"blue"` (que acaba em `'\0'`) em algum lugar da memória. Os literais string têm a classe de armazenamento `static` (eles existem até o fim do programa) e podem ou não ser compartilhados se o mesmo literal string for referenciado a partir de múltiplas localizações em um programa. Além disso, os literais string em C++ são constantes — seus caracteres não podem ser modificados.

A declaração `char color[] = "blue";` também poderia ser escrita

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

Ao declarar um array de caractere para conter uma string, o array deve ser suficientemente grande para armazenar a string e seu caractere de terminação nulo. A declaração anterior determina o tamanho do array, com base no número de inicializadores fornecidos na lista inicializadora.

Erro comum de programação 8.15

É um erro não alocar espaço suficiente em um array de caracteres para armazenar o caractere nulo que termina uma string.

Erro comum de programação 8.16

Criar ou utilizar uma string no estilo C que não contém um caractere nulo de terminação pode levar a erros de lógica.

Dica de prevenção de erro 8.4

Ao armazenar uma string de caracteres em um array de caracteres, certifique-se de que o array é grande o bastante para armazenar a maior string que será armazenada. O C++ permite que strings de qualquer comprimento sejam armazenadas. Se uma string for mais longa que o array de caracteres em que ela deve ser armazenada, os caracteres além do fim do array sobrescreverão os dados na memória seguinte ao array, resultando em erros de lógica.

Uma string pode ser lida em um array de caracteres utilizando a extração de fluxo com `cin`. Por exemplo, a seguinte instrução pode ser utilizada a fim de ler uma string para o array de caracteres `word[ 20 ]`:

```
cin >> word;
```

A string inserida pelo usuário é armazenada em `word`. A instrução precedente lê caracteres até que um caractere de espaço em branco ou indicador de fim do arquivo seja encontrado. Observe que a string não deve ter mais que 19 caracteres para deixar espaço para o

caractere de terminação nulo. O manipulador de fluxo `setw` pode ser utilizado para assegurar que a string lida em `word` não exceda o tamanho do array. Por exemplo, a instrução

```
cin >> setw( 20 ) >> word;
```

especifica que `cin` deve ler um máximo de 19 caracteres no array `word` e salvar a 20ª posição no array para armazenar o caractere de terminação nulo para a string. O manipulador de fluxo `setw` só se aplica ao próximo valor sendo inserido. Se mais de 19 caracteres forem inseridos, os caracteres restantes não são salvos em `word`, mas serão lidos e podem ser armazenados em outra variável.

Em alguns casos, é desejável inserir uma linha inteira de texto em um array. Para esse propósito, o C++ fornece a função **`cin.getline`** no arquivo de cabeçalho `<iostream>`. No Capítulo 3 você foi apresentado à função `getline` similar do arquivo de cabeçalho `<string>`, que lia a entrada até que um caractere de nova linha fosse inserido e armazenava a entrada (sem o caractere de nova linha) em uma `string` especificada como um argumento. A função `cin.getline` aceita três argumentos — um array de caracteres em que a linha de texto será armazenada, um comprimento e um caractere delimitador. Por exemplo, o segmento de programa

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```

declara o array `sentence` de 80 caracteres e lê uma linha de texto do teclado no array. A função pára de ler caracteres quando o caractere delimitador `'\n'` é encontrado, quando o indicador de fim do arquivo é inserido ou quando o número de caracteres lidos até agora é um menor que o comprimento especificado no segundo argumento. (O último caractere no array é reservado para o caractere de terminação nulo.) Se o caractere delimitador é encontrado, ele é lido e descartado. O terceiro argumento para `cin.getline` tem `'\n'` como um valor-padrão, então a chamada de função precedente poderia ter sido escrita como mostrado a seguir:

```
cin.getline( sentence, 80 );
```

O Capítulo 15, "Entrada/saída de fluxo", fornece uma discussão detalhada sobre `cin.getline` e outras funções de entrada/saída.

Erro comum de programação 8.17

Processar um único caractere como uma string `char *` pode levar a um erro fatal de tempo de execução. Uma string `char *` é um ponteiro — provavelmente um inteiro bem grande. Entretanto, um caractere é um inteiro pequeno (valores ASCII variam de 0–255). Em muitos sistemas, desreferenciar um valor `char` causa um erro, porque os endereços de memória baixa são reservados para usos especiais como handlers de interrupção de sistemas operacionais — portanto, ocorrem as 'violações de acesso de memória'.

Erro comum de programação 8.18

É um erro de compilação passar uma string como um argumento para uma função quando um caractere é esperado.

8.13.2 Funções de manipulação de string da biblioteca de tratamento de strings

A biblioteca de tratamento de strings fornece muitas funções úteis para manipular dados de string, comparar strings, pesquisar por caracteres e outras strings em strings, tokenizar strings (separá-las em partes lógicas como as palavras separadas em uma frase) e determinar seu comprimento. Esta seção apresenta algumas funções de manipulação de strings comuns da biblioteca de tratamento de strings (da biblioteca-padrão C++). As funções são resumidas na Figura 8.30; em seguida, cada uma delas é utilizada em um exemplo de 'código ativo' (*live-code*). Os protótipos para essas funções são encontrados no arquivo de cabeçalho `<cstring>`.

Observe que várias funções na Figura 8.30 contêm parâmetros com tipo de dados `size_t`. Esse tipo é definido no arquivo de cabeçalho `<cstring>` como um tipo integral sem sinal como `unsigned int` ou `unsigned long`.

Erro comum de programação 8.19

Esquecer de incluir o arquivo de cabeçalho `<cstring>` ao usar funções da biblioteca de tratamento de strings causa erros de compilação.

Copiando strings com ***strcpy*** *e* ***strncpy***

A função **`strcpy`** copia seu segundo argumento — uma string — para seu primeiro argumento — um array de caracteres que deve ser grande o bastante para armazenar a string e seu caractere de terminação nulo (que também é copiado). A função **`strncpy`** é equivalente à `strcpy`, exceto pelo fato de que `strncpy` especifica o número de caracteres a serem copiados da string para o array. Observe que a função `strncpy` não copia necessariamente o caractere de terminação nulo de seu segundo argumento — um caractere de terminação nulo só é escrito se o número de caracteres a ser copiado for pelo menos um maior que o comprimento da string. Por exemplo, se `"test"` é o segundo argumento, um caractere de terminação nulo só é escrito se o terceiro argumento para `strncpy` for de pelo menos `5` (quatro caracteres em `"test"` mais um caractere de terminação nulo). Se o terceiro argumento for maior que `5`, os caracteres nulos serão acrescentados ao array até que o número total de caracteres especificado pelo terceiro argumento seja escrito.

Protótipo da função	Descrição da função
`char *strcpy( char *s1, const char *s2 );`	Copia a string `s2` para o array de caracteres `s1`. O valor de `s1` é retornado.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copia no máximo `n` caracteres da string `s2` para o array de caractere `s1`. O valor de `s1` é retornado.
`char *strcat( char *s1, const char *s2 );`	Acrescenta a string `s2` a `s1`. O primeiro caractere de `s2` sobrescreve o caractere de terminação nulo de `s1`. O valor de `s1` é retornado.
`char *strncat( char *s1, const char *s2, size_t n );`	Acrescenta no máximo `n` caracteres da string `s2` à string `s1`. O primeiro caractere de `s2` sobrescreve o caractere de terminação nulo de `s1`. O valor de `s1` é retornado.
`int strcmp( const char *s1, const char *s2 );`	Compara a string `s1` com a string `s2`. A função retorna um valor zero, menor que zero (normalmente `-1`) ou maior que zero (normalmente 1) se `s1` for igual, maior ou menor que `s2`, respectivamente.
`int strncmp( const char *s1, const char *s2, size_t n );`	Compara até `n` caracteres da string `s1` com a string `s2`. A função retorna zero, menor que zero ou maior que zero se a parte de `n` caracteres de `s1` é igual a, maior ou menor que a parte correspondente de `n` caracteres de `s2`, respectivamente.
`char *strtok( char *s1, const char *s2 );`	Uma seqüência de chamadas a `strtok` quebra a string `s1` em 'tokens' — partes lógicas como as palavras em uma linha de texto. A string é dividida com base nos caracteres contidos na string `s2`. Por exemplo, se fôssemos quebrar a string `"this:is:a:string"` em tokens com base no caractere `':'`, os tokens resultantes seriam `"this"`, `"is"`, `"a"` e `"string"`. Entretanto, a função `strtok` retorna somente um token por vez. A primeira chamada contém `s1` como o primeiro argumento, e as chamadas subseqüentes que continuam tokenizando a mesma string contêm `NULL` como o primeiro argumento. Cada chamada retorna um ponteiro para o token atual. Se não houver mais tokens quando a função for chamada, `NULL` é retornado.
`size_t strlen( const char *s );`	Determina o comprimento da string `s`. O número de caracteres que precedem o caractere de terminação nulo é retornado.

Figura 8.30 Funções de manipulação de string da biblioteca de tratamento de strings.

Erro comum de programação 8.20

Ao utilizar `strncpy`, o caractere de terminação nulo do segundo argumento (uma string `char *`) não será copiado se o número de caracteres especificado pelo terceiro argumento de `strncpy` não for maior que o comprimento do segundo argumento. Nesse caso, pode ocorrer um erro fatal se o programador não terminar manualmente a string `char *` resultante com um caractere nulo.

A Figura 8.31 utiliza `strcpy` (linha 17) para copiar a string inteira do array `x` para o array `y`, e utiliza `strncpy` (linha 23) para copiar os primeiros 14 caracteres do array `x` para o array `z`. A linha 24 acrescenta um caractere nulo (`'\0'`) ao array `z`, porque a chamada a `strncpy` no programa não escreve um caractere de terminação nulo. (O terceiro argumento é menor que o comprimento de string do segundo argumento mais um.)

Concatenando strings com `strcat` *e* `strncat`

A função `strcat` acrescenta seu segundo argumento (uma string) a seu primeiro argumento (um array de caracteres contendo uma string). O primeiro caractere do segundo argumento substitui o caractere nulo (`'\0'`) que termina a string no primeiro argumento. O programador deve assegurar que o array utilizado para armazenar a primeira string é suficientemente grande para armazenar a combinação da primeira e da segunda strings e do caractere de terminação nulo (copiado a partir da segunda string). A função `strncat` acrescenta um número especificado de caracteres da segunda string para a primeira string e acrescenta um caractere de terminação nulo ao resultado. O programa da Figura 8.32 demonstra a função `strcat` (linhas 19 e 29) e a função `strncat` (linha 24).

```
1  // Figura 8.31: fig08_31.cpp
2  // Utilizando strcpy e strncpy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // protótipos para strcpy e strncpy
8  using std::strcpy;
9  using std::strncpy;
10
11 int main()
12 {
13    char x[] = "Happy Birthday to You"; // o comprimento da string é 21
14    char y[ 25 ];
15    char z[ 15 ];
16
17    strcpy( y, x ); // copia conteúdo de x para y
18
19    cout << "The string in array x is: " << x
20       << "\nThe string in array y is: " << y << '\n';
21
22    // copia os primeiros 14 caracteres de x para z
23    strncpy( z, x, 14 ); // não copia o caractere nulo
24    z[ 14 ] = '\0'; // acrescenta '\0' ao conteúdo de z
25
26    cout << "The string in array z is: " << z << endl;
27    return 0; // indica terminação bem-sucedida
28 } // fim de main
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Figura 8.31 strcpy e strncpy.

```
1  // Figura 8.32: fig08_32.cpp
2  // Utilizando strcat e strncat.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // protótipos para strcat e strncat
8  using std::strcat;
9  using std::strncat;
10
11 int main()
12 {
13    char s1[ 20 ] = "Happy "; // comprimento 6
14    char s2[] = "New Year "; // comprimento 9
15    char s3[ 40 ] = "";
16
17    cout << "s1 = " << s1 << "\ns2 = " << s2;
18
19    strcat( s1, s2 ); // concatena s2 com s1 (comprimento 15)
```

Figura 8.32 strcat e strncat. (continua)

```
20
21     cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23     // concatena os 6 primeiros caracteres de s1 a s3
24     strncat( s3, s1, 6 ); // coloca '\0' depois de último caractere
25
26     cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27        << "\ns3 = " << s3;
28
29     strcat( s3, s1 ); // concatena s1 a s3
30     cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31        << "\ns3 = " << s3 << endl;
32     return 0; // indica terminação bem-sucedida
33  } // fim de main
```

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

Figura 8.32 strcat e strncat. (continuação)

Comparando strings com ***strcmp*** *e* ***strncmp***

A Figura 8.33 compara três strings utilizando **strcmp** (linhas 21, 22 e 23) e **strncmp** (linhas 26, 27 e 28). A função strcmp compara, caractere por caractere, seu primeiro argumento de string com seu segundo argumento de string. A função retornará zero se as strings forem iguais, um valor negativo se a primeira string for menor que a segunda string e um valor positivo se a primeira string for maior que a segunda. A função strncmp é equivalente à strcmp, exceto pelo fato de que strncmp compara até um número especificado de caracteres. A função strncmp pára de comparar caracteres se alcançar o caractere nulo em um de seus argumentos de string. O programa imprime o valor inteiro retornado em cada chamada de função.

Erro comum de programação 8.21

Pressupor que strcmp *e* strncmp *retornam 1 (valor verdadeiro) quando seus argumentos são iguais é um erro de lógica. Ambas as funções retornam zero (valor falso do C++) para igualdade. Portanto, ao testar a igualdade de duas strings, o resultado da função* strcmp *ou* strncmp *deve ser comparado com zero para determinar se as strings são iguais.*

Para entender exatamente o que significa uma string ser 'maior que' ou 'menor que' outra string, considere o processo de alfabetar uma série de sobrenomes. O leitor iria, sem dúvida, colocar 'Jones' antes de 'Smith' porque a primeira letra de 'Jones' vem antes da primeira letra de 'Smith' no alfabeto. Mas o alfabeto é mais que uma simples lista de 26 letras — é uma lista ordenada de caracteres. Cada letra ocorre em uma posição específica dentro da lista. O 'z' é mais que apenas uma letra do alfabeto; 'z' é especificamente a 26ª letra do alfabeto.

Como o computador sabe que uma letra vem antes de outra? Todos os caracteres são representados dentro do computador como códigos numéricos; quando o computador compara duas strings, na realidade, ele compara os códigos numéricos dos caracteres nas strings.

Em um esforço de padronizar representações de caractere, a maioria dos fabricantes de computador projetou suas máquinas para utilizar um dos dois esquemas de codificação populares — o ASCII ou **EBCDIC**. Lembre-se de que ASCII significa 'American Standard Code for Information Interchange'. EBCDIC quer dizer 'Extended Binary Coded Decimal Interchange Code'. Existem outros esquemas de codificação, mas esses dois são os mais populares.

O ASCII e o EBCDIC são chamados de **códigos de caractere** ou conjuntos de caracteres. A maioria dos leitores deste livro utilizará computadores desktop ou notebooks que empregam o conjunto de caracteres ASCII. Os computadores mainframe da IBM utilizam o conjunto de caracteres EBCDIC. Como o uso da Internet e da World Wide Web torna-se intenso no mundo todo, o mais novo conjunto de caracteres Unicode está crescendo rapidamente em popularidade. Para informações adicionais sobre o Unicode, visite `www.unicode.org`. As manipulações de strings e caracteres, na realidade, envolvem a manipulação dos códigos numéricos apropriados e não os caracteres em si. Isso explica a intercambialidade entre caracteres e inteiros pequenos no C++. Visto que é significativo dizer que um código numérico é maior que, menor que ou igual a outro código numérico, torna-se possível relacionar vários caracteres ou strings entre si para se referir aos códigos de caractere. O Apêndice B contém os códigos de caractere ASCII.

Dica de portabilidade 8.5

Os códigos numéricos internos utilizados para representar caracteres podem ser diferentes em computadores diferentes, porque esses computadores podem utilizar conjuntos de caracteres diferentes.

Dica de portabilidade 8.6

Não teste explicitamente os códigos ASCII, como em `if ( rating == 65 )`; em vez disso, use a constante de caractere correspondente, como em `if ( rating == 'A' )`.

[*Nota:* Com alguns compiladores, as funções `strcmp` e `strncmp` sempre retornam `-1`, `0` ou `1`, como na saída de exemplo da Figura 8.33. Com outros compiladores, essas funções retornam 0 ou a diferença entre os códigos numéricos dos primeiros caracteres que diferem nas strings sendo comparadas. Por exemplo, quando `s1` e `s3` são comparados, os primeiros caracteres que diferem entre eles são os primeiros caracteres da segunda palavra em cada string — `N` (código numérico 78) em `s1` e `H` (código numérico 72) em `s3`, respectivamente. Nesse caso, o valor de retorno será `6` (ou `-6` se `s3` for comparado com `s1`).]

```
1  // Figura 8.33: fig08_33.cpp
2  // Utilizando strcmp e strncmp.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstring> // protótipos para strcmp e strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16    char *s1 = "Happy New Year";
17    char *s2 = "Happy New Year";
18    char *s3 = "Happy Holidays";
19
20    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
21       << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
22       << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
23       << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
24
25    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
26       << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
27       << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = " << setw( 2 )
28       << strncmp( s3, s1, 7 ) << endl;
29    return 0; // indica terminação bem-sucedida
30 } // fim de main
```

Figura 8.33 `strcmp` e `strncmp`. (continua)

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Figura 8.33 `strcmp` e `strncmp`. (continuação)

Tokenizando uma string com ***strtok***

A função **`strtok`** divide uma string em uma série de **tokens**. Token é uma seqüência de caracteres separados por **caracteres delimitadores** (normalmente espaços ou sinais de pontuação). Por exemplo, em uma linha de texto, cada palavra pode ser considerada um token, e os espaços que separam as palavras podem ser considerados delimitadores.

São necessárias múltiplas chamadas para `strtok` para se dividir uma string em tokens (supondo que a string contenha mais de um token). A primeira chamada a `strtok` contém dois argumentos, uma string a ser tokenizada e uma string contendo caracteres que separam os tokens (isto é, delimitadores). A linha 19 na Figura 8.34 atribui a `tokenPtr` um ponteiro para o primeiro token em `sentence`. O segundo argumento, `" "`, indica que os tokens em `sentence` são separados por espaços. A função `strtok` procura o primeiro caractere na `sentence` que não seja um caractere delimitador (espaço). Isso no primeiro token. A função então localiza o próximo caractere delimitador na string e o substitui por um caractere nulo (`'\0'`). Isso termina o token atual. A função `strtok` salva (em uma variável `static`) um ponteiro para o próximo caractere que se segue ao token em `sentence` e retorna um ponteiro para o token atual.

As chamadas subseqüentes a `strtok` para continuar tokenizando `sentence` contêm `NULL` como o primeiro argumento (linha 25). O argumento `NULL` indica que a chamada a `strtok` deve continuar tokenizando a partir da localização em `sentence` salva pela última chamada a `strtok`. Observe que `strtok` mantém essas informações salvas de modo que não fiquem visíveis ao programador. Se nenhum token restar quando `strtok` for chamado, `strtok` retornará `NULL`. O programa da Figura 8.34 utiliza `strtok` para tokenizar a string `"This is a sentence with 7 tokens"`. O programa imprime cada token em uma linha separada. A linha 28 gera saída de `sentence` depois da tokenização. Observe que *`strtok` modifica a string de entrada*; portanto, uma cópia da string deve ser feita se o programa exigir a original depois das chamadas para `strtok`. Quando `sentence` é enviada para a saída depois da tokenização, observe que somente a palavra '`This`' é impressa, porque `strtok` substituiu cada espaço em branco em `sentence` por um caractere nulo (`'\0'`) durante o processo de tokenização.

```
1  // Figura 8.34: fig08_34.cpp
2  // Utilizando strtok.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // protótipo para strtok
8  using std::strtok;
9
10 int main()
11 {
12    char sentence[] = "This is a sentence with 7 tokens";
13    char *tokenPtr;
14
15    cout << "The string to be tokenized is:\n" << sentence
16       << "\n\nThe tokens are:\n\n";
17
```

Figura 8.34 `strtok`. (continua)

```
18     // inicia a tokenização da frase
19     tokenPtr = strtok( sentence, " " );
20
21     // continua tokenizando a frase até tokenPtr tornar-se NULL
22     while ( tokenPtr != NULL )
23     {
24        cout << tokenPtr << '\n';
25        tokenPtr = strtok( NULL, " " ); // obtém o próximo token
26     } // fim do while
27
28     cout << "\nAfter strtok, sentence = " << sentence << endl;
29     return 0; // indica terminação bem-sucedida
30  } // fim de main
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

Figura 8.34 `strtok`. (continuação)

Erro comum de programação 8.22

Não perceber que `strtok` *modifica a string tokenizada e, então, tentar utilizar essa string como se ela fosse a string original não modificada é um erro de lógica.*

Determinando comprimentos de string

A função **`strlen`** aceita uma string como um argumento e retorna o número de caracteres na string — o caractere de terminação nulo não é incluído no comprimento. O comprimento também é o índice do caractere nulo. O programa da Figura 8.35 demonstra a função `strlen`.

```
1  // Figura 8.35: fig08_35.cpp
2  // Utilizando strlen.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // protótipo para strlen
8  using std::strlen;
9
10 int main()
11 {
```

Figura 8.35 `strlen` retorna o comprimento de uma string `char *`. (continua)

```
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "four";
   char *string3 = "Boston";

   cout << "The length of \"" << string1 << "\" is " << strlen( string1 )
      << "\nThe length of \"" << string2 << "\" is " << strlen( string2 )
      << "\nThe length of \"" << string3 << "\" is " << strlen( string3 )
      << endl;
   return 0; // indica terminação bem-sucedida
} // fim de main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Figura 8.35 `strlen` retorna o comprimento de uma string `char *`. (continuação)

8.14 Síntese

Neste capítulo fornecemos uma introdução detalhada sobre ponteiros ou variáveis que contêm endereços de memória como seus valores. Começamos demonstrando como declarar e inicializar ponteiros. Você viu como utilizar o operador de endereço (`&`) para atribuir o endereço de uma variável a um ponteiro e o operador de indireção (`*`) para acessar os dados armazenados na variável indiretamente referenciada por um ponteiro. Discutimos como passar argumentos por referência utilizando tanto os argumentos de ponteiro como os argumentos de referência.

Você aprendeu a utilizar const com ponteiros para impor o princípio do menor privilégio. Demonstramos como utilizar ponteiros não constantes para dados não constantes, ponteiros não constantes para dados constantes, ponteiros constantes para dados não constantes e ponteiros constantes para dados constantes. Em seguida, utilizamos a classificação por seleção para demonstrar como passar arrays e elementos do array individuais por referência. Discutimos o operador `sizeof`, que pode ser utilizado para determinar o tamanho, em bytes, de um tipo de dados durante a compilação de programa.

Continuamos demonstrando como utilizar os ponteiros em expressões aritméticas e de comparação. Você viu que a aritmética de ponteiros pode ser utilizada para pular de um elemento de um array para outro. Você aprendeu a utilizar arrays de ponteiros e, mais especificamente, arrays de string (uma string de arrays). Então continuamos discutindo os ponteiros de função, que permitem aos programadores passar funções como parâmetros. Concluímos o capítulo com uma discussão sobre as várias funções C++ que manipulam strings baseadas em ponteiro. Você aprendeu as capacidades de processamento de string, por exemplo, copiar strings, tokenizá-las e determinar seu comprimento.

No próximo capítulo, iniciamos nosso tratamento aprofundado de classes. Você aprenderá sobre o escopo de membros de uma classe e sobre como manter os objetos em um estado consistente. Também aprenderá a utilizar funções-membro especiais chamadas de construtores e destrutores, que executam quando um objeto é criado e destruído, respectivamente.

Resumo

- Os ponteiros são variáveis que contêm como seus valores endereços de memória de outras variáveis.
- A declaração

  ```
  int *ptr;
  ```

  declara `ptr` como um ponteiro para uma variável do tipo `int` e se lê como '`ptr` é um ponteiro para `int`'. O `*`, do modo usado aqui em uma declaração, indica que a variável é um ponteiro.
- Há três valores que podem ser utilizados para inicializar um ponteiro: `0`, `NULL` ou um endereço de um objeto do mesmo tipo. A inicialização de um ponteiro como `0` e a inicialização desse mesmo ponteiro como `NULL` são idênticas — `0` é a convenção em C++.
- O único inteiro que pode ser atribuído a um ponteiro sem coerção é zero.
- O operador `&` (endereço) retorna o endereço de memória de seu operando.
- O operando do operador de endereço deve ser um nome de variável (ou outro *lvalue*); o operador de endereço não pode ser aplicado a constantes ou expressões que não retornam uma referência.

- O operador *, referido como o operador de indireção (ou desreferenciação), retorna um sinônimo, alias ou apelido para o nome do objeto para o qual seu operando aponta na memória. Isso é chamado de desreferenciar o ponteiro.
- Ao chamar uma função com um argumento que o chamador quer que a função chamada modifique, o endereço do argumento pode ser passado. A função chamada então utiliza o operador de indireção (*) para desreferenciar o ponteiro e modificar o valor do argumento na função chamadora.
- Uma função que recebe um endereço como um argumento deve ter um ponteiro como seu parâmetro correspondente.
- O qualificador `const` permite que o programador informe ao compilador que o valor de uma variável particular não pode ser modificado pelo identificador especificado. Se uma tentativa de modificar um valor `const` for feita, o compilador emite um aviso ou um erro, dependendo do compilador particular.
- Há quatro maneiras de passar um ponteiro para uma função — um ponteiro não constante para dados não constantes, um ponteiro não constante para dados constantes, um ponteiro constante para dados não constantes e um ponteiro constante para dados constantes.
- O valor do nome do array é o endereço de (um ponteiro para) um primeiro elemento do array.
- Para passar um único elemento de um array por referência utilizando ponteiros, passe o endereço do elemento do array específico.
- O C++ fornece o operador unário `sizeof` para determinar o tamanho de um array (ou de qualquer outro tipo de dados, variável ou constante) em bytes em tempo de compilação.
- Quando aplicado ao nome de um array, o operador `sizeof` retorna o número total de bytes no array como um inteiro.
- As operações aritméticas que podem ser realizadas em ponteiros são incrementar (`++`) um ponteiro, decrementar (`--`) um ponteiro, adicionar (`+` ou `+=`) um inteiro a um ponteiro, subtrair (`-` ou `-=`) um inteiro de um ponteiro e subtrair um ponteiro de outro.
- Quando um inteiro é adicionado a ou subtraído de um ponteiro, o ponteiro é incrementado ou decrementado por esse inteiro vezes o tamanho do objeto que o ponteiro referencia.
- Dois ponteiros podem ser atribuídos reciprocamente se ambos forem do mesmo tipo. Caso contrário, uma coerção deve ser utilizada. A exceção a isso é um ponteiro `void *`, que é um tipo genérico de ponteiro que pode armazenar valores de ponteiro de qualquer tipo. Os ponteiros para `void` podem ser atribuídos a ponteiros de outros tipos. Um ponteiro `void *` pode ser atribuído a um ponteiro de outro tipo somente com uma coerção de tipo explícita.
- As únicas operações válidas em um ponteiro `void *` são a comparação de ponteiros `void *` com outros ponteiros, a atribuição de endereços a ponteiros `void *` e a coerção de ponteiros `void *` em tipos de ponteiros válidos.
- Os ponteiros podem ser comparados utilizando operadores de igualdade e relacionais. As comparações que utilizam operadores relacionais só são significativas se os ponteiros apontarem para membros do mesmo array.
- Os ponteiros que apontam para arrays podem ser subscritos exatamente como os nomes de array podem.
- Na notação de ponteiro/deslocamento, se o ponteiro apontar para o primeiro elemento do array, o deslocamento é o mesmo que um subscrito do array.
- Todas as expressões de array indexadas com subscritos podem ser escritas com um ponteiro e um deslocamento, utilizando o nome do array como um ponteiro ou um ponteiro separado que aponta para o array.
- Os arrays podem conter ponteiros.
- Um ponteiro para uma função é o endereço em que o código da função reside.
- Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros.
- Uma utilização comum de ponteiros de função é nos assim chamados sistemas baseados em menus. Os ponteiros de função são utilizados para selecionar qual função chamar para um item de menu particular.
- A função `strcpy` copia seu segundo argumento — uma string — para seu primeiro argumento — um array de caracteres. O programador deve assegurar que o array-alvo seja suficientemente grande para armazenar a string e seu caractere de terminação nulo.
- A função `strncpy` é equivalente à `strcpy`, exceto pelo fato de que uma chamada a `strncpy` especifica o número de caracteres a ser copiado da string para o array. O caractere de terminação nulo só será copiado se o número de caracteres a ser copiado for pelo menos um maior que o comprimento da string.
- A função `strcat` acrescenta seu segundo argumento de string — incluindo o caractere de terminação nulo — ao seu primeiro argumento de string. O primeiro caractere da segunda string substitui o caractere nulo (`'\0'`) da primeira string. O programador deve assegurar que o array-alvo utilizado para armazenar a primeira string é suficientemente grande para armazenar a primeira e a segunda strings.
- A função `strncat` é equivalente à `strcat`, exceto pelo fato de que uma chamada a `strncat` acrescenta um número especificado de caracteres da segunda string à primeira string. Um caractere de terminação nulo é acrescentado ao resultado.
- A função `strcmp` compara, caractere por caractere, seu primeiro argumento de string com seu segundo argumento de string. A função retornará zero se as strings forem iguais, um valor negativo se a primeira string for menor que a segunda string, e um valor positivo se a primeira string for maior que a segunda.

- A função `strncmp` é equivalente à `strcmp`, exceto pelo fato de que `strncmp` compara um número especificado de caracteres. Se o número de caracteres em uma das strings for menor que o número de caracteres especificado, `strncmp` compara os caracteres até o caractere nulo na string mais curta ser encontrado.
- Uma seqüência de chamadas a `strtok` divide uma string em tokens que são separados por caracteres contidos em um segundo argumento de string. A primeira chamada especifica a string tokenizada como o primeiro argumento, e as chamadas subseqüentes para continuar tokenizando a mesma string especificam `NULL` como o primeiro argumento. A função retorna um ponteiro para o token atual de cada chamada. Se não houver mais tokens quando `strtok` for chamada, `NULL` é retornado.
- A função `strlen` aceita uma string como um argumento e retorna o número de caracteres na string — o caractere de terminação nulo não é incluído no comprimento da string.

Terminologia

& (operador de endereço)
* (operador de desreferenciar ponteiro ou operador de indireção)
`'\0'` (caractere nulo)
adiamento indefinido
algoritmo de classificação por seleção
argumentos da linha de comando
aritmética de ponteiros
array de ponteiros para funções
array de strings
ASCII (American Standard Code for Information Interchange)
caractere de terminação nulo
caractere delimitador
caractere nulo (`'\0'`)
caracteres especiais
chamando funções por referência
código de caractere
comparando strings
concatenando strings
`const` com parâmetros de função
constante de caractere
constante string
cópia de string
copiando strings
decrementar um ponteiro
deslocamento para um ponteiro
desreferenciar um ponteiro
desreferenciar um ponteiro 0
EBCDIC (Extended Binary Coded Decimal Interchange Code)
endereço (&), operador de
função `getline` de `cin`
inanição
incrementar um ponteiro
indireção
intercambialidade de arrays e ponteiros
`islower`, função (`<cctype>`)
modificar endereço armazenado em variável ponteiro
modificar um ponteiro constante
operador de desreferência (*)
operador de desreferenciar ponteiro (*)
operador de indireção (*)
passagem por referência com argumentos de ponteiro
passagem por referência com argumentos de referência
ponteiro constante
ponteiro constante para dados constantes
ponteiro constante para dados não-constantes
ponteiro de função
ponteiro não-constante para dados constantes
ponteiro não-constante para dados não-constantes
ponteiro nulo
ponteiro para uma função
referência a dados constantes
referenciar diretamente um valor
referenciar elementos de array
referenciar um valor indiretamente
`size_t`, tipo
`sizeof`, operador
`strcat`, função do arquivo de cabeçalho `<cstring>`
`strcmp`, função do arquivo de cabeçalho `<cstring>`
`strcpy`, função do arquivo de cabeçalho `<cstring>`
string sendo tokenizada
string terminada por caractere nulo
strings baseadas em ponteiro
strings de tokenização
`strlen`, função do arquivo de cabeçalho `<cstring>`
`strncat`, função do arquivo de cabeçalho `<cstring>`
`strncmp`, função do arquivo de cabeçalho `<cstring>`
`strncpy`, função do arquivo de cabeçalho `<cstring>`
`strtok`, função do arquivo de cabeçalho `<cstring>`
subtração de ponteiro
token
`toupper`, função (`<cctype>`)

Exercícios de revisão

8.1 Complete cada uma das seguintes sentenças:
a) Um ponteiro é uma variável que contém como seu valor o(a) _____ de outra variável.
b) Os três valores que podem ser utilizados para inicializar um ponteiro são _____, _____ e _____.
c) O único inteiro que pode ser atribuído diretamente a um ponteiro é _____.

8.2 Determine se as seguintes sentenças são *verdadeiras* ou *falsas*. Se a resposta for *falsa*, explicar por quê.
a) O operador de endereço & pode ser aplicado somente a constantes e expressões.
b) Um ponteiro que é declarado como tipo `void *` pode ser desreferenciado.
c) Os ponteiros de tipos diferentes nunca podem ser atribuídos um ao outro sem a operação de coerção.

8.3 Para cada um dos itens a seguir, escreva instruções C++ que realizam a tarefa especificada. Suponha que números de dupla precisão com ponto flutuante sejam armazenados em oito bytes e que o endereço inicial do array esteja na posição 1002500 na memória. Cada parte do exercício deve utilizar os resultados de partes anteriores onde apropriado.

a) Declare um array do tipo `double` chamado `numbers` com 10 elementos, e inicialize os elementos para os valores `0.0`, `1.1`, `2.2`, ..., `9.9`. Suponha que a constante simbólica `SIZE` foi definida como `10`.
b) Declare um ponteiro `nPtr` que aponta para uma variável do tipo `double`.
c) Utilize uma instrução `for` para imprimir os elementos do array `numbers` usando notação de array subscrito. Imprima cada número com uma casa decimal de precisão à direita do ponto de fração decimal.
d) Escreva duas instruções separadas que atribuem, cada uma, o endereço inicial do array `numbers` à variável ponteiro `nPtr`.
e) Utilize uma instrução `for` para imprimir os elementos do array `numbers` utilizando a notação de ponteiro/deslocamento com o ponteiro `nPtr`.
f) Utilize uma instrução `for` para imprimir os elementos do array `numbers` utilizando a notação de ponteiro/deslocamento com o nome do array como o ponteiro.
g) Utilize uma instrução `for` para imprimir os elementos do array `numbers` utilizando a notação de ponteiro/subscrito com o ponteiro `nPtr`.
h) Referencie o quarto elemento do array `numbers` utilizando a notação de subscrito de array, a notação de ponteiro/deslocamento com o nome de array como o ponteiro, a notação de subscrito de ponteiro com `nPtr` e a notação de ponteiro/deslocamento com `nPtr`.
i) Supondo que `nPtr` aponta para o começo do array `numbers`, que endereço é referenciado por `nPtr + 8`? Que valor é armazenado nessa localização?
j) Supondo que `nPtr` aponta para `numbers[ 5 ]`, que endereço é referenciado por `nPtr` depois de `nPtr -= 4` ser executado? Qual valor é armazenado nessa localização?

8.4 Para cada uma das seguintes sentenças, escreva uma única instrução que realiza a tarefa especificada. Suponha que as variáveis de ponto flutuante `number1` e `number2` foram declaradas e que `number1` foi inicializado como `7.3`. Suponha que a variável `ptr` é do tipo `char *`. Suponha que os arrays `s1` e `s2` são, cada um, arrays `char` de 100 elementos que são inicializados com literais string.
a) Declare a variável `fPtr` como um ponteiro para um objeto do tipo `double`.
b) Atribua o endereço da variável `number1` à variável ponteiro `fPtr`.
c) Imprima o valor do objeto apontado por `fPtr`.
d) Atribua o valor do objeto apontado por `fPtr` à variável `number2`.
e) Imprima o valor de `number2`.
f) Imprima o endereço de `number1`.
g) Imprima o endereço armazenado em `fPtr`. O valor impresso é o mesmo endereço de `number1`?
h) Copie a string armazenada no array `s2` para o array `s1`.
i) Compare a string em `s1` com a string em `s2` e imprima o resultado.
j) Acrescente os primeiros 10 caracteres da string em `s2` à string em `s1`.
k) Determine o comprimento da string em `s1` e imprima o resultado.
l) Atribua à `ptr` a localização do primeiro token em `s2`. Os delimitadores de tokens são vírgulas (`,`).

8.5 Realize a tarefa especificada em cada uma das seguintes instruções:
a) Escreva o cabeçalho de função para uma função chamada `exchange` que aceita dois ponteiros para números de dupla precisão com ponto flutuante `x` e `y` como parâmetros e que não retorna um valor.
b) Escreva o protótipo de função para a função na parte (a).
c) Escreva o cabeçalho de função para uma função chamada `evaluate` que retorna um inteiro e aceita como parâmetros o inteiro `x` e um ponteiro para a função `poly`. A função `poly` aceita um parâmetro do tipo inteiro e retorna um inteiro.
d) Escreva o protótipo de função para a função na parte (c).
e) Escreva duas instruções que inicializam, cada uma, o array de caractere `vowel` com a string de vogais, `"AEIOU"`.

8.6 Encontre o erro em cada um dos seguintes segmentos de programa. Suponha as seguintes declarações e instruções:

```
int *zPtr;       // zPtr irá referenciar o array z
int *aPtr = 0;
void *sPtr = 0;
int number;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
```

a)
```
++zPtr;
```
b)
```
// utiliza ponteiro para obter o primeiro valor de array
number = zPtr;
```
c)
```
// atribui o array de 2 elementos (o valor 3) ao número
number = *zPtr[ 2 ];
```
d)
```
// imprime todo o array z
for ( int i = 0; i <= 5; i++ )
   cout << zPtr[ i ] << endl;
```
e)
```
// atribui o valor apontado por sPtr ao número
number = *sPtr;
```

f) `++z;`
g)
```
char s[ 10 ];
cout << strncpy( s, "hello", 5 ) << endl;
```
h)
```
char s[ 12 ];
strcpy( s, "Welcome Home" );
```
i)
```
if ( strcmp( string1, string2 ) )
   cout << "The strings are equal" << endl;
```

8.7 O que (se houver algo) é impresso quando cada uma das seguintes instruções é realizada? Se a instrução contiver um erro, descreva o erro e indique como corrigi-la. Suponha as seguintes declarações de variável:

```
char s1[ 50 ] = "jack";
char s2[ 50 ] = "jill";
char s3[ 50 ];
```

a) `cout << strcpy( s3, s2 ) << endl;`
b)
```
cout << strcat( strcat( strcpy( s3, s1 ), " and " ), s2 )
   << endl;
```
c) `cout << strlen( s1 ) + strlen( s2 ) << endl;`
d) `cout << strlen( s3 ) << endl;`

Respostas dos exercícios de revisão

8.1 a) endereço. b) 0, `NULL`, um endereço. c) 0.

8.2 a) Falsa. O operando do operador de endereço deve ser um *lvalue*; o operador de endereço não pode ser aplicado a constantes ou expressões que não resultam em referências.
b) Falsa. Um ponteiro para void não pode ser desreferenciado. Esse ponteiro não tem um tipo que permite ao compilador determinar o número de bytes de memória a desreferenciar e o tipo dos dados para os quais o ponteiro aponta.
c) Falsa. Qualquer tipo de ponteiro pode ser atribuído a ponteiros `void`. Os ponteiros do tipo `void` podem ser atribuídos a ponteiros de outros tipos somente com uma coerção de tipo explícita.

8.3 a) `double numbers[ SIZE ] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };`
b) `double *nPtr;`
c)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int i = 0; i < SIZE; i++ )
   cout << numbers[ i ] << ' ';
```
d)
```
nPtr = numbers;
nPtr = &numbers[ 0 ];
```
e)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int j = 0; j < SIZE; j++ )
   cout << *( nPtr + j ) << ' ';
```
f)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int k = 0; k < SIZE; k++ )
   cout << *( numbers + k ) << ' ';
```
g)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int m = 0; m < SIZE; m++ )
   cout << nPtr[ m ] << ' ';
```
h)
```
numbers[ 3 ]
*( numbers + 3 )
nPtr[ 3 ]
*( nPtr + 3 )
```
i) O endereço é `1002500 + 8 * 8 = 1002564`. O valor é `8.8`.
j) O endereço de `numbers[ 5 ]` é `1002500 + 5 * 8 = 1002540`.
O endereço de `nPtr -= 4` é `1002540 - 4 * 8 = 1002508`.
O valor nessa localização é `1.1`.

8.4 a) `double *fPtr;`
b) `fPtr = &number1;`
c) `cout << "The value of *fPtr is " << *fPtr << endl;`
d) `number2 = *fPtr;`
e) `cout << "The value of number2 is " << number2 << endl;`
f) `cout << "The address of number1 is " << &number1 << endl;`

g) `cout << "The address stored in fPtr is " << fPtr << endl;`
Sim, o valor é o mesmo.
h) `strcpy( s1, s2 );`
i) `cout << "strcmp(s1, s2) = " << strcmp( s1, s2 ) << endl;`
j) `strncat( s1, s2, 10 );`
k) `cout << "strlen(s1) = " << strlen( s1 ) << endl;`
l) `ptr = strtok( s2, "," );`

8.5 a)
```
void exchange( double *x, double *y )
```
b)
```
void exchange( double *, double * );
```
c)
```
int evaluate( int x, int (*poly)( int ) )
```
d)
```
int evaluate( int, int (*)( int ) );
```
e)
```
char vowel[] = "AEIOU";
char vowel[] = { 'A', 'E', 'I', 'O', 'U', '\0' };
```

8.6 a) Erro: `zPtr` não foi inicializado.
Correção: Inicialize `zPtr` com `zPtr = z;`
b) Erro: O ponteiro não é desreferenciado.
Correção: Altere a instrução para `number = *zPtr;`
c) Erro: `zPtr[ 2 ]` não é um ponteiro e não deve ser desreferenciado.
Correção: altere `*zPtr[ 2 ]` para `zPtr[ 2 ]`.
d) Erro: Referir-se a um elemento de array fora dos limites de array com subscrito de ponteiro.
Correção: Para impedir isso, mude o operador relacional na instrução `for` para <.
e) Erro: Desreferenciar um ponteiro void.
Correção: Para desreferenciar o ponteiro void, ele deve primeiro sofrer coerção para um ponteiro do tipo inteiro. Mude a instrução para `number = *static_cast< int * >( sPtr );`
f) Erro: Tentar modificar um nome de array com a aritmética de ponteiros.
Correção: Utilize uma variável ponteiro em vez do nome de array para realizar a aritmética de ponteiros ou utilize subscritos no nome de array para referenciar um elemento específico.
g) Erro: A função `strncpy` não escreve um caractere de terminação nulo para o array `s`, porque seu terceiro argumento é igual ao comprimento da string `"hello"`.
Correção: Torne 6 o terceiro argumento de `strncpy` ou atribua `'\0'` a `s[ 5 ]` para assegurar que o caractere de terminação nulo seja adicionado à string.
h) Erro: O array de caractere `s` não é grande o bastante para armazenar o caractere de terminação nulo.
Correção: Declare o array com mais elementos.
i) Erro: A função `strcmp` retornará 0 se as strings forem iguais; portanto, a condição na instrução `if` será falsa e a instrução de saída não será executada.
Correção: Compare explicitamente o resultado de `strcmp` com o 0 na condição da instrução `if`.

8.7 a)
```
jill
```
b)
```
jack and jill
```
c)
```
8
```
d)
```
13
```

Exercícios

8.8 Determine se as seguintes sentenças são *verdadeiras* ou *falsas*. Se *falsa*, explique por quê.
a) Dois ponteiros que apontam para arrays diferentes não podem ser comparados significativamente.
b) Como o nome de um array é um ponteiro para o primeiro elemento do array, os nomes de array podem ser manipulados precisamente da mesma maneira que os ponteiros.

8.9 Para cada um dos itens a seguir, escreva instruções C++ que realizam a tarefa especificada. Suponha que inteiros sem sinal estejam armazenados em dois bytes e que o endereço inicial do array esteja na posição 1002500 da memória.
a) Declare um array do tipo `unsigned int` chamado `values` com cinco elementos e inicialize os elementos para os inteiros pares de 2 a 10. Suponha que a constante simbólica `SIZE` foi definida como 5.
b) Declare um ponteiro `vPtr` que aponta para um objeto do tipo `unsigned int`.
c) Utilize uma instrução `for` para imprimir os elementos do array `values` usando notação de array subscrito.
d) Escreva duas instruções separadas que atribuem o endereço inicial do array `values` à variável ponteiro `vPtr`.
e) Utilize uma instrução `for` para imprimir os elementos do array `values` utilizando a notação de ponteiro/deslocamento.
f) Utilize uma instrução `for` para imprimir os elementos do array `values` utilizando a notação de ponteiro/deslocamento com o nome de array como o ponteiro.

g) Utilize uma instrução `for` para imprimir os elementos do array `values` utilizando subscritos no ponteiro para o array.
h) Referencie o quinto elemento de `values` utilizando a notação de subscrito de array, a notação de ponteiro/deslocamento com o nome de array como o ponteiro, a notação de subscrito de ponteiro e a notação de ponteiro/deslocamento.
i) Que endereço é referenciado por `vPtr + 3`? Que valor é armazenado nessa localização?
j) Supondo que `vPtr` aponte para `values[ 4 ]`, que endereço é referenciado por `vPtr -= 4`? Que valor é armazenado nessa localização?

8.10 Para cada um dos itens seguintes, escreva uma única instrução que realiza a tarefa indicada. Suponha que as variáveis do tipo inteiro `long value1` e `value2` tenham sido declaradas e que `value1` tenha sido inicializado como `200000`.
a) Declare a variável `longPtr` como um ponteiro para um objeto do tipo `long`.
b) Atribua o endereço da variável `value1` à variável ponteiro `longPtr`.
c) Imprima o valor do objeto apontado por `longPtr`.
d) Atribua o valor do objeto apontado por `longPtr` à variável `value2`.
e) Imprima o valor de `value2`.
f) Imprima o endereço de `value1`.
g) Imprima o endereço armazenado em `longPtr`. O valor impresso é o mesmo que o endereço de `value1`?

8.11 Realize a tarefa especificada por cada uma das seguintes instruções:
a) Escreva o cabeçalho de função para a função `zero` que aceita um parâmetro de array `bigIntegers` do tipo inteiro long e que não retorna um valor.
b) Escreva o protótipo de função para a função na parte (a).
c) Escreva o cabeçalho de função para a função `add1AndSum` que aceita um parâmetro de array `oneTooSmall` do tipo inteiro e que retorna um inteiro.
d) Escreva o protótipo de função para a função descrita na parte (c).

Nota: Os exercícios 8.12 a 8.15 são razoavelmente desafiadores. Uma vez que tiver resolvido esses problemas, você deve ser capaz de implementar muitos jogos de cartas populares.

8.12 Modifique o programa na Figura 8.27 de modo que a função de distribuição de cartas distribua uma mão de pôquer de cinco cartas. Então escreva as funções para realizar cada uma das seguintes tarefas:
a) Determine se a mão contém um par.
b) Determine se a mão contém dois pares.
c) Determine se a mão contém uma trinca (por exemplo, três valetes).
d) Determine se a mão contém uma quadra (por exemplo, quatro ases).
e) Determine se a mão contém um *flush* (isto é, todas cinco cartas do mesmo naipe).
f) Determine se a mão contém um *straight* (isto é, cinco cartas de valores consecutivos).

8.13 Utilize as funções desenvolvidas no Exercício 8.12 para escrever um programa que distribui duas mãos de pôquer de cinco cartas, avalia cada mão e determina qual é a melhor mão.

8.14 Modifique o programa desenvolvido no Exercício 8.13 de modo que você possa simular o carteador. A mão de cinco cartas do carteador é distribuída 'no escuro', então o jogador não pode vê-la. O programa deve então avaliar a mão do carteador e, com base na qualidade da mão, o carteador deve distribuir uma, duas ou três mais cartas para substituir o número correspondente de cartas desnecessárias na mão original. O programa então deve reavaliar a mão do carteador. [*Atenção:* Este é um problema difícil!]

8.15 Modifique o programa desenvolvido no Exercício 8.14 de modo que ele trate a mão do carteador, mas permita ao jogador decidir quais cartas ele quer substituir. O programa então deve avaliar ambas as mãos e determinar quem ganha. Agora utilize esse novo programa para disputar 20 jogos contra o computador. Quem ganha mais jogos, você ou o computador? Faça um de seus amigos disputar 20 jogos contra o computador. Quem ganha mais jogos? Com base nos resultados desses jogos, faça modificações apropriadas para refinar seu programa de jogar de pôquer. [*Nota:* Este também é um problema difícil.] Dispute mais 20 jogos. Seu programa modificado reproduziu um jogo melhor?

8.16 No programa de embaralhamento e distribuição de cartas das figuras 8.25–8.27, utilizamos intencionalmente um algoritmo ineficiente de embaralhamento que introduziu a possibilidade de adiamento indefinido. Nesse problema, você criará um algoritmo de embaralhamento de alto desempenho que evita o adiamento indefinido.

Modifique as figuras 8.25–8.27 como mostrado a seguir. Inicialize o array `deck` como mostrado na Figura 8.36. Modifique a função `shuffle` para iterar linha por linha e coluna por coluna pelo array, tocando uma vez em cada elemento. Cada elemento deve ser trocado por um elemento do array aleatoriamente selecionado. Imprima o array resultante para determinar se o baralho é embaralhado satisfatoriamente (como na Figura 8.37, por exemplo). Você pode querer que o programa chame a função `shuffle` várias vezes para assegurar um embaralhamento satisfatório.

Observe que, embora a abordagem nesse problema melhore o algoritmo de embaralhamento, o algoritmo de distribuição de cartas ainda requer pesquisar o array `deck` para encontrar a carta 1, depois a carta 2, a carta 3 e assim por diante. Pior ainda, mesmo depois de o

Array deck não embaralhado													
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Figura 8.36 Array deck não embaralhado.

Exemplo do array deck embaralhado													
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Figura 8.37 Exemplo do array deck embaralhado.

algoritmo de distribuição de cartas localizar e distribuir a carta, ele continua pesquisando pelo restante do baralho. Modifique o programa das figuras 8.25–8.27 de modo que, uma vez que uma carta é distribuída, nenhuma tentativa adicional de localizar esse número de carta seja feita, e o programa imediatamente prossiga com a distribuição da próxima carta.

8.17 (*Simulação: A tartaruga e a lebre*) Neste exercício, você recriará a clássica corrida da tartaruga e da lebre. Você utilizará geração de números aleatórios para desenvolver uma simulação desse memorável evento.

Nossos competidores começam a corrida no 'quadrado 1' de 70 quadrados. Cada quadrado representa uma possível posição ao longo do percurso da competição. A linha de chegada está no quadrado 70. O primeiro competidor a alcançar ou passar o quadrado 70 é recompensado com um cesto de cenouras frescas e alface. O percurso envolve subir uma montanha escorregadia, então ocasionalmente os competidores perdem terreno.

Há um relógio que toca uma vez por segundo. A cada tique do relógio, seu programa deve ajustar a posição dos animais de acordo com as regras na Figura 8.38.

Animal	Tipo de movimento	Porcentagem do tempo	Movimento real
Tartaruga	Caminhada rápida	50%	3 quadrados à direita
	Escorregão	20%	6 quadrados à esquerda
	Caminhada lenta	30%	1 quadrado à direita
Lebre	Parada	20%	Nenhum movimento
	Salto grande	20%	9 quadrados à direita
	Escorregão grande	10%	12 quadrados à esquerda
	Salto pequeno	30%	1 quadrado à direita
	Escorregão pequeno	20%	2 quadrados à esquerda

Figura 8.38 Regras para mover a tartaruga e a lebre.

Utilize variáveis para monitorar a posição dos animais (isto é, os números de posição são 1–70). Inicie cada animal na posição 1 (isto é, a 'partida'). Se um animal escorrega para a esquerda antes do quadrado 1, mova o animal de volta para quadrado 1.

Gere as porcentagens da tabela precedente produzindo um inteiro aleatório i no intervalo $1 \le i \le 10$. Para a tartaruga, realize uma 'caminhada rápida' quando $1 \le i \le 5$, um 'escorregão' quando $6 \le i \le 7$ ou uma 'caminhada lenta' quando $8 \le i \le 10$. Utilize uma técnica semelhante para mover a lebre.

Inicie a corrida imprimindo

```
BANG !!!!!
AND THEY'RE OFF !!!!!
[BANG !!!!!
E LÁ SE VÃO ELES !!!!!]
```

Para cada tique do relógio (isto é, cada repetição de um loop), imprima uma linha de 70 posições para mostrar a letra `T` na posição da tartaruga e a letra `H` na posição da lebre. Ocasionalmente, os competidores cairão no mesmo quadrado. Nesse caso, a tartaruga morde a lebre e seu programa deve imprimir `OUCH!!!` (Ai!) começando nessa posição. Todas as outras posições diferentes de `T`, `H` e o `OUCH!!!` (no caso de um empate) devem estar em branco.

Depois de imprimir cada linha, teste se algum animal alcançou ou passou o quadrado 70. Em caso positivo, imprima o vencedor e termine a simulação. Se a tartaruga ganhar, imprima `TORTOISE WINS!!! YAY!!!` [A TARTARUGA GANHOU!!! ÊHH!!!]. Se a lebre ganhar, imprima `Hare wins. Yuch.` [A Lebre ganhou. Uhh.]. Se ambos tiverem a mesma marcação no relógio, você pode querer favorecer a tartaruga (a 'coitadinha') ou querer imprimir `It's a tie` [Empate]. Se nenhum dos animais ganhar, realize o loop novamente para simular o próximo tique do relógio. Quando você estiver pronto para executar seu programa, monte um grupo de fãs para observar a corrida. Você ficará surpreso com o envolvimento da sua audiência!

Seção especial: construindo seu próprio computador

Nos próximos problemas, nós nos desviamos temporariamente do mundo da linguagem de programação de alto nível. Vamos 'abrir' um computador e examinar sua estrutura interna. Introduzimos programação de linguagem de máquina e escrevemos vários programas de linguagem de máquina. Para tornar essa uma experiência especialmente valiosa, construímos em seguida um computador (utilizando a *simulação* baseada em software) em que você pode executar seus programas de linguagem de máquina!

8.18 (*Programação de linguagem de máquina*) Vamos criar um computador que chamaremos de Simpletron. Como seu nome implica, é uma máquina simples, mas como logo veremos também é uma máquina poderosa. O Simpletron executa programas escritos na única linguagem que ele entende diretamente, isto é, a Simpletron Machine Language ou, abreviadamente, SML.

O Simpletron contém um *acumulador* — um 'registrador especial' em que as informações são colocadas antes de o Simpletron utilizá-las em cálculos ou examiná-las de várias maneiras. Todas as informações no Simpletron são tratadas em termos de *palavras*. Uma palavra é um número decimal de quatro dígitos com sinal como `+3364`, `-1293`, `+0007`, `-0001` etc. O Simpletron é equipado com uma memória de 100 palavras e essas palavras são referenciadas por seus números de posição `00`, `01`, ..., `99`.

Antes de executar um programa de SML, devemos *carregar*, ou colocar, o programa na memória. A primeira instrução de cada programa de SML sempre é colocada na posição `00`. O simulador começará a executar nessa posição.

Cada instrução escrita em SML ocupa uma palavra da memória do Simpletron; portanto, as instruções são números decimais de quatro dígitos com sinal. Suponha que o sinal de uma instrução de SML seja sempre positivo, mas o sinal de uma palavra de dados pode ser positivo ou negativo. Cada localização na memória de Simpletron pode conter uma instrução, um valor de dados utilizado por um programa ou uma área da memória não-utilizada (e portanto indefinida). Os primeiros dois dígitos de cada instrução do SML são o *código de operação* que especifica a operação a ser realizada. Os códigos de operação SML são mostrados na Figura 8.39.

Os últimos dois dígitos de uma instrução de SML são *operandos* — o endereço da posição da memória contendo a palavra à qual a operação se aplica.

Agora vamos considerar dois programas SML simples. O primeiro programa SML (Figura 8.40) lê dois números do teclado e calcula e imprime sua soma. A instrução `+1007` lê o primeiro número do teclado e o coloca na posição `07` (que foi inicializada como zero). A instrução `+1008` lê o próximo número na posição `08`. A instrução *load*, `+2007`, coloca (copia) o primeiro número no acumulador; e a instrução *add*, `+3008`, adiciona o segundo número ao número no acumulador. *Todas as instruções aritméticas da SML deixam seus resultados no acumulador*. A instrução store, `+2109`, coloca (copia) o resultado de volta na posição da memória `09`. Então a instrução *write*, `+1109`, pega o número e o imprime (como um número decimal de quatro dígitos com sinal). A instrução *halt*, `+4300`, termina a execução.

O programa SML na Figura 8.41 lê dois números a partir do teclado, então determina e imprime o maior valor. Note o uso da instrução `+4107` como uma transferência condicional de controle, muito parecida com a instrução `if` do C++.

Código de operação	Significado
Operações de entrada/saída	
`const int READ = 10;`	Lê uma palavra do teclado para uma posição específica da memória.
`const int WRITE = 11;`	Escreve na tela uma palavra de uma posição específica da memória.
Operações de carregamento e armazenamento	
`const int LOAD = 20;`	Carrega uma palavra de uma posição específica na memória para o acumulador.
`const int STORE = 21;`	Armazena uma palavra do acumulador para uma posição específica na memória.
Operações aritméticas	
`const int ADD = 30;`	Adiciona uma palavra de uma posição específica na memória à palavra no acumulador (deixa o resultado no acumulador).
`const int SUBTRACT = 31;`	Subtrai uma palavra de uma posição específica na memória da palavra no acumulador (deixa o resultado no acumulador)
`const int DIVIDE = 32;`	Divide uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
`const int MULTIPLY = 33;`	Multiplica uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
Operações de transferência de controle	
`const int BRANCH = 40;`	Desvia para uma posição específica na memória.
`const int BRANCHNEG = 41;`	Desvia para uma posição específica na memória se o acumulador for negativo.
`const int BRANCHZERO = 42;`	Desvia para uma posição específica na memória se o acumulador for zero.
`const int HALT = 43;`	Suspende — o programa completou sua tarefa.

Figura 8.39 Códigos de operação de Simpletron Machine Language (SML).

Posição	Número	Instrução
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Figura 8.40 Exemplo de SML 1.

Posição	Número	Instrução
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Desvio negativo para 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Figura 8.41 Exemplo de SML 2.

Agora escreva programas de SML para realizar cada uma das seguintes tarefas:

a) Utilize um loop controlado por sentinela para ler números positivos e calcular e imprimir sua soma. Termine a entrada quando um número negativo for inserido.
b) Utilize um loop controlado por contador para ler sete números, alguns negativos e alguns positivos e compute e imprima sua média.
c) Leia uma série de números, e determine e imprima o maior número. O primeiro número lido indica quantos números devem ser processados.

8.19 (*Simulador de computador*) À primeira vista, pode parecer pretensioso, mas nesse problema você construirá seu próprio computador. Não, você não irá soldar componentes. Em vez disso, você utilizará a poderosa técnica de *simulação baseada em software* para criar um *modelo de software* do Simpletron. Você não se decepcionará. Seu Simpletron Simulator transformará o computador que você está utilizando em um Simpletron, e você realmente será capaz de executar, testar e depurar os programas de SML escritos no Exercício 8.18.

Quando você executar seu simulador Simpletron, ele deve começar imprimindo

```
*** Bem vindo ao Simpletron!                              ***
*** Por favor insira uma instrução                        ***
*** (ou data word) por vez em seu programa. Eu vou digitar ***
*** o número de alocação e o ponto de interrogação (?).    ***
*** Então você digita a palavra para a alocação.           ***
*** Digite o número -99999 para parar de inserir dados     ***
*** no seu programa.                                       ***
```

Seu programa deve simular a memória do Simpletron com um de um único subscrito, o array `memory` de 100 elementos. Agora assuma que o simulador está executando e vamos examinar o diálogo à medida que inserimos o programa do Exemplo 2 do Exercício 8.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins  ***
```

Observe que os números à direita de cada ? no diálogo anterior representam as instruções do programa SML inseridos pelo usuário.

O programa de SML agora foi colocado (ou carregado) no array `memory`. Agora o Simpletron executa seu programa SML. A execução inicia com a instrução na posição 00 e, como o C++, continua seqüencialmente, a menos que dirigido para alguma outra parte do programa por uma transferência de controle.

Utilize a variável `accumulator` para representar o registrador acumulador. Utilize a variável `counter` para monitorar a posição na memória que contém a instrução sendo realizada. Utilize a variável `operationCode` para indicar a operação que está sendo atualmente realizada (isto é, os dois dígitos esquerdos da palavra da instrução). Utilize a variável `operand` para indicar a posição da memória em que a instrução atual opera. Portanto, `operand` são os dois dígitos mais à direita da instrução sendo atualmente realizada. Não execute instruções diretamente de memória. Mais precisamente, transfira a próxima instrução que será realizada da memória para uma variável chamada `instructionRegister`. Então 'pegue' os dois dígitos esquerdos e os coloque em `operationCode` e 'pegue' os dois dígitos direitos e os coloque no `operand`. Quando o Simpletron começa a executar, todos os registradores especiais são inicializados como zero.

Agora vamos 'percorrer' a execução da primeira instrução de SML, +1009 na posição da memória 00. Isso é chamado um *ciclo de execução de instrução*.

O `counter` informa a posição da próxima instrução que será realizada. Realizamos uma *busca* (*fetch*) do conteúdo dessa posição a partir de `memory` utilizando a instrução C++

```
instructionRegister = memory[ counter ];
```

O código de operação e o operando são extraídos do registrador de instrução pelas instruções

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Agora o Simpletron deve determinar que o código de operação é na realidade uma *read* (versus um *write*, um *load* etc.) Um `switch` diferencia entre as 12 operações de SML.

Na instrução `switch`, o comportamento de várias instruções SML é simulado como mostrado na Figura 8.42 (deixamos os outros para o leitor).

A instrução *halt* também faz com que o Simpletron imprima o nome e o conteúdo de cada registrador, bem como o conteúdo completo da memória. Esse tipo de impressão é freqüentemente chamado de *dump de computador*. Para ajudá-lo a programar sua função de dump, um formato de dump de exemplo é mostrado na Figura 8.43. Observe que um dump, depois de executar um programa Simpletron, mostraria os valores reais das instruções e os valores dos dados no momento em que a execução terminasse. Para formatar os números com seu sinal como mostrado no dump, utilize o manipulador de fluxo **`showpos`**. Para desativar a exibição do sinal, utilize o manipulador de fluxo **`noshowpos`**. Para números que tenham menos de quatro dígitos, você pode formatar números com zeros à esquerda entre o sinal e o valor utilizando a seguinte instrução antes de gerar a saída do valor:

```
cout << setfill( '0' ) << internal;
```

O manipulador de fluxo parametrizado **`setfill`** (do cabeçalho `<iomanip>`) especifica o caractere de preenchimento que irá aparecer entre o sinal e o valor quando um número for exibido com uma largura de campo de cinco caracteres mas sem quatro dígitos. (Uma posição na largura de campo é reservada para o sinal.) O manipulador de fluxo **`internal`** indica que os caracteres de preenchimento aparecem entre o sinal e o valor numérico.

Vamos prosseguir com a execução da primeira instrução de nosso programa — +1009 na posição 00. Como indicamos, a instrução `switch` simula isso executando a instrução C++.

```
cin >> memory[ operand ];
```

Um ponto de interrogação (?) deve ser exibido na tela antes de a instrução `cin` executar para solicitar a entrada ao usuário. O Simpletron espera o usuário digitar um valor e pressionar a tecla *Enter*. O valor então é lido na posição 09.

read:	`cin >> memory[ operand ];`
load:	`accumulator = memory[ operand ];`
add:	`accumulator += memory[ operand ];`
branch:	Discutiremos as instruções de desvio brevemente.
halt:	Essa instrução imprime a mensagem `*** Simpletron execution terminated ***`

Figura 8.42 Comportamento de instruções SML.

```
REGISTERS:
accumulator          +0000
counter                00
instructionRegister  +0000
operationCode          00
operand                00

MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Figura 8.43 Um dump de exemplo.

Neste ponto, a simulação da primeira instrução é concluída. Tudo o que resta é preparar o Simpletron para executar a próxima instrução. A instrução que acabamos de realizar não era uma transferência de controle, portanto precisamos meramente incrementar os registradores de contadores de instruções como mostrado a seguir:

```
++counter;
```

Isso completa a execução simulada da primeira instrução. O processo inteiro (isto é, o ciclo de execução de instrução) começa de novo com a busca da próxima instrução a executar.

Agora vamos considerar como simular as instruções de desvio (isto é, as transferências de controle). Tudo o que precisamos fazer é ajustar o valor no contador de instrução apropriadamente. Portanto, a instrução de desvio incondicional (`40`) é simulada no `switch` como

```
counter = operand;
```

A instrução condicional 'desvie se acumulador for zero' é simulada como

```
if ( accumulator == 0 )
   counter = operand;
```

Nesse ponto, você deve implementar seu Simpletron Simulator e executar cada um dos programas em SML que você escreveu no Exercício 8.18. Você pode sofisticar a SML com recursos adicionais e adaptá-los ao seu simulador.

Seu simulador deve verificar vários tipos de erros. Durante a fase de carregamento do programa, por exemplo, cada número que usuário digita no Simpletron `memory` deve estar no intervalo `-9999` a `+9999`. Seu simulador deve utilizar um loop `while` para testar se cada número inserido está nesse intervalo e, se não tiver, continuar pedindo para ao usuário tentar novamente o número até inserir um número correto.

Durante a fase de execução, seu simulador deve verificar vários erros sérios, como tentativas de divisão por zero, tentativas de execução de códigos de operação inválidos, estouros de acumulador (isto é, operações aritméticas resultando em valores maiores que `+9999` ou menores que `-9999`) e assim por diante. Os erros sérios são chamados **erros fatais**. Quando um erro fatal é detectado, seu simulador deve imprimir uma mensagem de erro como

```
*** Tentou dividir por zero ***
*** A execução do Simpletron foi interrompida ***
```

e deve imprimir um dump de computador completo no formato discutido previamente. Isso ajudará o usuário localizar o erro no programa.

Mais exercícios sobre ponteiros

8.20 Modifique o programa de embaralhamento e distribuição de cartas das figuras 8.25–8.27 para que as operações de embaralhamento e distribuição sejam realizadas pela mesma função (`shuffleAndDeal`). A função deve conter uma instrução de loop aninhada que seja semelhante à função `shuffle` na Figura 8.26.

8.21 O que esse programa faz?

```
// Ex. 8.21: ex08_21.cpp
// O que esse programa faz?
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

void mystery1( char *, const char * ); // protótipo

int main()
{
   char string1[ 80 ];
   char string2[ 80 ];

   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   mystery1( string1, string2 );
   cout << string1 << endl;
   return 0; // indica terminação bem-sucedida
} // fim de main

// O que essa função faz?
void mystery1( char *s1, const char *s2 )
{
   while ( *s1 != '\0' )
      ++s1;

   for ( ; *s1 = *s2; s1++, s2++ )
      ; // estrutura vazia
} // fim da função mystery1
```

8.22 O que esse programa faz?

```
// Ex. 8.22: ex08_22.cpp
// O que esse programa faz?
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int mystery2( const char * ); // protótipo

int main()
{
   char string1[ 80 ];

   cout << "Enter a string: ";
   cin >> string1;
   cout << mystery2( string1 ) << endl;
   return 0; // indica terminação bem-sucedida
} // fim de main

// O que essa função faz?
int mystery2( const char *s )
{
```

```
23     int x;
24
25     for ( x = 0; *s != '\0'; s++ )
26        ++x;
27
28     return x;
29  } // fim da função mystery2
```

8.23 Localize o erro em cada um dos seguintes segmentos de código. Se for possível corrigir o erro, explique como.

a)
```
int *number;
cout << number << endl;
```

b)
```
double *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```

d)
```
char s[] = "this is a character array";
for ( ; *s != '\0'; s++)
   cout << *s << ' ';
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

f)
```
double x = 19.34;
double xPtr = &x;
cout << xPtr << endl;
```

g)
```
char *s;
cout << s << endl;
```

8.24 (*Quicksort – Técnica de classificação recursiva*) Você viu previamente as técnicas de classificação do tipo *bucket sort* e classificação por seleção. Agora apresentamos a técnica de classificação recursiva chamada Quicksort. O algoritmo básico para um array de um único subscrito de valores é como segue:

a) *Passo de particionamento:* Pegue o primeiro elemento do array não-classificado e determine sua localização final no array classificado (isto é, todos os valores à esquerda do elemento no array são menores que o elemento e todos os valores à direita do elemento no array são maiores que o elemento). Agora temos um elemento em sua posição adequada e dois subarrays não-classificados.

b) *Passo recursivo:* Realize o *Passo 1* em cada subarray não-classificado.

Toda vez que o *Passo 1* for realizado em um subarray, outro elemento é colocado em sua posição final no array classificado, e dois subarrays não-classificados são criados. Quando um subarray consiste em um elemento, esse subarray deve ser classificado; portanto, esse elemento está em sua localização final.

O algoritmo básico parece suficientemente simples, mas como determinamos a posição final do primeiro elemento de cada subarray? Como um exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de particionamento — ele será colocado em sua localização final no array classificado) :

37 2 6 4 89 8 10 12 68 45

a) Iniciando do elemento mais à direita do array, compare cada elemento com **37** até um elemento menor que **37** ser encontrado. Então troque **37** e esse elemento. O primeiro elemento menor que **37** é 12, então **37** e 12 são permutados. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 89 8 10 **37** 68 45

O elemento 12 está em itálico para indicar que acabou ser permutado com 37.

b) Iniciando à esquerda do array, mas começando com o elemento depois de 12, compare cada elemento com **37** até que um elemento maior que **37** seja localizado. Então troque **37** e esse elemento. O primeiro elemento maior que **37** é 89, então **37** e 89 foram permutados. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 **37** 8 10 *89* 68 45

c) Iniciando à direita, mas começando com o elemento antes de 89, compare cada elemento com **37** até que um elemento menor que **37** seja localizado. Então troque **37** e esse elemento. O primeiro elemento menor que **37** é 10, então **37** e 10 são permutados. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 *10* 8 **37** 89 68 45

d) Iniciando à esquerda, mas começando com o elemento depois de 10, compare cada elemento com **37** até que elemento maior que **37** seja localizado. Então troque **37** e esse elemento. Não há mais elementos maiores que **37**, então, quando comparamos **37** com ele mesmo sabemos que **37** foi colocado na sua localização final do array classificado.

Depois que o particionamento foi aplicado ao array, restam dois subarrays não classificados. O subarray com valores menores que 37 contém 12, 2, 6, 4, 10 e 8. O subarray com valores maiores que 37 contém 89, 68 e 45. A classificação continua com ambos os subarrays sendo particionados da mesma maneira que o array original.

Com base na discussão precedente, escreva uma função recursiva `quickSort` para classificar um array de inteiros de um único subscrito. A função deve receber como argumentos um array de inteiros, um subscrito inicial e um subscrito final. A função `partition` deve ser chamada por `quickSort` para realizar o passo de particionamento.

8.25 (*Percorrendo um labirinto*) A grade de cerquilhas (#) e pontos (`.`) na Figura 8.44 é uma representação de um array bidimensional de um labirinto. No array bidimensional, as cerquilhas representam as paredes do labirinto e os pontos representam quadrados nos possíveis caminhos pelo labirinto. Movimentos são permitidos apenas nas posições do array que contiverem um ponto.

Há um algoritmo simples para percorrer um labirinto que garante a localização da saída (supondo que existe uma saída). Se não houver uma saída, você chegará à localização inicial novamente. Coloque a sua mão direita na parede à sua direita e comece a andar para a frente. Nunca tire a sua mão da parede. Se o labirinto virar para a direita, siga a parede à direita. Contanto que você não remova a sua mão da parede, você acabará chegando à saída do labirinto. É possível que haja um caminho mais curto do que o que você tomou, mas a saída do labirinto é garantida se o algoritmo for seguido.

Escreva a função recursiva `mazeTraverse` para percorrer o labirinto. A função deve receber argumentos que incluem um array de 12 por 12 caracteres que representa o labirinto e a localização inicial do labirinto. À medida que `mazeTraverse` tenta localizar a saída do labirinto, ele deve colocar o caractere `X` em cada quadrado no caminho. A função deve exibir o labirinto depois de cada movimento de modo que o usuário possa observar enquanto o problema da saída do labirinto é resolvido.

8.26 (*Gerando labirintos aleatoriamente*) Escreva uma função `mazeGenerator` que recebe como um argumento um array de 12 caracteres bidimensional e produza aleatoriamente um labirinto. A função também deve fornecer as posições inicial e final do labirinto. Experimente a função `mazeTraverse` do Exercício 8.25, utilizando vários labirintos gerados aleatoriamente.

8.27 (*Labirintos de qualquer tamanho*) Generalize as funções `mazeTraverse` e `mazeGenerator` dos exercícios 8.25 e 8.26 para processar labirintos de qualquer largura e altura.

8.28 (*Modificações para o Simpletron Simulator*) No Exercício 8.19, você escreveu uma simulação de software de um computador que executa programas escritos em Simpletron Machine Language (SML). Nesse exercício, são propostas várias modificações e aprimoramentos para o Simpletron Simulator. Nos exercícios 21.26 e 21.27, propomos construir um compilador que converte programas escritos em uma linguagem de programação de alto nível (uma variação do BASIC) em SML. Algumas das seguintes modificações e melhorias podem ser necessárias para executar os programas produzidos pelo compilador. [*Nota:* Algumas modificações podem entrar em conflito com outras e, portanto, devem ser feitas separadamente.]

a) Estender a memória do Simpletron Simulator para conter 1.000 posições de memória a fim de permitir que o Simpletron trate programas maiores.
b) Permitir que o simulador realize cálculos de módulo. Isso requer uma instrução SML adicional.
c) Permitir que o simulador realize cálculos de exponenciação. Isso requer uma instrução SML adicional.
d) Modificar o simulador para utilizar valores hexadecimais em vez de valores inteiros para representar instruções SML.
e) Modificar o simulador para permitir saída de uma nova linha. Isso requer uma instrução SML adicional.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Figura 8.44 Representação de um labirinto com um array bidimensional.

f) Modificar o simulador para processar valores de ponto flutuante além de valores inteiros.
g) Modificar o simulador para tratar entrada de string. [*Dica:* Cada palavra do Simpletron pode ser dividida em dois grupos, cada uma armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente ASCII decimal de um caractere. Adicione uma instrução de linguagem de máquina que irá inserir uma string e armazenar a string inicial em uma posição da memória específica do Simpletron. A primeira metade da palavra nessa posição será uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada meia-palavra sucessiva contém um caractere ASCII como dois dígitos decimais expressos. A instrução de linguagem de máquina converte cada caractere em seu equivalente ASCII e atribui a ele uma meia-palavra.]
h) Modificar o simulador para tratar saída de strings armazenadas no formato da parte (g). [*Dica:* Adicione uma instrução de linguagem de máquina que imprimirá uma string inicial em certa posição da memória de Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada meia-palavra sucessiva contém um caractere ASCII como dois dígitos decimais expressos. Cada meia-palavra sucessiva contém um caractere de ASCII como dois dígitos decimais expressos.]
i) Modifique o simulador para incluir a instrução `SML_DEBUG` que imprime um dump de memória depois que cada instrução executa. Forneça ao `SML_DEBUG` um código de operação de `44`. A palavra `+4401` ativa o modo de depuração, e `+4400` o desativa.

8.29 O que esse programa faz?

```
1  // Ex. 8.29: ex08_29.cpp
2  // O que esse programa faz?
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  bool mystery3( const char *, const char * ); // protótipo
9
10 int main()
11 {
12    char string1[ 80 ], string2[ 80 ];
13
14    cout << "Enter two strings: ";
15    cin >> string1 >> string2;
16    cout << "The result is " << mystery3( string1, string2 ) << endl;
17    return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // O que essa função faz?
21 bool mystery3( const char *s1, const char *s2 )
22 {
23    for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )
24
25       if ( *s1 != *s2 )
26          return false;
27
28    return true;
29 } // fim da função mystery3
```

Exercícios de manipulação de string

[*Nota:* Os exercícios a seguir devem ser implementados utilizando strings baseadas em ponteiro no estilo C.]

8.30 Escreva um programa que utiliza a função `strcmp` para comparar duas strings inseridas pelo usuário. O programa deve declarar se a primeira string é menor que, igual a ou maior que a segunda string.

8.31 Escreva um programa que utiliza a função `strncmp` para comparar duas strings inseridas pelo usuário. O programa deve inserir o número de caracteres a ser comparado. O programa deve declarar se a primeira string é menor que, igual a ou maior que a segunda string.

8.32 Escreva um programa que utiliza a geração de números aleatórios para criar frases. O programa deve utilizar quatro arrays de ponteiros para `char` chamados `article`, `noun`, `verb` e `preposition`. O programa deve criar uma frase selecionando aleatoriamente uma palavra de cada array na seguinte ordem: `article`, `noun`, `verb`, `preposition`, `article` e `noun`. Quando cada palavra é selecionada, ela deve

ser concatenada pelas palavras anteriores em um array suficientemente grande para armazenar a frase inteira. As palavras devem ser separadas por espaços. Quando a frase final for enviada para saída, ela deve iniciar com uma letra maiúscula e terminar com um ponto. O programa deve gerar 20 dessas frases.

Os arrays devem ser preenchidos como segue: O array `article` deve conter os artigos `"the"`, `"a"`, `"one"`, `"some"` e `"any"`; e array `noun` deve conter os nomes `"boy"`, `"girl"`, `"dog"`, `"town"` e `"car"`; o array `verb` deve conter os verbos `"drove"`, `"jumped"`, `"ran"`, `"walked"` e `"skipped"`; o array `preposition` deve conter as preposições `"to"`, `"from"`, `"over"`, `"under"` e `"on"`.

Depois de concluir o programa, modifique-o para produzir uma pequena história com várias dessas frases. (E que tal um escritor de teses aleatório?!)

8.33 (*Limericks*) Um limerick é um poema humorístico de cinco versos em que a primeira e a segunda linha rimam com a quinta, e a terceira linha rima com a quarta. Utilizando técnicas semelhantes àquelas desenvolvidas no Exercício 8.32, escreva um programa C++ que produz limericks aleatórios. Aprimorar esse programa para produzir bons limericks é um problema desafiador, mas o resultado vale o esforço!

8.34 Escreva um programa que codifica frases da língua inglesa em 'latim de porco'. O latim de porco é uma forma de linguagem codificada freqüentemente utilizada por diversão. Existem muitas variações nos métodos utilizados para formar frases em latim de porco. Para simplificar, utilize o seguinte algoritmo: Para formar uma frase em latim de porco a partir do inglês, tokenize a frase em palavras com função `strtok`. Para traduzir cada palavra inglesa em uma palavra do latim de porco, coloque a primeira letra da palavra inglesa no final da palavra e adicione as letras `"ay"`. Portanto, a palavra `"jump"` torna-se `"umpjay"`, a palavra `"the"` torna-se `"hetay"` e a palavra `"computer"` torna-se `"omputercay"`. Os espaços entre as palavras permanecem iguais. Suponha que a escrita do inglês consista em palavras separadas por espaços, não haja nenhuma marcação de pontuação e todas as palavras tenham duas ou mais letras. A função `printLatinWord` deve exibir cada palavra. [*Dica:* Toda vez que um token for localizado em uma chamada para `strtok`, passe o ponteiro do token para a função `printLatinWord` e imprima a palavra em latim de porco.]

8.35 Escreva um programa que insere um número de telefone como uma string na forma `(555) 555-5555`. O programa deve utilizar a função `strtok` para extrair o código de área como um token, os três primeiros dígitos do número de telefone como um segundo token e os últimos quatro dígitos do número de telefone como um terceiro token. Os sete dígitos do número de telefone devem ser concatenados em uma string. O código de área e o número de telefone devem ser impressos.

8.36 Escreva um programa que insere uma linha de texto, tokeniza a linha com a função `strtok` e gera saída dos tokens na ordem inversa.

8.37 Utilize as funções de comparação de string discutidas na Seção 8.13.2 e as técnicas para classificar arrays desenvolvidas no Capítulo 7 para escrever um programa que alfabetiza uma lista de strings. Utilize os nomes de 10 ou 15 bairros em sua área como dados para seu programa.

8.38 Escreva duas versões de cada função de cópia de string e concatenação de string na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.

8.39 Escreva duas versões de cada função de comparação de string na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.

8.40 Escreva duas versões da função `strlen` na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.

Seção especial: exercícios avançados de manipulação de string

Os exercícios precedentes são voltados para texto e projetados para testar o entendimento do leitor de conceitos fundamentais de manipulação de string. Esta seção inclui uma coleção de exercícios de manipulação de string avançados e intermediários. O leitor deve achar esses problemas desafiadores, e até divertidos. Os problemas variam consideravelmente em dificuldade. Alguns requerem uma hora ou duas para escrever e implementar o programa. Outros são úteis para atribuições de laboratório que talvez requeiram duas ou três semanas de estudo e implementação. Alguns são projetos de conclusão de curso desafiadores.

8.41 *(Análise de texto)* A disponibilidade de computadores com capacidades de manipulação de string resultou em algumas abordagens bastante interessantes para analisar textos de grandes autores. Muita atenção foi dada à polêmica de que William Shakespeare não teria existido de fato. Alguns especialistas acreditam que há evidência substancial para indicar que Christopher Marlowe ou outros foram os verdadeiros autores das obras-primas atribuídas a Shakespeare. Os pesquisadores têm utilizado computadores para encontrar semelhanças na escrita desses autores. Esse exercício examina três métodos para analisar textos com um computador. Observe que milhares de textos, inclusive os de Shakespeare, estão disponíveis on-line em `www.gutenberg.org`.

a) Escreva um programa que lê várias linhas de texto do teclado e imprime uma tabela que indica o número de ocorrências de cada letra do alfabeto no texto. Por exemplo, a frase

```
To be, or not to be: that is the question:
```

contém um 'a', dois 'b', nenhum 'c' etc.

b) Escreva um programa que lê várias linhas de texto e imprime uma tabela que indica o número de palavras de uma letra, palavras de duas letras, palavras de três letras etc. que aparecem no texto. Por exemplo, a frase

```
Whether 'tis nobler in the mind to suffer
```

contém os seguintes comprimentos de palavra e ocorrências:

Comprimento de palavra	Ocorrências
1	0
2	2
3	1
4	2 (incluindo `'tis`)
5	0
6	2
7	1

c) Escreva um programa que lê várias linhas de texto e imprime uma tabela que indica o número de ocorrências de cada palavra diferente no texto. A primeira versão de seu programa deve incluir as palavras na tabela na mesma ordem em que elas aparecem no texto. Por exemplo, as linhas

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

Contém a palavra 'to' três vezes, a palavra 'be' duas vezes, a palavra 'or' uma vez etc. Então você deve tentar uma impressão mais interessante (e útil) em que as palavras são classificadas alfabeticamente.

8.42 (*Processamento de textos*) Uma função importante em sistemas de processamento de texto é a *justificação de texto* — que é o alinhamento à esquerda e à direita de uma página simultaneamente. Isso gera um documento com um *layout* profissional que dá a impressão de que o documento foi editorado em vez de preparado em uma máquina de escrever. A justificação de texto pode ser realizada em sistemas de computador inserindo caracteres de espaço em branco entre cada uma das palavras em uma linha para que a última palavra de cada linha seja alinhada com a margem direita.

Escreva um programa que lê várias linhas de texto e imprime esse texto no formato alinhado. Suponha que o texto deva ser impresso no papel de 8 1/2 polegadas de largura e que as margens de uma polegada sejam adotadas nos lados esquerdo e direito. Suponha que o computador imprima 10 caracteres por polegada horizontal. Portanto, seu programa deve imprimir 6-½ polegadas de texto ou 65 caracteres por linha.

8.43 (*Imprimindo datas em vários formatos*) As datas são comumente impressas em vários formatos diferentes na correspondência comercial. Dois dos formatos mais comuns em inglês são

```
07/21/1955
July 21, 1955
```

Escreva um programa que lê uma data no primeiro formato e imprime essa data no segundo formato.

8.44 (*Proteção de cheque*) Os computadores são freqüentemente empregados em sistemas de verificação de cheque como aplicativos de folha de pagamento e contas a pagar. Circulam muitas histórias estranhas relacionadas com cheques de pagamento de salário semanal impressos (por equívoco) com quantias acima de $ 1 milhão. Valores de cheque absurdos são impressos por sistemas computadorizados de preenchimento de cheque devido a erro humano ou falha mecânica. Os projetistas de sistemas embutem controles em seus sistemas para evitar a emissão desses cheques errados.

Outro problema sério é a alteração intencional de um valor do cheque por alguém que pretende receber um cheque fraudulentamente. Para evitar que uma quantia monetária seja alterada, a maioria dos sistemas computadorizados de preenchimento de cheque emprega uma técnica chamada *proteção de cheque*.

Cheques projetados para imprimir por computador contém um número fixo de espaços em que o computador pode imprimir uma quantia. Suponha que um cheque de pagamento contenha oito espaços em branco em que o computador deve imprimir a quantidade de um cheque de pagamento semanal. Se a quantidade for grande, então todos os oito espaços serão preenchidos, por exemplo:

```
1,230.60    (valor do cheque)
--------
12345678    (números de posição)
```

Por outro lado, se a quantidade for menor que $ 1000, então vários dos espaços seriam comumente deixados em branco. Por exemplo,

```
   99.87
--------
12345678
```

contém três espaços em branco. Se um cheque é impresso com espaços em branco, é mais fácil para alguém alterar o valor do cheque. Para evitar que um cheque seja alterado, muitos sistemas de preenchimento de cheque inserem *asteriscos iniciais* para proteger o valor como segue:

```
***99.87
--------
12345678
```

Escreva um programa que aceita como entrada uma quantia monetária para ser impressa em um cheque e então imprime o valor em formato de cheque protegido com asteriscos iniciais se necessário. Suponha que nove espaços estão disponíveis para imprimir uma quantia.

8.45 (*Valor de um cheque por extenso*) Continuando a discussão do exemplo anterior, reiteramos a importância de projetar sistemas de preenchimento de cheque para evitar alteração de seus valores. Um método comum de segurança requer que o valor do cheque seja escrito tanto em números como 'por extenso'. Mesmo se alguém for capaz de alterar o valor numérico do cheque, é extremamente difícil alterar o valor por extenso.

Escreva um programa que aceita como entrada o valor do cheque em números e gera como saída o valor por extenso correspondente. Seu programa deve ser capaz de tratar valores de cheques tão altos quanto $ 99,99. Por exemplo, o valor 112,43 deve ser escrito assim

```
ONE HUNDRED TWELVE and 43/100
```

8.46 (*Código Morse*) Talvez o mais famoso de todos os esquemas de codificação seja o código Morse, desenvolvido por Samuel Morse em 1832 para utilização com o sistema de telégrafo. O código Morse atribui uma série de pontos e traços a cada letra do alfabeto, a cada dígito e a alguns caracteres especiais (como ponto, vírgula, dois-pontos e ponto-e-vírgula). Em sistemas voltados para áudio, o ponto representa um som breve e o traço representa um som longo. Outras representações de pontos e traços são utilizadas com sistemas baseados em sinais luminosos e sistemas baseados em sinais de bandeira.

A separação entre palavras é indicada por um espaço ou, simplesmente, pela ausência de um ponto ou traço. Em um sistema baseado em áudio, um espaço é indicado por um ponto breve de tempo durante o qual nenhum som é transmitido. A versão internacional do código Morse aparece na Figura 8.45.

Escreva um programa que lê uma frase em inglês e a codifica em código Morse. Escreva também um programa que lê uma frase em código Morse e a converte no equivalente em inglês. Utilize um espaço em branco entre cada letra codificada em Morse e três espaços em branco entre cada palavra codificada em Morse.

Caractere	Código	Caractere	Código	Dígito	Código
A	.-	N	-.	1	.----
B	-...	O	---	2	..---
C	-.-.	P	.--.	3	...--
D	-..	Q	--.-	4	....-
E	.	R	.-.	5	.....
F	..-.	S	...	6	-....
G	--.	T	-	7	--...
H	....	U	..-	8	---..
I	..	V	...-	9	----.
J	.---	W	.--	0	-----
K	-.-	X	-..-		
L	.-..	Y	-.--		
M	--	Z	--..		

Figura 8.45 Alfabeto do código Morse.

8.47 (*Um programa de conversão métrica*) Escreva um programa que ajudará o usuário a realizar conversões métricas. Seu programa deve permitir que o usuário especifique o nome das unidades como strings (isto é, centímetros, litros, gramas etc. para o sistema métrico e polegadas, quartos, libras etc. para o sistema inglês) e deve responder a perguntas simples como

```
"Quantas polegadas temos em 2 metros?"
"Quantos litros temos em 10 quartos?"
```

Seu programa deve reconhecer conversões inválidas. Por exemplo, a pergunta

```
"Quantos pés temos em 5 quilos?"
```

não é significativo, porque `"pés"` são unidades de comprimento, enquanto `"quilos"` são unidades de peso.

Um projeto desafiador de manipulação de string

8.48 (*Um gerador de palavras cruzadas*) A maioria das pessoas já brincou de palavras cruzadas, mas poucos tentaram gerar um jogo de palavras cruzadas. Gerar um programa de palavras cruzadas é um problema difícil. Isso é sugerido aqui como um projeto de manipulação de string que exige bastante sofisticação e esforço. Há muitas questões que o programador deve resolver para fazer funcionar até mesmo o mais simples programa gerador de palavras cruzadas. Por exemplo, como se representa a grade das palavras cruzadas dentro do computador? Você deve utilizar uma série de strings ou arrays bidimensionais? O programador precisa de uma fonte de palavras (isto é, um dicionário computadorizado) que possa ser referenciado diretamente pelo programa. De que forma essas palavras devem ser armazenadas para facilitar as complexas manipulações requeridas pelo programa? O leitor realmente ambicioso vai querer gerar a parte das 'pistas' do jogo, em que as breves dicas para as palavras 'horizontais' e para as palavras 'verticais' são impressas para o gerador de quebra-cabeça. A mera impressão de uma versão da parte em branco do jogo não é um problema simples.