Técnicas de Programação

TP0601

Prof. Giovane Barcelos
giovane_barcelos@uniritter.edu.br

Plano de Ensino
Conteúdo programático

1. Introdução à programação em C 2. Desenvolvimento estruturado de programas em C 3. Controle de programa 4. Funções 5. Arrays 6. Ponteiros	**N1**
7. Caracteres e strings 8. Entrada/Saída formatada 9. Estruturas, uniões, manipulações de bits e enumerações 10. Processamento de arquivos 11. Estruturas de dados 12. O pré-processador 13. Outros tópicos sobre C	**N2**

Objetivos

- A construir programas de forma modular a partir de pequenas partes chamadas funções.
- As funções matemáticas comuns na biblioteca-padrão em C.
- A criar novas funções.
- Os mecanismos usados para passar informações entre funções.
- Como o mecanismo de chamada/retorno de função é aceito pela pilha de chamada de função e pelos registros de ativação.
- Técnicas de simulação a partir da geração de números aleatórios.
- Como escrever e usar funções que chamam a si mesmas.

Objetivos

- Sobre ponteiros e operadores de ponteiros.
- A usar ponteiros para passar argumentos a funções por referência.
- Sobre a relação entre ponteiros, arrays e strings.
- A usar os ponteiros em funções.
- A definir e usar arrays de strings.

Introdução

- **Neste capítulo, discutiremos um dos recursos mais poderosos da linguagem de programação em C, o ponteiro.**
- **Os ponteiros permitem que os programas simulem uma chamada por referência e criem e manipulem estruturas dinâmicas de dados, ou seja, estruturas de dados que podem crescer e encolher no tempo de execução, por exemplo, listas interligadas, filas, pilhas e árvores.**
- **O Capítulo 10 examinará o uso de ponteiros com estruturas.**
- **O Capítulo 12 introduzirá técnicas de gerenciamento dinâmico de memória e apresentará exemplos de criação e uso de estruturas dinâmicas de dados..**

Declarações e inicialização de variáveis-ponteiro

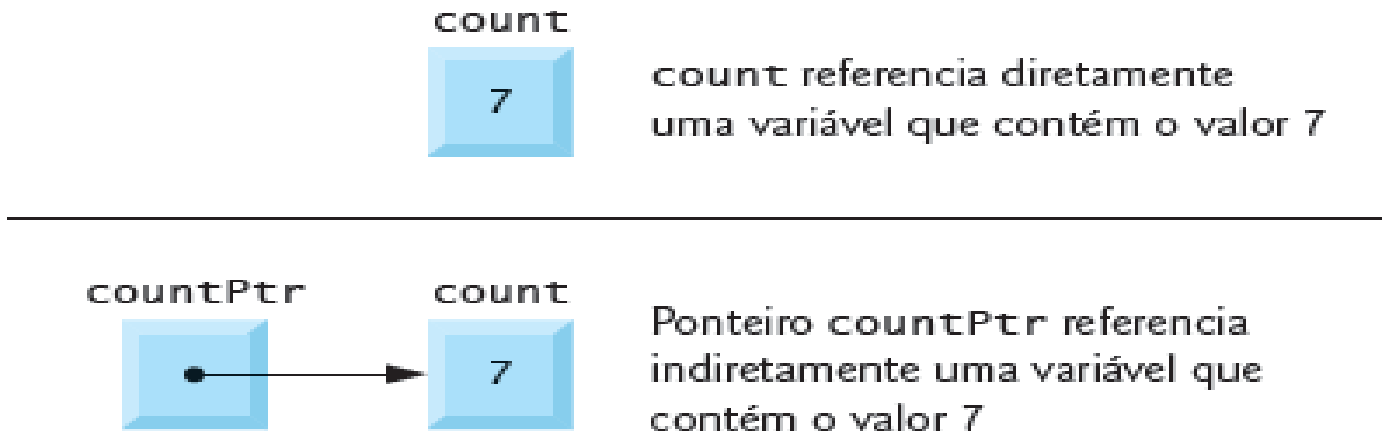


Figura 7.1 ■ Referências direta e indireta de uma variável.

Declarações e inicialização de variáveis-ponteiro

- **Os ponteiros são variáveis cujos valores são endereços de memória.**
- **Normalmente, uma variável claramente contém um valor específico.**
- **Um ponteiro, por outro lado, contém um endereço de uma variável que contém um valor específico.**
- **De certa forma, um nome de variável referencia um valor *diretamente*, enquanto um ponteiro referencia um valor *indiretamente* (Figura 7.1).**
- **A referência de um valor por meio de um ponteiro é chamada de indireção.**

Declarações e inicialização de variáveis-ponteiro

- Ponteiros, assim como todas as variáveis, precisam ser definidos antes de poderem ser usados.
- A definição
  - `int *countPtr, count;`

  especifica que a variável countPtr é do tipo int * (ou seja, um ponteiro para um inteiro), e que é lida como 'countPtr é um ponteiro para int' ou 'countPtr aponta para um objeto do tipo int'. Além disso, a variável count é definida para ser um int, e não um ponteiro para um int.
- O * pode ser aplicado somente a countPtr na definição.
- Quando * é usado dessa maneira em uma definição, ele indica que a variável que está sendo definida é um ponteiro.
- Os ponteiros podem ser definidos para apontar objetos de qualquer tipo.

Declarações e inicialização de variáveis-ponteiro

Erro comum de programação 7.1

*A notação asterisco (*), usada para declarar variáveis de ponteiro, não distribui para todos os nomes de variáveis em uma declaração. Cada ponteiro precisa ser declarado com o * prefixado ao nome; por exemplo, se você quiser declarar xPtr e yPtr como ponteiros int, use int *xPtr, *yPtr;.*

Erro comum de programação 7.2

Inclua as letras ptr nos nomes de variáveis de ponteiro para deixar claro que essas variáveis são ponteiros, e, portanto, precisam ser tratadas de modo apropriado.

Declarações e inicialização de variáveis-ponteiro

- Os ponteiros devem ser inicializados quando são definidos, ou em uma instrução de atribuição.
- Um ponteiro pode ser inicializado com NULL, 0 ou um endereço.
- Um ponteiro de valor NULL não aponta para nada.
- NULL é uma constante simbólica definida no cabeçalho <stddef.h> (e em vários outros cabeçalhos, como <stdio.h>).
- Inicializar um ponteiro em 0 é equivalente a inicializar um ponteiro com NULL, mas NULL é mais conveniente.
- Quando 0 é atribuído, ele é, em primeiro lugar, convertido em um ponteiro apropriado.
- O valor 0 é o único valor inteiro que pode ser atribuído diretamente a uma variável de ponteiro.

Declarações e inicialização de variáveis-ponteiro

Dica de prevenção de erro 7.1

Inicialize os ponteiros para evitar resultados inesperados.

Operadores de ponteiros

- O &, ou operador de endereço, é um operador unário que retorna o endereço de seu operando.
- Por exemplo, considerando as definições
  - ```
    int y = 5;
    int *yPtr;
    ```

  a instrução
  - ```
    yPtr = &y;
    ```

  atribui o endereço da variável y à variável de ponteiro yPtr.
- A variável yPtr, então, 'aponta para' y.
- A Figura 7.2 é uma representação esquemática da memória após essa atribuição ter sido executada.

Operadores de ponteiros

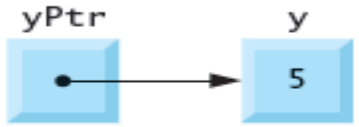


Figura 7.2 ■ Representação gráfica de um ponteiro apontando para uma variável inteira na memória.

Operadores de ponteiros

- A Figura 7.3 representa o ponteiro na memória, supondo que a variável inteira `y` esteja armazenada no local `600000`, e a variável de ponteiro `yPtr` esteja armazenada no local `500000`.
- O operando do operador de endereço precisa ser uma variável; o operador de endereço não pode ser aplicado a constantes, a expressões ou a variáveis declaradas com a classe de armazenamento `register`.

Operadores de ponteiros

Figura 7.3 ■ Representação de y e yPtr na memória.

Operadores de ponteiros

- O operador unário *, normalmente chamado operador de indireção ou de desreferenciação, retorna o valor do objeto apontado por seu operando (ou seja, um ponteiro).
- Por exemplo, a instrução.
  - printf("%d", *yPtr);

  imprime o valor da variável y, a saber, 5.
- Esse uso de * é chamado desreferenciação de um ponteiro.

Operadores de ponteiros

Erro comum de programação 7.3

Acessar um conteúdo com um ponteiro que não foi devidamente inicializado ou que não foi designado para apontar
um local específico na memória é um erro. Isso poderia causar um erro fatal no tempo de execução, ou poderia acidentalmente modificar dados e permitir que o programa fosse executado até o fim com resultados incorretos.

Operadores de ponteiros

```c
/* Fig. 7.4: fig07_04.c
   Usando os operadores & e *  */
#include <stdio.h>

int main( void )
{
   int a; /* a é um inteiro */
   int *aPtr; /* aPtr é um ponteiro para um inteiro */

   a = 7;
   aPtr = &a; /* aPtr definido para o endereço de a */

   printf( "O endereço de a é %p"
       "\nO valor de aPtr é %p", &a, aPtr );

   printf( "\n\nO valor de a é %d"
       "\nO valor de *aPtr é %d", a, *aPtr );

   printf( "\n\nMostrando que * e & são complementos um "
      "do outro\n&*aPtr = %p"
      "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */
```

```
O endereço de a é 0012FF7C
O valor de aPtr é 0012FF7C

O valor de a é 7
O valor de *aPtr é 7

Mostrando que * e & são complementos um do outro.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Figura 7.4 ■ Usando os operadores de ponteiros & e *.

Operadores de ponteiros

Operadores	Associatividade	Tipo
( ) [ ]	esquerda para direita	mais alta
+ - ++ -- ! * & (*tipo*)	direita para esquerda	unário
* / %	esquerda para direita	multiplicativo
+ -	esquerda para direita	aditivo
< <= > >=	esquerda para direita	relacional
== !=	esquerda para direita	igualdade
&&	esquerda para direita	AND lógico
||	esquerda para direita	OR lógico
?:	direita para esquerda	condicional
= += -= *= /= %=	direita para esquerda	atribuição
,	esquerda para direita	vírgula

Figura 7.5 ■ Precedência e associatividade de operadores.

Operadores de ponteiros

- A Figura 7.4 demonstra os operadores de ponteiro & e *.
- O especificador de conversão %p de printf mostra o local da memória como um inteiro hexadecimal na maioria das plataformas.
- (Veja o Apêndice C, Sistemas de Numeração, para obter mais informações sobre inteiros hexadecimais.) Observe que o endereço de a e o valor de aPtr são idênticos na saída, confirmando, assim, que o endereço de a é realmente atribuído à variável de ponteiro aPtr (linha 11).
- Os operadores & e * são complementos um do outro — quando ambos são aplicados consecutivamente a aPtr em qualquer ordem (linha 21), o mesmo resultado é impresso.
- A Figura 7.5 lista a precedência e a associatividade dos operadores introduzidos até agora

Passando argumentos para funções por referência

- Existem duas maneiras de passar argumentos a uma função, denominadas chamada por valor e chamada por referência.
- Todos os argumentos em C são passados por valor.
- Muitas funções exigem a capacidade de modificar uma ou mais variáveis na função chamadora ou passar um ponteiro para um objeto com grandequantidade de dados, para evitar o overhead de passar o objeto por valor (o que implicaria na sobrecarga de ter de fazer uma cópia do objeto inteiro).
- Para essas finalidades, C oferece recursos para uma simulação de chamada por referência.
- Em C, você usa ponteiros e operadores de indireção para simular uma chamada por referência.

Passando argumentos para funções por referência

- Ao chamar uma função com argumentos que devem ser modificados, os endereços dos argumentos são passados.
- Isso normalmente é feito ao se aplicar o operador (&) à variável (na função chamadora), cujo valor será modificado.
- Como vimos no Capítulo 6, os arrays não são passados usando-se o operador &, pois C passa automaticamente o local inicial para a memória do array (o nome de um array é equivalente a&arrayName[0]).
- Quando o endereço de uma variável é passado para uma função, o operador de indireção (*) pode ser usado na função para modificar o valor nesse local da memória da função chamadora.

Passando argumentos para funções por referência

```
1  /* Fig. 7.6: fig07_06.c
2     Cubo de uma variável usando chamada por valor */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* protótipo */
6
7  int main( void )
8  {
9     int number = 5; /* inicializa número */
10
11    printf( "O valor original do número é %d", number );
12
13    /* passa número por valor a cubeByValue */
14    number = cubeByValue( number );
15
```

Figura 7.6 ■ Cubo de uma variável usando chamada por valor. (Parte 1 de 2.)

Passando argumentos para funções por referência

```
16      printf( "\nO novo valor do número é %d\n", number );
17      return 0; /* indica conclusão bem-sucedida */
18   } /* fim do main */
19
20   /* calcula e retorna cubo do argumento inteiro */
21   int cubeByValue( int n )
22   {
23      return n * n * n; /* calcula cubo da variável local n e retorna resultado */
24   } /* fim da função cubeByValue */
```

```
O valor original do número é 5
O novo valor do número é 125
```

Figura 7.6 ■ Cubo de uma variável usando chamada por valor. (Parte 2 de 2.)

Passando argumentos para funções por referência

```
1  /* Fig. 7.7: fig07_07.c
2     Calcula o cubo de uma variável usando chamada por referência com argumento ponteiro */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /* protótipo */
7
8  int main( void )
9  {
10    int number = 5; /* inicializa número */
11
12    printf( "O valor original do número é %d", number );
13
14    /* passa endereço do número a cubeByReference */
15    cubeByReference( &number );
16
17    printf( "\nO novo valor do número é %d\n", number );
18    return 0; /* indica conclusão bem-sucedida */
19 } /* fim de main */
20
21 /* calcula cubo de *nPtr; modifica variável number em main */
22 void cubeByReference( int *nPtr )
23 {
24    *nPtr = *nPtr * *nPtr * *nPtr; /* cubo de *nPtr */
25 } /* fim da função cubeByReference */
```

```
O valor original do número é 5
O novo valor do número é 125
```

Figura 7.7 ■ Cubo de uma variável usando chamada por referência com um argumento ponteiro.

Passando argumentos para funções por referência

Etapa1: Antes de chamar cubeByValue:

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

   number = cubeByValue( number );
}
```

number: 5

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

n: indefinido

Etapa 2: Depois de cubeByValue ter recebido a chamada:

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

   number = cubeByValue( number );
}
```

number: 5

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

n: 5

Etapa 3: Depois de cubeByValue elevar ao cubo o parâmetro n e antes de cubeByValue retornar para main:

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

   number = cubeByValue( number );
}
```

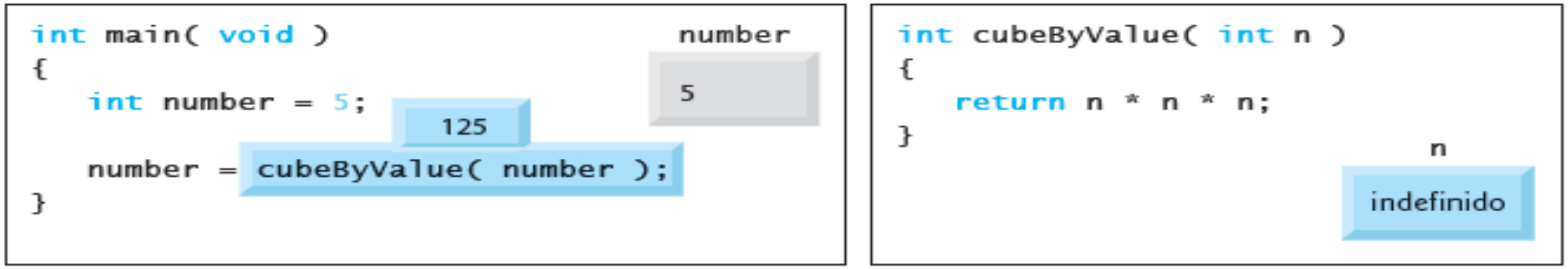
number: 5

```
int cubeByValue( int n )
{
   return 125
   return n * n * n;
}
```

n: 5

Passando argumentos para funções por referência

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



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

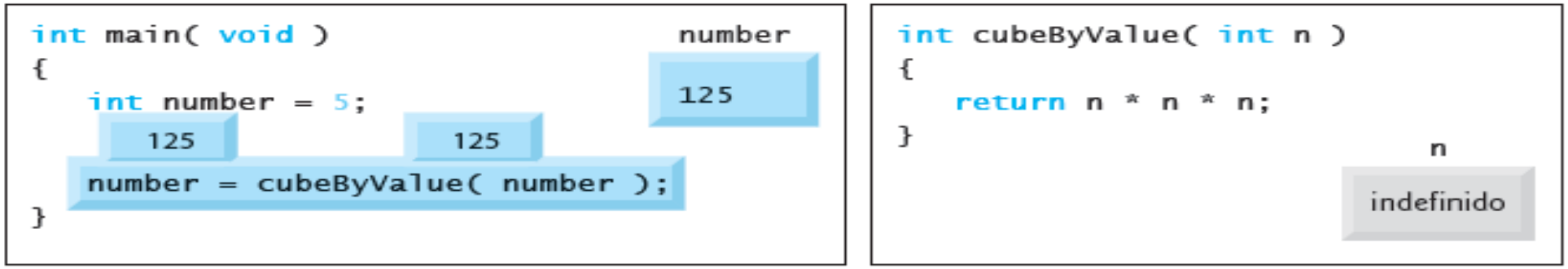


Figura 7.8 ■ Análise de uma típica chamada por valor.

Passando argumentos para funções por referência

Etapa 1: Antes de main chamar cubeByReference:

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

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr: indefinido

Etapa 2: Depois de cubeByReference receber a chamada e antes de *nPtr ser elevado ao cubo.

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

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

chamada estabelece esse ponteiro → nPtr

Etapa 3: Depois de *nPtr ser elevado ao cubo e antes de o controle do programa retornar a main:

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

   cubeByReference( &number );
}
```

number: 125

```
void cubeByReference( int *nPtr )
{
   125
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

função chamada modifica variável chamadora → nPtr

Figura 7.9 ■ Análise de uma típica chamada por referência com um argumento de ponteiro.

Passando argumentos para funções por referência

- Os programas na Figura 7.6 e na Figura 7.7 apresentam duas versões de uma função que calcula o cubo de um inteiro — cubeByValue and cubeByReference.
- A Figura 7.6 passa a variável number à função cubeByValue usando a chamada por valor (linha 14).
- A função cubeByValue calcula o cubo de seu argumento e passa o novo valor de volta a main, usando um comando return.
- O novo valor é atribuído a number em main (linha 14).

Passando argumentos para funções por referência

- A Figura 7.7 passa a variável number usando a chamada por referência (linha 15) — o endereço de number passado — à função cubeByReference.
- A função cubeByReference usa como parâmetro um ponteiro para um int chamado nPtr (linha 22).
- A função desreferencia o ponteiro e calcula o cubo do valor para o qual nPtr aponta (linha 24), e depois atribui o resultado a *nPtr (que na realidade é number em main), mudando assim o valor de number em main.
- A Figura 7.8 e a Figura 7.9 analisam graficamente os programas na Figura 7.6 e na Figura 7.7, respectivamente.

Passando argumentos para funções por referência

- Uma função que recebe um endereço como argumento precisa definir um parâmetro de ponteiro para receber esse endereço.
- Por exemplo, na Figura 7.7, o cabeçalho para a função cubeByReference (linha 22) é:
  - `void cubeByReference( int *nPtr )`
- O cabeçalho especifica que cubeByReference recebe o endereço de uma variável inteira como argumento, armazena o endereço localmente em nPtr e não retorna um valor.
- O protótipo de função para cubeByReference contém int * entre parênteses.
- Os nomes incluídos para fins de documentação são ignorados pelo compilador C.

Passando argumentos para funções por referência

- No cabeçalho da função e no protótipo para uma função que espera por um array de subscrito único como argumento, a notação de ponteiro da lista de parâmetros da função cubeByReference pode ser usada.
- O compilador não diferencia uma função que recebe
- um ponteiro de uma função que recebe um array de subscrito único.
- Isso, naturalmente, significa que a função precisa 'saber' quando vai receber um array ou simplesmente uma única variável para a qual deve executar a chamada por referência.
- Quando o compilador encontra um parâmetro de função para um array de subscrito único na formaint b[], o compilador converte o parâmetro para a notação de ponteiro int *b.
- As duas formas são intercambiáveis..

Passando argumentos para funções por referência

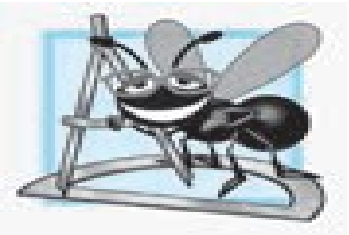
Dica de prevenção de erro 7.2

Use a chamada por valor para passar argumentos a uma função, a menos que a função chamadora exija explicitamente
que a função chamada modifique o valor da variável do argumento no ambiente da função chamadora. Isso impede a modificação acidental dos argumentos da função chamadora, e também é outro exemplo do princípio do
menor privilégio.

Usando o qualificador const com ponteiros

- O qualificador const permite que você informe ao compilador que o valor de determinada variável deve ser modificado.

Usando o qualificador const com ponteiros



Observação sobre engenharia de software 7.1

O qualificador const pode ser usado para impor o princípio do menor privilégio. A utilização do princípio do menor privilégio para projetar o software corretamente reduz o tempo de depuração e os efeitos colaterais impróprios,
o que torna o programa mais fácil de modificar e de ser mantido.

Dica de portabilidade 7.1

Embora const seja bem-definida na linguagem C padrão, ela não é imposta por alguns compiladores.

Usando o qualificador const com ponteiros

- Com o passar dos anos, uma grande base de código legado foi escrita nas primeiras versões de C, que não usavam const porque ele não estava disponível.
- Por esse motivo, existem oportunidades significativas para a melhoria na reengenharia do código C antigo.
- Existem seis possibilidades de uso (ou de não uso) para const com parâmetros de função — duas com passagem de parâmetros por chamada por valor e quatro com passagem de parâmetros por chamada por referência.
- Como escolher uma dentre as seis possibilidades? Deixe que o princípio do menor privilégio seja seu guia.
- Sempre conceda a uma função acesso suficiente aos dados em seus parâmetros para que ela realize a tarefa especificada, mas não mais que isso.

Usando o qualificador const com ponteiros

- No último encontro explicamos que todas as chamadas em C são chamadas por valor — uma cópia do argumento na chamada de função é feita e passada à função.
- Se a cópia for modificada na função, o valor original na função chamadora não muda.
- Em muitos casos, um valor passado a uma função é modificado de modo que a função possa realizar sua tarefa.
- Porém, em alguns casos, o valor não deve ser alterado na função chamada, embora ela manipule apenas uma cópia do valor original.
- Considere uma função que recupere um array de subscrito único e seu tamanho como argumentos, e imprima na tela o array.

Usando o qualificador const com ponteiros

- **Essa função deverá percorrer o array e enviar cada um de seus elementos individualmente.**
- **O tamanho do array é usado no corpo da função para determinar o subscrito alto do array, de modo que o loop possa terminar quando a impressão for concluída.**
- **Nem o tamanho do array nem o conteúdo deverão mudar no corpo da função.**

Usando o qualificador const com ponteiros

Dica de prevenção de erro 7.3

Se uma variável não mudar (ou não tiver de mudar) no corpo de uma função à qual ela for passada, ela deverá ser declarada const para garantir que não seja modificada acidentalmente.

Usando o qualificador const com ponteiros

- **Se uma tentativa de modificação de um valor declarado como const, feita, o compilador apanhará isso e emitirá uma advertência ou um erro, a depender do compilador em questão.**

Usando o qualificador const com ponteiros

Observação sobre engenharia de software 7.2
Em uma função chamadora, apenas um valor pode ser alterado na utilização da chamada por valor. Esse valor deve ser atribuído a partir do valor de retorno da função para uma variável na função chamadora. Para modificar diversas variáveis de uma função chamadora em uma função chamada, use a chamada por referência.

Dica de prevenção de erro 7.4
Antes de usar uma função, verifique seu protótipo de função para determinar se a função é capaz de modificar os valores passados a ela.

Usando o qualificador const com ponteiros

Erro comum de programação 7.4

Não estar ciente de que uma função está à espera de ponteiros como argumentos para a chamada por referência e, então, passar argumentos com chamada por valor. Alguns compiladores capturam os valores supondo que eles sejam ponteiros, e os desreferenciam como tais. No tempo de execução, normalmente são geradas violações de acesso à memória ou falhas de segmentação. Outros compiladores detectam a divergência em tipos entre argumentos e parâmetros, e geram mensagens de erro.

Usando o qualificador const com ponteiros

- **Existem quatro maneiras de passar um ponteiro para uma função: um ponteiro não constante para dados não constantes, um ponteiro constante para dados não constantes, um ponteiro não constante para dados constantes e um ponteiro constante para dados constantes.**
- **Cada uma dessas quatro combinações oferece diferentes privilégios de acesso, que serão discutidos nos próximos exemplos.**

Usando o qualificador const com ponteiros

- O nível mais alto de acesso a dados é concedido por um ponteiro não constante para dados não constantes.
- Nesse caso, os dados podem ser modificados por meio de um ponteiro desreferenciado, e o ponteiro pode ser modificado para apontar para outros itens de dados.
- Uma declaração para um ponteiro não constante para dados não constantes não inclui const.
- Esse ponteiro poderia ser usado para receber uma string como argumento para uma função que usa aritmética de ponteiro para processar (e, possivelmente, modificar) cada caractere da string..

Usando o qualificador const com ponteiros

```
1  /* Fig. 7.10: fig07_10.c
2     Convertendo uma string em maiúsculas usando um
3     ponteiro não constante para dados não constantes */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUppercase( char *sPtr ); /* protótipo */
9
10 int main( void )
11 {
12    char string[] = "caracteres e R$32,98"; /* inicializa array de char */
13
14    printf( "A string antes da conversão é: %s", string );
15    convertToUppercase( string );
16    printf( "\nA string após a conversão é: %s\n", string );
17    return 0; /* indica conclusão bem-sucedida */
18 } /* fim do main */
19
20 /* converte string em letras maiúsculas */
21 void convertToUppercase( char *sPtr )
22 {
```

Usando o qualificador const com ponteiros

```
23      while ( *sPtr != '\0' ) { /* caractere atual não é '\0' */
24
25         if ( islower( *sPtr ) ) { /* se o caractere é minúsculo, */
26            *sPtr = toupper( *sPtr ); /* converte em maiúsculas */
27         } /* fim do if */
28
29         ++sPtr; /* desloca sPtr para o caracter seguinte */
30      } /* fim do while */
31   } /* fim da função convertToUppercase */
```

```
A string antes da conversão é: caracteres e R$32,98
A string após a conversão é: CARACTERES E R$32,98
```

Figura 7.10 ■ Convertendo uma string em maiúsculas usando um ponteiro não constante para dados não constantes.

Usando o qualificador const com ponteiros

- A função convertToUppercase da Figura 7.10 declara seu parâmetro, um ponteiro não constante para dados não constantes, chamado sPtr (char *sPtr), na linha 21.
- A função processa o array string (apontado por sPtr) um caractere por vez, usando a aritmética de ponteiro.
- A função da biblioteca-padrão de C islower (chamada na linha 25) testa o conteúdo de caractere do endereço apontado por sPtr.
- Se um caractere estiver no intervalo de a até z, islower retorna verdadeiro e a função da biblioteca-padrão de C toupper (linha 26) é chamada para converter o caractere na sua letra maiúscula correspondente; caso contrário, islower retorna falso e o próximo caractere na string é processado.
- A linha 29 move o ponteiro até o próximo caractere na string.

Usando o qualificador const com ponteiros

- **Um ponteiro não constante para dados constantes pode ser modificado para apontar qualquer item de dados do tipo apropriado, mas os dados aos quais ele aponta não podem ser modificados.**
- **Esse ponteiro poderia ser usado para receber um argumento de array em uma função que processaria todos os elementos sem modificar os dados.**

Usando o qualificador const com ponteiros

```
1  /* Fig. 7.11: fig07_11.c
2     Imprimindo uma string um caractere por vez usando
3     um ponteiro não constante para dados constantes */
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main( void )
10 {
11    /* inicializa array de char */
12    char string[] = "imprime caracteres de uma string";
13
14    printf( "A string é:\n" );
15    printCharacters( string );
16    printf( "\n" );
17    return 0; /* indica conclusão bem-sucedida */
18 } /* fim do main */
19
20 /* sPtr não pode modificar o caractere ao qual aponta,
21    ou seja, sPtr é um ponteiro "somente de leitura" */
22 void printCharacters( const char *sPtr )
23 {
24    /* loop pela string inteira */
25    for ( ; *sPtr != '\0'; sPtr++ ) { /* sem inicialização */
26       printf( "%c", *sPtr );
27    } /* fim do for */
28 } /* fim da função printCharacters */
```

```
A string é:
imprime caracteres de uma string
```

Figura 7.11 ■ Imprimindo uma string, um caractere por vez, usando um ponteiro não constante para dados constantes.

Usando o qualificador const com ponteiros

- Por exemplo, a função printCharacters (Figura 7.11) declara o parâmetro sPtr como do tipo const char * (linha 22).
- A declaração é lida da direita para a esquerda como ‘sPtr é um ponteiro para uma constante de caractere’. A função usa uma estrutura for para exibir cada caractere na string até que o caractere nulo seja encontrado.
- Após a impressão de cada caractere, o ponteiro sPtr é incrementado para apontar o próximo caractere na string.

Usando o qualificador const com ponteiros

```
1  /* Fig. 7.12: fig07_12.c
2     Tentando modificar dados por meio de um
3     ponteiro não constante para dados constantes. */
4  #include <stdio.h>
5  void f( const int *xPtr ); /* protótipo */
6
7
8  int main( void )
9  {
10    int y; /* define y */
11
12    f( &y ); /* f tenta modificação ilegal */
13    return 0; /* indica conclusão bem-sucedida */
```

Figura 7.12 ■ Tentativa de modificação de dados por um ponteiro não constante para dados constantes. (Parte 1 de 2.)

Usando o qualificador const com ponteiros

```
14  } /* fim do main */
15
16  /* xPtr não pode ser usado para modificar o valor
17     da variável à qual ele aponta */
18  void f( const int *xPtr )
19  {
20     *xPtr = 100; /* erro: não pode modificar um objeto const */
21  } /* fim da função f */
```

```
Compiling...
FIG07_12.c
c:\examples\ch07\fig07_12.c(22) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_12.exe - 1 error(s), 0 warning(s)
```

Figura 7.12 ■ Tentativa de modificação de dados por um ponteiro não constante para dados constantes. (Parte 2 de 2.)

Usando o qualificador const com ponteiros

- A Figura 7.12 ilustra a tentativa de compilação de uma função que recebe um ponteiro não constante (xPtr) para dados constantes.
- Essa função tenta modificar os dados apontados por xPtr na linha 20 — o que resulta em um erro de compilação.

Usando o qualificador const com ponteiros

- Como sabemos, os arrays são tipos de dados agregados que armazenam itens de dados relacionados e do mesmo tipo sob um único nome.
- No Capítulo 10, discutiremos outra forma de tipo de dado agregado chamado estrutura (às vezes, em outras linguagens, é chamado de registro).
- Uma estrutura é capaz de armazenar dados relacionados de diferentes tipos sob um único nome (por exemplo, informações sobre cada funcionário de uma empresa).
- Quando uma função é chamada com um array como argumento, o array é automaticamente passado à função por referência.
- Porém, as estruturas sempre são passadas por valor — uma cópia da estrutura inteira é passada.

Usando o qualificador const com ponteiros

- Isso exige a sobrecarga do tempo de execução da realização de uma cópia de cada dado na estrutura, e sua armazenagem na pilha de chamada da função de comunicação.
- Quando os dados da estrutura precisam ser passados a uma função, podemos usar ponteiros para dados constantes para conseguir a execução da chamada por referência e a proteção da chamada por valor.
- Quando um ponteiro de uma estrutura é passado, apenas uma cópia do endereço em que a estrutura está armazenada deve ser feita.
- Em uma máquina com endereços de 4 bytes, é feita uma cópia dos 4 bytes de memória em vez de uma cópia de, possivelmente, centenas ou milhares de bytes da estrutura.

Usando o qualificador const com ponteiros

Dica de desempenho 7.1

Objetos grandes como estruturas devem ser passados por meio de ponteiros para dados constantes, a fim de que se
obtenham os benefícios de desempenho da chamada por referência e de segurança da chamada por valor.

Usando o qualificador const com ponteiros

- **Esse uso dos ponteiros para dados constantes é um exemplo de dilema de tempo/espaço.**
- **Se a memória for pequena e a eficiência da execução for um problema, use ponteiros.**
- **Se a memória for abundante e a eficiência não for uma questão importante, passe dados por valor, para impor o princípio do menor privilégio.**
- **Lembre-se de que alguns sistemas não impõem const muito bem, de modo que a chamada por valor ainda é a melhor maneira de impedir que os dados sejam modificados.**

Usando o qualificador const com ponteiros

- Um ponteiro constante para dados não constantes sempre aponta para o mesmo local da memória, e os dados nesse local podem ser modificados por meio do ponteiro.
- Este é o padrão para um nome de array.
- O nome do array é um ponteiro constante para o início do array.
- Todos os dados do array podem ser acessados e alterados pelo uso de seu nome e subscrito.
- Um ponteiro constante para dados não constantes pode ser usado para receber um array como um argumento para uma função que acessa elementos do array, usando apenas a notação de subscrito deste.

Usando o qualificador const com ponteiros

```
1  /* Fig. 7.13: fig07_13.c
2     Tentando modificar um ponteiro constante para dados não constantes */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int x; /* define x */
8     int y; /* define y */
9
10    /* ptr é um ponteiro constante para um inteiro que pode ser modificado por
11       meio de ptr, mas ptr sempre aponta para o mesmo local da memória */
12    int * const ptr = &x;
13
14    *ptr = 7; /* permitido: *ptr não é const */
15    ptr = &y; /* erro: ptr é const; não pode atribuir novo endereço */
16    return 0; /* indica conclusão bem-sucedida */
17 } /* fim do main */
```

```
Compiling...
FIG07_13.c
c:\examples\ch07\FIG07_13.c(15) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
```

Figura 7.13 ■ Tentando modificar um ponteiro constante para dados não constantes.

Usando o qualificador const com ponteiros

- Ponteiros que sejam declarados const precisam ser inicializados ao serem definidos (se o ponteiro for um parâmetro de função, ele será inicializado com um ponteiro que será passado para a função).
- Na Figura 7.13, vemos uma tentativa de modificação de um ponteiro constante.
- O ponteiro ptr é definido na linha 12 para ser do tipo int * const.
- A definição é lida da direita para a esquerda como: `ptr é um ponteiro constante para um inteiro'. O ponteiro é inicializado (linha 12) com o endereço da variável inteira x.
- O programa tenta atribuir o endereço de y a ptr (linha 15), mas o compilador gera uma mensagem de erro.

Usando o qualificador const com ponteiros

- O privilégio mínimo de acesso é concedido por um ponteiro constante para dados constantes.
- Esse ponteiro sempre aponta o mesmo local da memória, e os dados nesse local da memória não podem ser modificados.
- Um array deve ser passado dessa forma a uma função, que somente vai examiná-lo usando a notação de subscrito do array, sem modificá-lo.

Usando o qualificador const com ponteiros

```
1  /* Fig. 7.14: fig07_14.c
2     Tentando modificar um ponteiro constante para dados constantes. */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int x = 5; /* inicializa x */
8     int y; /* define y */
9
10    /* ptr é um ponteiro constante para um inteiro constante. ptr sempre
11       aponta o mesmo local; o inteiro nesse local
12       não pode ser modificado */
13    const int *const ptr = &x;
14
15    printf( "%d\n", *ptr );
16    *ptr = 7; /* erro: *ptr é const; não pode atribuir novo valor */
17    ptr = &y; /* erro: ptr é const; não pode atribuir novo endereço */
18    return 0; /* indica conclusão bem-sucedida */
19 } /* fim do main */
```

Figura 7.14 ■ Tentando modificar um ponteiro constante para dados constantes. (Parte 1 de 2.)

Usando o qualificador const com ponteiros

```
Compiling...
FIG07_14.c
c:\examples\ch07\FIG07_14.c(17) : error C2166: l-value specifies const object
c:\examples\ch07\FIG07_14.c(18) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_12.exe - 2 error(s), 0 warning(s)
```

Figura 7.14 ■ Tentando modificar um ponteiro constante para dados constantes. (Parte 2 de 2.)

Usando o qualificador const com ponteiros

- A Figura 7.14 define a variável de ponteiro ptr (linha 13) para ser do tipo const int *const, que é lido da direita para a esquerda como: ‘ptr é um ponteiro constante para um inteiro constante’.
- A figura mostra as mensagens de erro geradas quando é feita uma tentativa de modificar os dados aos quais ptr aponta (linha 16), e quando é feita uma tentativa de modificar o endereço armazenado na variável do ponteiro (linha 17).

Bubble sort usando chamada por referência

```
1  /* Fig. 7.15: fig07_15.c
2     Esse programa coloca valores em um array, ordena os valores em
3     ordem crescente e imprime o array resultante. */
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort( int * const array, const int size ); /* protótipo */
8
9  int main( void )
10 {
11    /* inicializa array a */
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    int i; /* contador */
```

Figura 7.15 ■ Bubble sort com chamada por referência. (Parte 1 de 2.)

Bubble sort usando chamada por referência

```
15
16      printf( "Itens de dados na ordem original\n" );
17
18      /* loop pelo array a */
19      for ( i = 0; i < SIZE; i++ ) {
20         printf( "%4d", a[ i ] );
21      } /* fim do for */
22
23      bubbleSort( a, SIZE ); /* ordena o array */
24
25      printf( "\nItens de dados em ordem crescente\n" );
26
27      /* loop pelo array a */
28      for ( i = 0; i < SIZE; i++ ) {
29         printf( "%4d", a[ i ] );
30      } /* fim do for */
31
32      printf( "\n" );
33      return 0; /* indica conclusão bem-sucedida */
34   } /* fim do main */
35
36   /* ordena um array de inteiros usando algoritmo bubble sort */
37   void bubbleSort( int * const array, const int size )
38   {
39      void swap( int *element1Ptr, int *element2Ptr ); /* protótipo */
40      int pass; /* contador de passadas */
41      int j; /* contador de comparação */
42
43      /* loop para controlar passadas */
44      for ( pass = 0; pass < size - 1; pass++ ) {
45
```

Bubble sort usando chamada por referência

```
46          /* loop para controlar comparações durante cada passada */
47          for ( j = 0; j < size - 1; j++ ) {
48
49             /* troca elementos adjacentes se estiverem fora de ordem */
50             if ( array[ j ] > array[ j + 1 ] ) {
51                swap( &array[ j ], &array[ j + 1 ] );
52             } /* fim do if */
53          } /* fim do for interno */
54       } /* fim do for externo */
55    } /* fim da função bubbleSort */
56
57    /* troca valores nos locais da memória apontados por element1Ptr
58       e element2Ptr */
59    void swap( int *element1Ptr, int *element2Ptr )
60    {
61       int hold = *element1Ptr;
62       *element1Ptr = *element2Ptr;
63       *element2Ptr = hold;
64    } /* fim da função swap */
```

```
Itens de dados na ordem original
   2  6  4  8  10  12  89  68  45  37
Itens de dados em ordem crescente
   2  4  6  8  10  12  37  45  68  89
```

Figura 7.15 ■ Bubble sort com chamada por referência. (Parte 2 de 2.)

Bubble sort usando chamada por referência

- Vamos melhorar o programa de bubble sort mostrado na Figura 6.15 para que possamos usar duas funções —bubbleSort e swap.
- A função bubbleSort ordena o array.
- Ela chama a função swap (linha 51) para que ela troque os elementos do array array[j] e array[j + 1] (ver Figura 7.15).
- Lembre-se de que C impõe a ocultação de informações entre funções, de modo que swap não tem acesso a elementos individuais de array em bubbleSort.
- Como bubbleSort *deseja* que swap tenha acesso aos elementos do array para que eles sejam trocados, bubbleSort passa cada um desses elementos chamados por referência a swap — o endereço de cada elemento do array é passado explicitamente.

Bubble sort usando chamada por referência

- Embora arrays inteiros sejam automaticamente passados por referência, elementos de array individuais são escalares e normalmente passados por valor.
- Portanto, bubbleSort usa o operador de endereço (&) em cada um dos elementos do array na chamada swap (linha 51) para efetuar a chamada por referência da seguinte forma
  - `swap( &array[ j ], &array[ j + 1 ] );`
- A função swap recebe &array[j] na variável de ponteiro element1Ptr (linha 59).

Bubble sort usando chamada por referência

- Embora swap não tenha permissão para saber o nome array[j] — devido à ocultação de informações —, swap pode usar *element1Ptr como um sinônimo para array[j] — quando swap referencia *element1Ptr, ela está, na verdade, referenciando array[j] em bubbleSort.
- De modo semelhante, quando swap referencia *element2Ptr, ele está, na verdade, referenciando array[j + 1] em bubbleSort.
- Embora swap não possa dizer
  - hold = array[j];
    array[j] = array[j + 1];
    array[j + 1] = hold;

  o mesmo efeito é alcançado pelas linhas de 61 a 63

Bubble sort usando chamada por referência

- Vários recursos da função bubbleSort devem ser observados.
- O cabeçalho da função (linha 37) declara array como int * const array em vez de int array[] para indicar que bubbleSort recebe um array de subscrito único como argumento (novamente, essas notações são intercambiáveis).
- O parâmetro size é declarado 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 mudar o valor em main, bubbleSort não precisa alterar size para realizar essa tarefa.

Bubble sort usando chamada por referência

- O tamanho do array permanecerá fixo durante a execução da função bubbleSort.
- Portanto, size é declarado const para garantir que não será modificado.
- Se o tamanho do array for modificado durante o processo de ordenação, é possível que o algoritmo de ordenação não funcione corretamente.
- O protótipo para a função swap (linha 39) está incluído no corpo da função bubbleSort, pois bubbleSort é a única função que chama swap.

Bubble sort usando chamada por referência

- **Colocar o protótipo em bubbleSort restringe chamadas apropriadas de swap àquelas feitas a partir de bubbleSort.**
- **Outras funções que tentarem chamar swap não terão acesso a um protótipo de função apropriado, de modo que o compilador gerará um automaticamente.**
- **Isso normalmente resulta em um protótipo que não combina com o cabeçalho de função (e gera uma advertência ou erro de compilação), pois o compilador assume int para o tipo de retorno e para os tipos de parâmetro.**

Bubble sort usando chamada por referência

Observação sobre engenharia de software 7.3

Colocar protótipos de função nas definições de outras funções impõe o princípio do menor privilégio restringindo chamadas de função apropriadas às funções em que os protótipos aparecem.

Bubble sort usando chamada por referência

- A função bubbleSort recebe o tamanho do array como um parâmetro (linha 37).
- A função precisa saber o tamanho do array para ordená-lo.
- Quando um array é passado para uma função, o endereço de memória do primeiro elemento do array é recebido pela função.
- O endereço, naturalmente, não transmite o número de elementos do array.
- Portanto, você precisa passar o tamanho do array para a função.
- [*Nota*: outra prática comum é passar um ponteiro para o início do array e um ponteiro para o local logo após o final do array.

Bubble sort usando chamada por referência

- A diferença dos dois ponteiros é o comprimento do array, e o código resultante é mais simples.]
- No programa, o tamanho do array é passado explicitamente para a função bubbleSort.
- Essa técnica implica dois benefícios principais: a reutilização do software e a engenharia de software apropriada.
- Ao definir a função para receber o tamanho do array como um argumento, permitimos que ela seja usada por qualquer programa que ordene arrays de inteiros de subscrito único de qualquer tamanho.

Bubble sort usando chamada por referência

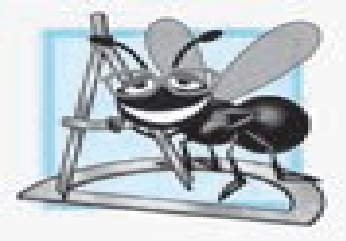
Observação sobre engenharia de software 7.4

Ao passar um array para uma função, passe também o tamanho do array. Isso ajuda a tornar a função reutilizável em muitos programas.

Bubble sort usando chamada por referência

- Poderíamos ter armazenado o tamanho do array em uma variável global acessível ao programa inteiro.
- Isso seria mais eficiente, pois não é feita uma cópia de seu tamanho para passar à função.
- Porém, outros programas que exigem uma capacidade de ordenação de array de inteiros podem não ter a mesma variável global, de modo que a função não pode ser usada nesses programas.

Bubble sort usando chamada por referência



Observação sobre engenharia de software 7.5
As variáveis globais normalmente violam o princípio do menor privilégio, e podem levar a uma engenharia de software ineficiente. As variáveis globais devem ser usadas somente para representar recursos verdadeiramente compartilhados, como a hora do dia.

Bubble sort usando chamada por referência

- O tamanho do array poderia ter sido programado diretamente na função.
- Isso restringiria o uso da função a um array de um tamanho específico, e reduziria significativamente sua reutilização.
- Somente programas que processam arrays de inteiros de subscrito único do tamanho específico codificado na função podem usar a função.

Operador sizeof

```c
/* Fig. 7.16: fig07_16.c
   A aplicação de sizeof a um nome de array retorna
   o número de bytes no array. */
#include <stdio.h>

size_t getSize( float *ptr ); /* protótipo */

int main( void )
{
   float array[ 20 ]; /* cria array */

   printf( "O número de bytes no array é %d"
           "\nO número de bytes retornados por getSize é %d\n",
           sizeof( array ), getSize( array ) );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */

/* retorna tamanho de ptr */
size_t getSize( float *ptr )
{
   return sizeof( ptr );
} /* fim da função getSize */
```

```
O número de bytes no array é 80
O número de bytes retornados por getSize é 4
```

Figura 7.16 ■ A aplicação do `sizeof` a um nome de array retorna o número de bytes no array.

Operador sizeof

- C oferece o operador unário especial sizeof para que o tamanho de um array (ou qualquer outro tipo de dado) seja determinado em bytes durante a compilação do programa.
- Quando aplicado ao nome de um array, como vemos na Figura 7.16 (linha 14), o operador sizeof retorna o número total de bytes no array como um inteiro.
- As variáveis do tipo float normalmente são armazenadas em 4 bytes de memória, e array é definido para ter 20 elementos.
- Portanto, existe um total de 80 bytes no array.

Operador sizeof

Dica de desempenho 7.2

O sizeof é um operador do tempo de compilação, de modo que não inclui nenhum overhead no tempo de
execução.

Operador sizeof

- O número de elementos em um array também pode ser determinado a partir do sizeof.
- Por exemplo, considere a seguinte definição de array:
  - `double real[ 22 ];`
- Variáveis do tipo double normalmente são armazenadas em 8 bytes de memória.
- Assim, o array real contém um total de 176 bytes.
- Para determinar o número de elementos no array, pode-se usar a expressão a seguir:
  - `sizeof( real ) / sizeof( real[ 0 ] )`

Operador sizeof

- A expressão determina o número de bytes no array real e divide esse valor pelo número de bytes usados na memória para armazenar o primeiro elemento do array real (um valor double).
- A função getSize retorna o tipo size_t.
- O tipo size_t é definido pelo padrão C como um tipo inteiro (unsigned ou unsigned long) do valor retornado pelo operador sizeof.
- O tipo size_t é definido no cabeçalho <stddef.h> (que está contido em vários cabeçalhos, como <stdio.h>).
- [*Nota*: se você tentar compilar a Figura 7.16 e receber erros, basta incluir *<stddef.h>* em seu programa.

Operador sizeof

```c
/* Fig. 7.17: fig07_17.c
   Demonstrando o operador sizeof */
#include <stdio.h>

int main( void )
{
   char c;
   short s;
   int i;
   long l;
   float f;
   double d;
   long double ld;
   int array[ 20 ]; /* cria array de 20 elementos int */
   int *ptr = array; /* cria ponteiro para array */

   printf( "    sizeof c = %d\tsizeof(char)  = %d"
           "\n     sizeof s = %d\tsizeof(short) = %d"
           "\n     sizeof i = %d\tsizeof(int) = %d"
           "\n     sizeof l = %d\tsizeof(long) = %d"
           "\n     sizeof f = %d\tsizeof(float) = %d"
           "\n     sizeof d = %d\tsizeof(double) = %d"
           "\n     sizeof ld = %d\tsizeof(long double) = %d"
           "\n sizeof array = %d"
           "\n   sizeof ptr = %d\n",
           sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
           sizeof( int ), sizeof l, sizeof( long ), sizeof f,
           sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
           sizeof( long double ), sizeof array, sizeof ptr );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */
```

Operador sizeof

```
      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 7.17 ■ Usando o operador sizeof para determinar os tamanhos dos tipos de dados-padrão.

Operador sizeof

- A Figura 7.17 calcula o número de bytes usados para armazenar cada um dos tipos de dados-padrão.
- Os resultados podem variar, a depender do computador.

Operador sizeof

Dica de portabilidade 7.2

O número de bytes utilizados no armazenamento de determinado tipo de dado pode variar entre os sistemas. Ao
escrever programas que dependem do tamanho dos tipos de dados, e que serão executados em vários sistemas de
computador, use sizeof para determinar o número de bytes usados para armazenar os tipos de dados.

Operador sizeof

- O operador sizeof pode ser aplicado a qualquer nome de variável, tipo ou valor (inclusive o valor de uma expressão).
- Quando aplicado ao nome de uma variável (que não é um nome de array) ou de uma constante, o número de bytes usados para armazenar o tipo específico da variável ou constante é retornado.
- Os parênteses usados com sizeof são necessários se o nome do tipo com duas palavras for fornecido como seu operando (comolong double ou unsigned short).
- Omitir os parênteses, nesse caso, resultará em um erro de sintaxe.
- Os parênteses não serão necessários se o nome da variável ou o nome de um tipo contendo uma única palavra for fornecido como seu operando, mas eles ainda poderão ser incluídos sem causar erro.

Expressões com ponteiros e aritmética de ponteiros

- Ponteiros são operandos válidos em expressões aritméticas, expressões de atribuição e expressões de comparação.
- Porém, nem todos os operadores normalmente usados nessas expressões são válidos em conjunto com variáveis de ponteiro.
- Esta seção descreverá os operadores que podem ter ponteiros como operandos, e como esses operadores são usados.
- Um conjunto limitado de operações aritméticas pode ser realizado com ponteiros.
- Um ponteiro pode ser incrementado (++) ou decrementado (--), um inteiro pode ser somado a um ponteiro (+ ou +=), um inteiro pode ser subtraído de um ponteiro (- ou -=) e um ponteiro pode ser subtraído de outro.

Expressões com ponteiros e aritmética de ponteiros

- Suponha que o array int v[5] tenha sido definido, e que seu primeiro elemento esteja no local 3000 na memória.
- Suponha que o ponteiro vPtr tenha sido inicializado para apontar v[0] — ou seja, o valor de vPtr é 3000.
- A Figura 7.18 ilustra essa situação no caso de uma máquina com inteiros de 4 bytes.
- A variável vPtr pode ser inicializada para apontar o array v com uma dessas instruções:

Expressões com ponteiros e aritmética de ponteiros

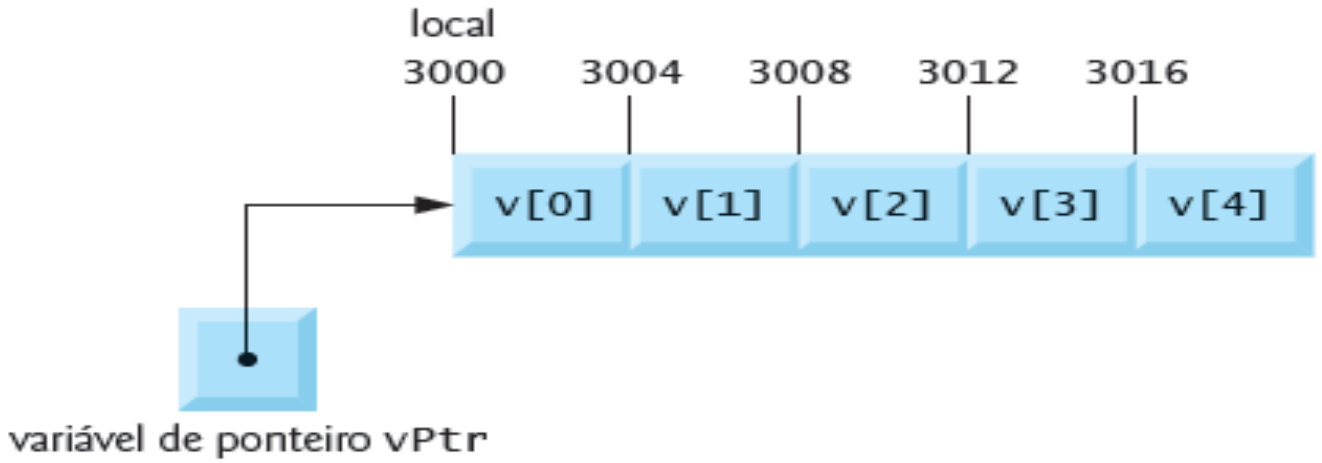


Figura 7.18 ■ Array v e uma variável de ponteiro vPtr que aponta para v.

Expressões com ponteiros e aritmética de ponteiros

Dica de portabilidade 7.3

A maioria dos computadores de hoje tem inteiros de 2 bytes ou 4 bytes. Algumas das máquinas mais novas utilizam
inteiros de 8 bytes. Como os resultados da aritmética de ponteiro dependem do tamanho dos objetos que um ponteiro
aponta, a aritmética de ponteiro é dependente da máquina.

Expressões com ponteiros e aritmética de ponteiros

- Na aritmética convencional, 3000 + 2 gera o valor 3002.
- Isso normalmente não acontece com a aritmética de ponteiro.
- Quando um inteiro é somado ou subtraído de um ponteiro, o ponteiro não é simplesmente incrementado ou decrementado por esse inteiro, mas pelo inteiro multiplicado pelo tamanho do objeto ao qual o ponteiro se refere.
- O número de bytes depende do tipo de dado do objeto.
- Por exemplo, a instrução
  - vPtr += 2;

produzirá 3008 (3000 + 2 * 4), supondo que um inteiro seja armazenado em 4 bytes de memória

Expressões com ponteiros e aritmética de ponteiros

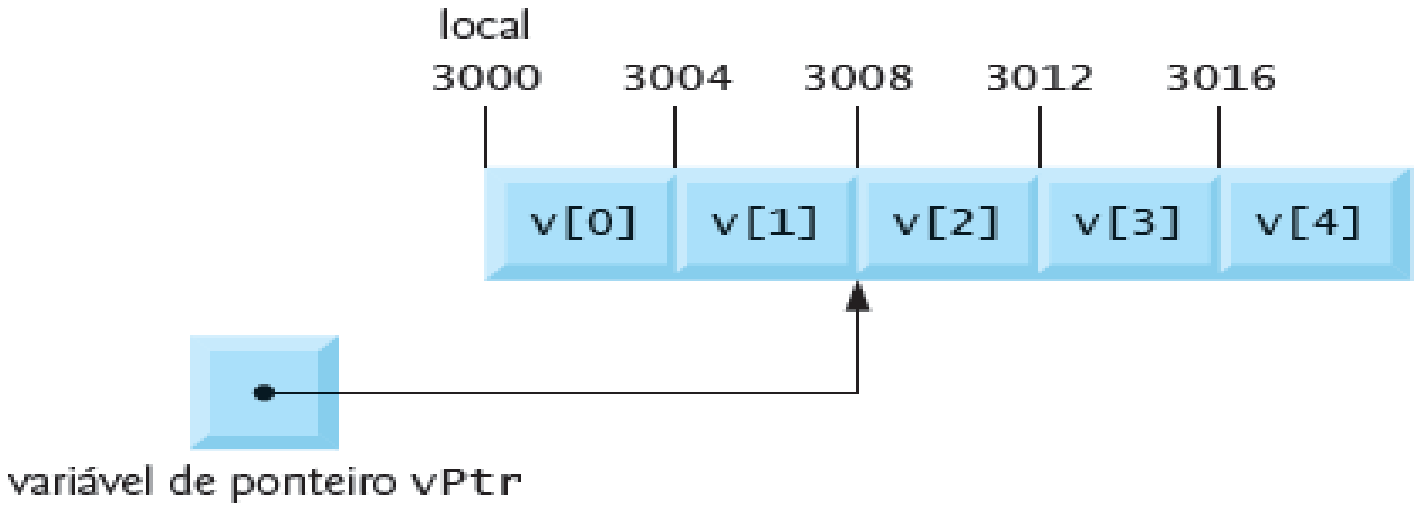


Figura 7.19 ■ O ponteiro `vPtr` após a execução da aritmética de ponteiro.

Expressões com ponteiros e aritmética de ponteiros

- No array v, vPtr agora apontaria v[2] (Figura 7.19).
- Se um inteiro for armazenado em 2 bytes de memória, então o cálculo anterior resultaria no local de memória 3004 (3000 + 2 * 2).
- Se o array fosse de um tipo de dado diferente, a instrução anterior incrementaria o ponteiro pelo dobro do número de bytes necessários para armazenar um objeto desse tipo de dado.
- Os resultados da execução da aritmética de ponteiro em um array de caracteres serão consistentes com a aritmética comum, pois cada caractere tem 1 byte de extensão.

Expressões com ponteiros e aritmética de ponteiros

- Se vPtr tivesse sido incrementado para 3016que aponta para v[4], a instrução
  - vPtr -= 4;

  definiria vPtr novamente em 3000 — o início do array.
- Se um ponteiro estiver sendo incrementado ou decrementado em um, os operadores de incremento (++) e de decremento (--) poderão ser usados.
- incrementa o ponteiro para que ele aponte o próximo local no array.++vPtr;
  vPtr++;

incrementa o ponteiro para que ele aponte o próximo local no array.

Expressões com ponteiros e aritmética de ponteiros

- Qualquer uma das instruções a seguir
  - --vPtr;
    vPtr--;

  decrementa o ponteiro para que ele aponte o elemento anterior no array.
- As variáveis de ponteiro podem ser subtraídas umas das outras.

Expressões com ponteiros e aritmética de ponteiros

- Por exemplo, se vPtr contiver o local 3000, e v2Ptr contiver o endereço 3008, a instrução
  - x = v2Ptr - vPtr;

  atribuirá a x o número de elementos de array de vPtr para v2Ptr, nesse caso, 2 (e não 8).
- A aritmética de ponteiro não fará sentido a menos que seja realizada em um array.
- Não é possível supor que duas variáveis do mesmo tipo serão armazenadas em locais contíguos na memória, a menos que elas sejam elementos adjacentes de um array.

Expressões com ponteiros e aritmética de ponteiros

Erro comum de programação 7.5
Usar aritmética de ponteiro em um ponteiro que não se aplique a um elemento em um array.

Erro comum de programação 7.6
Subtrair ou comparar dois ponteiros que não se refiram a elementos no mesmo array.

Erro comum de programação 7.7
Ultrapassar o final de um array ao usar a aritmética de ponteiro.

Expressões com ponteiros e aritmética de ponteiros

- Um ponteiro pode ser atribuído a outro se ambos forem do mesmo tipo.
- A exceção a essa regra é o ponteiro para void (ou seja, void *), que é um ponteiro genérico que pode representar qualquer tipo de ponteiro.
- Todos os tipos de ponteiro podem receber um ponteiro para void, e este pode receber um ponteiro de qualquer tipo.
- Uma operação de coerção (cast) não é necessária em nenhum desses casos.
- Um ponteiro para void não pode ser desreferenciado.

Expressões com ponteiros e aritmética de ponteiros

- **Considere o seguinte: o compilador sabe que um ponteiro para int refere-se a 4 bytes de memória em uma máquina com inteiros de 4 bytes, mas um ponteiro para void simplesmente contém um local da memória para um tipo de dado desconhecido — o número exato de bytes aos quais o ponteiro se refere não é conhecido pelo compilador.**
- **O compilador precisa conhecer o tipo de dado para determinar o número de bytes a ser desreferenciado para um ponteiro em particular.**

Expressões com ponteiros e aritmética de ponteiros

Erro comum de programação 7.8
*Atribuir um ponteiro de um tipo a um ponteiro de outro tipo se nenhum deles for do tipo void * consiste em um erro de sintaxe.*

Erro comum de programação 7.9
*Desreferenciar um ponteiro void * é um erro de sintaxe.*

Expressões com ponteiros e aritmética de ponteiros

- **Ponteiros podem ser comparados se usarmos operadores de igualdade e relacionais, mas essas comparações não farão sentido, a menos que os ponteiros apontem elementos do mesmo array.**
- **As comparações de ponteiro comparam os endereços armazenados nos ponteiros.**
- **Uma comparação entre dois ponteiros que apontam elementos no mesmo array poderia mostrar, por exemplo, que um deles aponta para um elemento de número mais alto do array do que o outro.**
- **A comparação é comumente usada para determinar se um ponteiro é NULL.**

A relação entre ponteiros e arrays

- Arrays e ponteiros estão intimamente relacionados em C, e geralmente são usados da mesma forma.
- Um nome de array pode ser considerado um ponteiro constante.
- Os ponteiros podem ser usados para realizar qualquer operação que envolva um subscrito de array.
- Suponha que o array de inteiros b[5] e a variável de ponteiro de inteiros bPtr tenham sido definidos.
- Como o nome do array (sem um subscrito) é um ponteiro para o primeiro elemento do array, podemos definir bPtr igual ao endereço do primeiro elemento no array b com a instrução
  - bPtr = b;

A relação entre ponteiros e arrays

- Essa instrução é equivalente a obter o endereço do primeiro elemento do array da seguinte forma:
  - bPtr = &b[0];
- O elemento do array b[3] pode, como alternativa, ser referenciado com a expressão de ponteiro
  - *(bPtr + 3)
- O 3 na expressão acima é o deslocamento até o ponteiro.
- Quando o ponteiro aponta o início de um array, o deslocamento indica qual elemento do array deve ser referenciado, e o valor do deslocamento é idêntico ao subscrito do array.
- A notação anterior é conhecida como notação de ponteiro/deslocamento

A relação entre ponteiros e arrays

- Os parênteses são necessários porque a precedência de * é mais alta que a precedência de +.
- Sem os parênteses, a expressão acima somaria 3 ao valor da expressão *bPtr (ou seja, 3 seria somado a b[0], supondo que bPtr aponte para o início do array).
- Assim como o elemento do array pode ser referenciado com uma expressão com ponteiro, o endereço
  - &b[3]

  pode ser escrito com a expressão com ponteiro
  - bPtr + 3
- O próprio array pode ser tratado como um ponteiro e usado na aritmética de ponteiro.

A relação entre ponteiros e arrays

- Por exemplo, a expressão
  - `*( b + 3 )`

  também se refere ao elemento de array b[3].
- Em geral, todas as expressões utilizando valores de array subscritadas podem ser escritas com um ponteiro e um deslocamento.
- Nesse caso, a notação de ponteiro/deslocamento foi usada com o nome do array como um ponteiro.
- A instrução anterior não modifica o nome do array de nenhuma forma; b ainda aponta o primeiro elemento no array.
- Os ponteiros podem ser subscritados exatamente como os arrays.

A relação entre ponteiros e arrays

- Por exemplo, se bPtr tem o valor b, a expressão
  - `bPtr[ 1 ]`

  refere-se ao elemento de array b[1].
- Isso é conhecido como notação de ponteiro/subscrito.
- Lembre-se de que um nome de array é basicamente um ponteiro constante; ele sempre aponta para o início do array.
- Assim, a expressão
  - `b += 3`

  é inválida, pois tenta modificar o valor do nome do array com a aritmética de ponteiro.

A relação entre ponteiros e arrays

Erro comum de programação 7.10

Tentar modificar um nome de array usando aritmética de ponteiro consiste em um erro de sintaxe.

A relação entre ponteiros e arrays

```
1  /* Fig. 7.20: fig07_20.cpp
2     Usando notações de subscrito e ponteiro com arrays */
3
4  #include <stdio.h>
5
6  int main( void )
7  {
8     int b[] = { 10, 20, 30, 40 }; /* inicializa array b */
9     int *bPtr = b; /* define bPtr para apontar para array b */
10    int i; /* contador */
11    int offset; /* contador */
12
13    /* mostra array b usando notação de subscrito de array */
14    printf( "Array b impresso com:\nNotação de subscrito de array\n" );
15
16    /* loop pelo array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* fim do for */
20
21    /* mostra array b usando nome do array e notação de ponteiro/deslocamento */
22    printf( "\nNotação de ponteiro/offset onde\n"
23            "o ponteiro é o nome do array\n" );
24
25    /* loop pelo array b */
26    for ( offset = 0; offset < 4; offset++ ) {
27       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28    } /* fim do for */
29
```

A relação entre ponteiros e arrays

```
30      /* mostra array b usando bPtr e notação de subscrito de array */
31      printf( "\nNotação de subscrito de ponteiro\n" );
32
33      /* loop pelo array b */
34      for ( i = 0; i < 4; i++ ) {
35         printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36      } /* fim do for */
37
38      /* mostra array b usando bPtr e notação de ponteiro/deslocamento */
39      printf( "\nNotação de ponteiro/deslocamento\n" );
40
41      /* loop pelo array b */
```

Figura 7.20 ■ Usando os quatro métodos para referenciar elementos do array. (Parte 1 de 2.)

A relação entre ponteiros e arrays

```
42      for ( offset = 0; offset < 4; offset++ ) {
43         printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44      } /* fim do for */
45
46      return 0; /* indica conclusão bem-sucedida */
47   } /* fim do main */
```

```
Array b impresso com:
Notação de subscrito de array
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Notação de ponteiro/deslocamento onde
o ponteiro é o nome do array
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Notação de subscrito de ponteiro
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Notação de ponteiro/deslocamento
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Figura 7.20 ■ Usando os quatro métodos para referenciar elementos do array. (Parte 2 de 2.)

A relação entre ponteiros e arrays

- A Figura 7.20 usa os quatro métodos que discutimos para referenciar elementos do array — subscrito do array, ponteiro/deslocamento com o nome do array como um ponteiro, subscrito de ponteiro e ponteiro/deslocamento com um ponteiro — para imprimir os quatro elementos do array de inteiros b.

A relação entre ponteiros e arrays

```
1  /* Fig. 7.21: fig07_21.c
2     Copiando uma string usando notação de array e notação de ponteiro. */
3  #include <stdio.h>
4
5  void copy1( char * const s1, const char * const s2 ); /* protótipo */
6  void copy2( char *s1, const char *s2 ); /* protótipo */
7
8  int main( void )
9  {
10    char string1[ 10 ]; /* cria array string1 */
11    char *string2 = "Olá"; /* cria um ponteiro para uma string */
12    char string3[ 10 ]; /* cria array string3 */
13    char string4[] = "Adeus"; /* cria um ponteiro para uma string */
14
15    copy1( string1, string2 );
16    printf( "string1 = %s\n", string1 );
```

Figura 7.21 ■ Copiando uma string usando a notação de array e a notação de ponteiro. (Parte 1 de 2.)

A relação entre ponteiros e arrays

```
17
18      copy2( string3, string4 );
19      printf( "string3 = %s\n", string3 );
20      return 0; /* indica conclusão bem-sucedida */
21   } /* fim do main */
22
23   /* copia s2 para s1 usando notação de array */
24   void copy1( char * const s1, const char * const s2 )
25   {
26      int i; /* contador */
27
28      /* loop pelas strings */
29      for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
30         ; /* não faz nada no corpo */
31      } /* fim do for */
32   } /* fim da função copy1 */
33
34   /* copia s2 para s1 usando notação de ponteiro */
35   void copy2( char *s1, const char *s2 )
36   {
37      /* loop pelas strings */
38      for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
39         ; /* não faz nada no corpo */
40      } /* fim do for */
41   } /* fim da função copy2 */
```

```
string1 = Olá
string3 = Adeus
```

Figura 7.21 ■ Copiando uma string usando a notação de array e a notação de ponteiro. (Parte 2 de 2.)

A relação entre ponteiros e arrays

- Para ilustrar um pouco a permutabilidade de arrays e ponteiros, vejamos duas funções de cópia de string —copy1 e copy2 — no programa da Figura 7.21.
- As duas funções copiam uma string (possivelmente, um array de caracteres) para um array de caracteres.
- Após uma comparação dos protótipos de função paracopy1 e copy2, as funções parecem ser idênticas.
- Elas realizam a mesma tarefa; porém, são implementadas de formas diferentes.

A relação entre ponteiros e arrays

- A função copy1 usa a notação de subscrito de array para copiar a string de s2 no array de caracteres s1.
- A função define a variável contadora i como o subscrito do array.
- O cabeçalho da estrutura for (linha 29) realiza a operação de cópia inteira — em seu corpo não há comando.
- O cabeçalho especifica que i é inicializado em zero e incrementado em um a cada iteração do loop.
- A expressão s1[i] = s2[i] copia um caractere de s2 em s1.
- Quando o caractere nulo é encontrado em s2, ele é atribuído a s1, e o valor da atribuição se torna o valor atribuído ao operando da esquerda (s1).

A relação entre ponteiros e arrays

- O loop termina porque o valor inteiro do caractere nulo é zero (falso).
- A função copy2 usa ponteiros e aritmética de ponteiro para copiar a string de s2 no array de caracteres s1.
- Novamente, o cabeçalho da estrutura for (linha 38) realiza a operação de cópia inteira.
- O cabeçalho não inclui qualquer inicialização de variável.
- Como na funçãon copy1, a expressão (*s1 = *s2) realiza a operação de cópia.
- O ponteiro s2 é desreferenciado e o caractere resultante é atribuído ao ponteiro desreferenciado*s1.

A relação entre ponteiros e arrays

- Após a atribuição na condição, os ponteiros são incrementados para que apontem para o próximo elemento do array s1 e o próximo caractere de s2, respectivamente.
- Quando o caractere nulo é encontrado em s2, ele é atribuído ao ponteiro desreferenciado s1 e o loop termina.
- Os primeiros argumentos de copy1 e copy2 precisam ser um array grande o suficiente para manter a string no segundo argumento.
- Caso contrário, é possível que ocorra um erro quando houver uma tentativa de escrever no local da memória que não faz parte do array.
- Além disso, o segundo parâmetro de cada função é declarado como const char * (uma string constante).

A relação entre ponteiros e arrays

- **Em ambas as funções, o segundo argumento é copiado no primeiro argumento — os caracteres são lidos ali um por vez, mas nunca são modificados.**
- **Portanto, o segundo parâmetro é declarado para apontar um valor constante, de modo que o princípio do menor privilégio é imposto — nenhuma função requer a capacidade de modificar o segundo argumento, para que essa capacidade não é fornecida a nenhuma delas.**

Arrays de ponteiros

- Arrays podem conter ponteiros.
- Um array de ponteiros é comumente usado para formar um array de strings.
- Cada entrada no array é uma string, mas em C uma string é, basicamente, um ponteiro para o seu primeiro caractere.
- Assim, cada entrada em um array de strings é, na realidade, um ponteiro para o primeiro caractere de uma string.
- Considere a definição do array de strings suit, (naipe), que poderia ser útil na representação de um baralho.
  - `const char *suit[ 4 ] = { "Copas", "Ouros", "Paus", "Espadas" };`

Arrays de ponteiros

- A parte suit[4] da definição indica um array de 4 elementos.
- A parte char * da declaração indica que cada elemento do array suit é do tipo ‘ponteiro para char’.
- O qualificador const indica que as strings apontadas por ponteiro de elemento não serão modificadas.
- Os quatro valores a serem colocados no array são “Copas", “Ouros", “Paus" e “Espadas".
- Cada um é armazenado na memória como uma string de caracteres terminada em nulo, que é um caractere maior que o número de caracteres entre as aspas.

Arrays de ponteiros

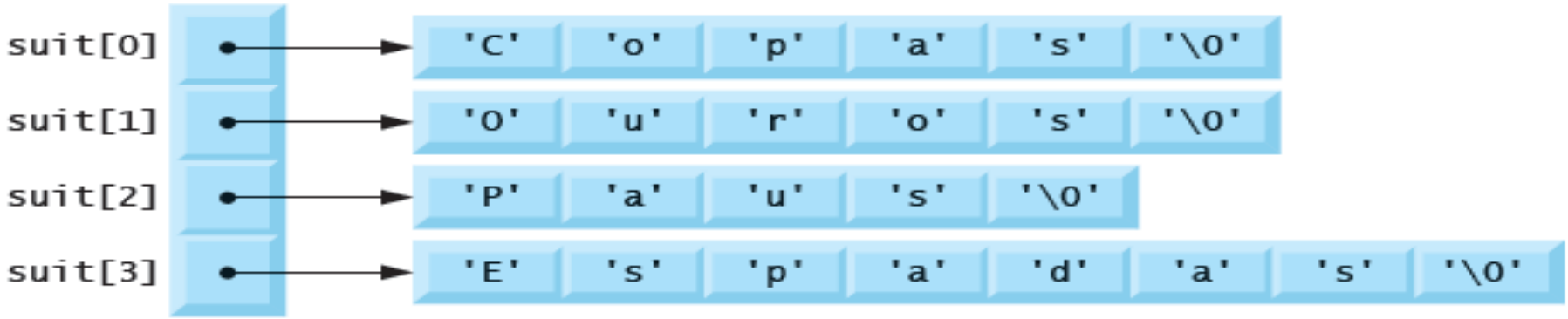


Figura 7.22 ■ Representação gráfica do array `suit`.

Arrays de ponteiros

- As quatro strings possuem 6, 6, 5 e 8 caracteres de extensão, respectivamente.
- Embora pareça que essas strings estejam sendo colocadas no array suit, somente ponteiros são realmente armazenados no array (Figura 7.22).
- Cada ponteiro aponta para o primeiro caractere de sua string correspondente.
- Assim, embora o array suit seja fixo em tamanho, ele oferece acesso a strings de caracteres de qualquer tamanho.
- Essa flexibilidade é um exemplo das poderosas capacidades de estruturação de dados da linguagem C.

Arrays de ponteiros

- Os naipes poderiam ter sido colocados em um array bidimensional, no qual cada linha representaria um naipe, e cada coluna representaria uma letra do nome do naipe.
- Essa estrutura de dados precisaria ter um número fixo de colunas por linha, e esse número teria de ser tão grande quanto a maior das strings.
- Portanto, uma memória considerável poderia ser desperdiçada quando um número grande de strings estivesse sendo armazenado, sendo a maior parte das strings mais curtas do que a string mais longa.

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

- Nesta seção, usaremos a geração de números aleatórios para desenvolver um programa que simule o embaralhamento e a distribuição de cartas.
- Esse programa poderá ser usado, então, para implementar programas que simulam jogos de carta específicos.
- Para revelar alguns problemas de desempenho sutis, usaremos intencionalmente algoritmos para embaralhar e distribuir cartas abaixo do ideal.
- Nos exercícios deste capítulo e no Capítulo 10, desenvolveremos algoritmos mais eficientes.
- Usando a técnica de refinamentos sucessivos, top-down, desenvolveremos um programa que embaralhará 52 cartas de jogo, e depois distribuirá cada uma delas.

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

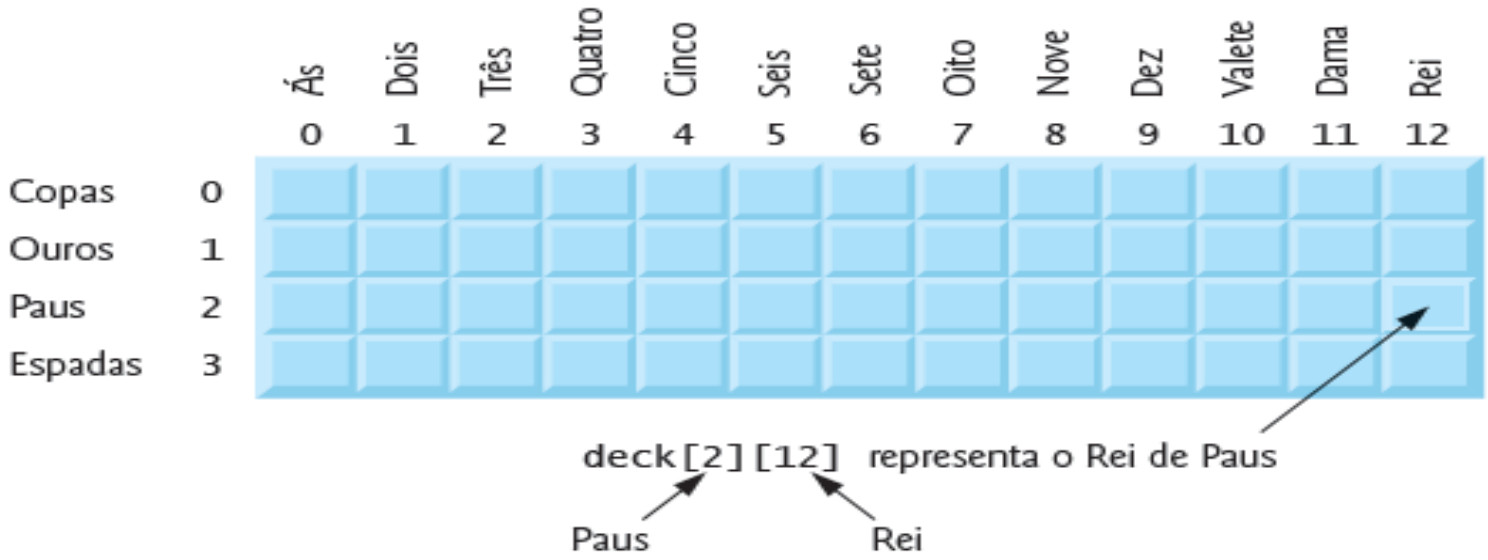


Figura 7.23 ■ Representação de um array com duplo subscrito de um baralho.

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

- A técnica top-down é particularmente útil no ataque a problemas maiores e mais complexos que aqueles vistos
- nos capítulos iniciais.
- Usamos um array deck com duplo subscrito, de 4 por 13, para representar o baralho (Figura 7.23).
- As linhas correspondem aos naipes — linha 0 corresponde a copas, linha 1 a ouros, linha 2 a paus e linha 3 a espadas.
- As colunas correspondem aos valores de face das cartas — as colunas de 0 a 9 correspondem a ás até 10, respectivamente, e as colunas de 10 a 12 correspondem a valete, dama e rei.
- Carregaremos o array de strings suit com strings de caracteres que representem os quatro naipes, e o array de strings face com strings de caracteres que representem os treze valores de face.

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

- Esse baralho simulado pode ser embaralhado da seguinte forma.
- Primeiro, o array deck é zerado.
- Depois, uma linha (0–3) e uma coluna (0–12) são escolhidas aleatoriamente.
- O número 1 é inserido no elemento do array deck[linha][coluna] para indicar que essa carta será a primeira a ser distribuída.
- Esse processo continua com os números 2, 3, ..., 52 sendo inseridos aleatoriamente no array deck para indicar quais cartas devem ser colocadas em segundo, terceiro, ... e quinquagésimo segundo lugar no baralho.

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

- À medida que o array deck começa a ser preenchido com números de carta, é possível que uma carta seja selecionada duas vezes — ou seja, deck[linha] [coluna] será diferente de zero quando for selecionado.
- Essa seleção é simplesmente ignorada, e outras rows e columns são repetidamente escolhidas aleatoriamente, até que uma carta não selecionada seja encontrada.
- Eventualmente, os números de 1 a 52 ocuparão os 52 slots do array deck.
- Nesse ponto, as cartas estarão totalmente embaralhadas.

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

- Esse algoritmo de embaralhamento poderia ser executado indefinidamente se as cartas que já tivessem sido embaralhadas fossem repetidamente selecionadas de forma aleatória.
- Esse fenômeno é conhecido como **adiamento indefinido**.
- Nos exercícios, discutiremos um algoritmo de embaralhamento melhor, que elimina a possibilidade do adiamento indefinido.

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

Dica de desempenho 7.3

Às vezes, um algoritmo que surge de uma maneira 'natural' pode conter problemas de desempenho sutis, como o de adiamento indefinido. Procure algoritmos que evitem o adiamento indefinido.

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

- Para dar a primeira carta, procuramos o deck[row][column] igual a 1 no array.
- Isso é feito com uma estrutura for aninhada, que varia row de 0 a 3 e column de 0 a 12.
- A que carta esse elemento do array corresponde?
- O array suit foi previamente carregado com os quatro naipes, de modo que, para obter o naipe, imprimimos a string de caracteres suit[row].
- De maneira semelhante, para obter o valor de face da carta, imprimimos a string de caracteres face[column].
- Também imprimimos a string de caracteres ‘de’.

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

- A impressão dessa informação na ordem correta permite imprimir cada carta na forma "Rei de Paus", "Às de Ouros" e assim por diante.
- Continuemos com o processo de refinamentos sucessivos top-down.
- No topo teremos, simplesmente,
  - *Embaralhar e distribuir 52 cartas*
- Nosso primeiro refinamento gera:
  - Inicializar o array de naipes
    Inicializar o array de faces
    Inicializar o array do baralho
    Embaralhar
    Distribuir 52 cartas

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

- ‘Embaralhar’ pode ser desenvolvido da seguinte forma:
  - Para cada uma das 52 cartas
    Colocar aleatoriamente o número da carta em um slot desocupado do baralho
- ‘Distribuir 52 cartas’ pode ser desenvolvido da seguinte forma:
  - Para cada uma das 52 cartas
    Achar número de carta no array do baralho e imprimir face e naipe da carta

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

- A incorporação desses desenvolvimentos gera nosso segundo refinamento completo:
  - Inicializar o array de naipes
    Inicializar o array de faces
    Inicializar o array do baralho

    Para cada uma das 52 cartas
    Colocar aleatoriamente o número da carta em um espaço desocupado do baralho

    Para cada uma das 52 cartas
    Achar número da carta no array do baralho e imprimir face e naipe da carta

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

- ‘Colocar aleatoriamente o número da carta em um espaço desocupado do baralho’ pode ser desenvolvido como:
  - Escolher espaço do baralho aleatoriamente

    Embora espaço escolhido tenha sido escolhido anteriormente
    Escolher espaço do baralho aleatoriamente

    Colocar número da carta no espaço escolhido do baralho
- ‘Achar número da carta no array do baralho e imprimir face e naipe da carta’ pode ser desenvolvido como:
  - Para cada espaço do array do baralho
    Se o espaço contém número da carta
     Imprimir face e naipe da carta

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

- **A incorporação desses desenvolvimentos gera nosso terceiro refinamento:**
  - **Inicializar o array de naipes**
    Inicializar o array de faces
    Inicializar o array do baralho

    Para cada uma das 52 cartas
    Escolher espaço do baralho aleatoriamente

    Embora espaço escolhido tenha sido escolhido anteriormente
    Escolher espaço do baralho aleatoriamente

    Colocar número da carta no espaço escolhido do baralho

    Para cada uma das 52 cartas
    Para cada espaço do array do baralho
     Se o espaço contém número desejado de carta
     Imprimir a face e o naipe da carta

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

- Isso completa o processo de refinamento.
- Esse programa é mais eficiente se as partes de embaralhar e distribuir do algoritmo forem combinadas, de modo que cada carta seja distribuída assim que for colocada no baralho.
- Escolhemos programar essas operações separadamente, porque, normalmente, as cartas são distribuídas depois de serem embaralhadas (não enquanto estão sendo embaralhadas).

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

```c
/* Fig. 7.24: fig07_24.c
   Programa de embaralhamento e distribuição de cartas */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* protótipos */
void embaralha( int wbaralho[][ 13 ] );
void distribui( const int wbaralho[][ 13 ], const char *wNaipe[],
            const char *wNaipe[] );

int main( void )
{
   /* inicializa array naipe */
   const char *naipe[ 4 ] = { "Copas", "Ouros", "Paus", "Espadas" };

   /* inicializa array naipe */
   const char *naipe[ 13 ] =
      { "Ás", "Dois", "Três", "Quatro",
        "Cinco", "Seis", "Sete", "Oito",
        "Nove", "Dez", "Valete", "Dama", "Rei" };

   /* inicializa array baralho */
   int baralho[ 4 ][ 13 ] = { 0 };

   srand( time( 0 ) ); /* semente do gerador de número aleatório */

   shuffle( baralho ); /* embaralha */
   distribui( deck, face, suit ); /* distribui as cartas do baralho */
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */

/* embaralha cartas */
void embaralha( int wbaralho[][ 13 ] )
{
```

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

```
36      int linha; /* número de linha */
37      int coluna; /* número de coluna */
38      int carta; /* contador */
39
40      /* para cada uma das 52 cartas, escolhe slot de deck aleatoriamente */
41      for ( carta = 1; carta <= 52; carta++ ) {
42
43         /* escolhe novo local aleatório até que slot não ocupado seja encontrado */
44         do {
45            linha = rand() % 4;
46            coluna = rand() % 13;
47         } while( wBaralho[ linha ][ coluna ] != 0 ); /* fim do do...while */
48
49         /* coloca número da carta no slot escolhido do baralho */
50         wBaralho[ linha ][ coluna ] = carta;
51      } /* fim do for */
52   } /* fim da função shuffle */
53
```

Figura 7.24 ■ Programa de distribuição de cartas. (Parte 1 de 2.)

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

```
54  /* distribui cartas no baralho */
55  void distribui( const int wBaralho[][ 13 ], const char *wNaipe[],
56  const char *wNaipe[] )
57  {
58     int carta; /* contador de cartas */
59     int linha; /* contador de linhas */
60     int coluna; /* contador de coluna */
61
62     /* distribui cada uma das 52 cartas */
63     for ( carta = 1; carta <= 52; carta++ ) {
64        /* loop pelas linhas de wBaralho */
65
66        for ( linha = 0; linha <= 3; linha++ ) {
67
68           /* loop pelas colunas de wBaralho para linha atual */
69           for ( coluna = 0; coluna <= 12; coluna++ ) {
70
71              /* se slot contém cartão atual, mostra carta */
72              if ( wBaralho[ linha ][ coluna ] == carta ) {
73                 printf( "%5s of %-8s%c", wNaipe[ coluna ], wNaipe[ linha ],
74                    carta % 2 == 0 ? '\n' : '\t' );
75              } /* fim do if */
76           } /* fim do for */
77        } /* fim do for */
78     } /* fim do for */
79  } /* fim da função distribui */
```

Figura 7.24 ■ Programa de distribuição de cartas. (Parte 2 de 2.)

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

```
  Nove de Copas              Cinco de Paus
  Dama de Espadas             Três de Espadas
  Dama de Copas                 Ás de Paus
   Rei de Copas              Seis de Espadas
Valete de Ouros             Cinco de Espadas
  Sete de Copas               Rei de Paus
  Três de Paus               Oito de Copas
  Três de Ouros            Quatro de Ouros
  Dama de Ouros             Cinco de Ouros
  Seis de Ouros             Cinco de Copas
    Ás de Espadas            Seis de Copas
  Nove de Ouros              Dama de Paus
  Oito de Espadas            Nove de Paus
  Dois de Paus               Seis de Paus
  Dois de Espadas          Valete de Paus
Quatro de Paus               Oito de Paus
Quatro de Espadas            Sete de Espadas
  Sete de Ouros              Sete de Paus
   Rei de Espadas             Dez de Ouros
Valete de Copas               Ás de Copas
Valete de Espadas            Dez de Paus
  Oito de Ouros             Dois de Ouros
    Ás de Ouros             Nove de Espadas
Quatro de Copas             Dois de Copas
   Rei de Ouros              Dez de Espadas
  Três de Copas              Dez de Copas
```

Figura 7.25 ■ Exemplo de execução do programa de distribuição de cartas.

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

- O programa de embaralhamento e distribuição de cartas aparece na Figura 7.24, e um exemplo de execução é mostrado na Figura 7.25.
- O especificador de conversão %s é usado para imprimir strings de caracteres nas chamadas printf.
- O argumento correspondente na chamada printf dente na chamada char (ou um array de char).
- A especificação de formato "%5s de %-8s" (linha 73) imprime uma string de caracteres alinhada à direita em um campo de cinco caracteres, seguido por ‘de’ e por uma string de caracteres alinhada à esquerda em um campo de oito caracteres.
- O sinal de menos em %-8s significa alinhamento à esquerda.

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

- Há um ponto fraco no algoritmo de distribuição.
- Quando uma combinação é encontrada, as duas estruturas for internas continuam a procurar uma combinação nos elementos restantes de deck.
- Corrigiremos essa deficiência nos exercícios referentes a este capítulo e em um estudo de caso do Capítulo 10.

Ponteiros para funções

- Um ponteiro para uma função contém o endereço da função na memória.
- Vimos que o nome de um array é, na realidade, o endereço na memória do primeiro elemento do array.
- De modo semelhante, um nome de 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 para funções.

Ponteiros para funções

```c
/* Fig. 7.26: fig07_26.c
   Programa de classificação de múltiplas finalidades usando ponteiros para função */
#include <stdio.h>
#define SIZE 10

/* protótipos */
void bubble( int work[], const int size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order; /* 1 para ordem crescente ou 2 para ordem decrescente */
   int counter; /* contador */

   /* inicializa array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Digite 1 para classificar em ordem crescente,\n"
           "Digite 2 para classificar em ordem decrescente: " );
   scanf( "%d",  &order );

   printf( "\nItens de dados na ordem original\n" );

   /* mostra array original */
   for ( contador = 0; contador < SIZE; contador++ ) {
      printf( "%5d", a[ contador ] );
   } /* fim do for */

   /* classifica array em ordem crescente; passa função crescente como
      um argumento para especificar classificação crescente */
   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
      printf( "\nItens de dados em ordem crescente\n" );
   } /* fim do if */
   else { /* passa função decrescente */
      bubble( a, SIZE, decrescente );
      printf( "\nItens de dados em ordem decrescente\n" );
   } /* fim do else */
```

Figura 7.26 ■ Programa de classificação de múltipla finalidade usando ponteiros para função. (Parte 1 de 2.)

Ponteiros para funções

```
40
41       /* mostra array ordenado */
42       for ( contador = 0; contador < SIZE; contador++ ) {
43          printf( "%5d", a[ contador ] );
44       } /* fim do for */
45
46       printf( "\n" );
47       return 0; /* indica conclusão bem-sucedida */
48    } /* fim do main */
49
50    /* bubble sort de múltipla finalidade; parâmetro compare é um ponteiro
51       para a função de comparação que determina classificação */
52    void bubble( int work[], const int size, int (*compare)( int a, int b ) )
53    {
54       int passada; /* contador de passadas */
55       int contador; /* contador de comparação */
56
57       void inverte( int *element1Ptr, int *element2ptr ); /* protótipo */
58
59       /* loop para controlar passadas */
60       for ( pass = 1; pass < size; pass++ ) {
61
62          /* loop para controlar número de comparações por passada */
63          for ( contador = 0; contador < size - 1; contador++ ) {
64
65             /* se elementos adjacentes estão fora de ordem, inverta-os */
66             if ( (*compare)( trabalho[ contador ], trabalho[ contador + 1 ] ) ) {
67                inverte( &trabalho[ contador ], &trabalho[ contador + 1 ] );
68             } /* fim do if */
69          } /* fim do for */
70       } /* fim do for */
71    } /* fim da função bubble */
72
```

Ponteiros para funções

```
73  /* troca valores nos locais da memória aos quais element1Ptr e
74     element2Ptr apontam */
75  void inverte( int *element1Ptr, int *element2Ptr )
76  {
77     int manutenção; /* variável de manutenção temporária */
78
79     manutenção = *element1Ptr;
80     *element1Ptr = *element2Ptr;
81     *element2Ptr = manutenção;
82  } /* fim da função inverte */
83
84  /* determina se os elementos estão fora de ordem para uma ordem
85     de classificação crescente */
86  int crescente( int a, int b )
87  {
88     return b < a; /* troca se b for menor que a */
89  } /* fim da função crescente */
90
91  /* determina se os elementos estão fora de ordem para uma ordem
92     de classificação decrescente */
93  int decrescente( int a, int b )
94  {
95     return b > a; /* troca se b for maior que a */
96  } /* fim da função decrescente */
```

Figura 7.26 ■ Programa de classificação de múltipla finalidade usando ponteiros para função. (Parte 2 de 2.)

Ponteiros para funções

```
Digite 1 para classificar em ordem crescente,
Digite 2 para classificar em ordem decrescente: 1

Itens de dados na ordem original
2  6  4  8  10  12  89  68  45  37
Itens de dados em ordem crescente
2  4  6  8  10  12  37  45  68  89
```

```
Digite 1 para classificar em ordem crescente,
Digite 2 para classificar em ordem decrescente: 2

Itens de dados na ordem original
 2   6   4   8  10  12  89  68  45  37
Itens de dados em ordem decrescente
89  68  45  37  12  10   8   6   4   2
```

Figura 7.27 ■ As saídas do programa de bubble sort da Figura 7.26.

Ponteiros para funções

- Para ilustrar o uso de ponteiros para funções, a Figura 7.26 apresenta uma versão modificada do programa bubble sort na Figura 7.15.
- A nova versão consiste em main e nas funções bubble, swap, ascending e descending.
- A função bubbleSort recebe um ponteiro para uma função — ascending ou descending — como um argumento, além de um array de inteiros e o tamanho do array.

Ponteiros para funções

- O programa pede ao usuário que escolha se o array deve ser classificado em ordem crescente ou decrescente.
- Se o usuário digitar 1, um ponteiro para a função ascending é passado para a função bubble, fazendo com que o array seja classificado em ordem crescente.
- Se o usuário digitar 2, um ponteiro para a função descending é passado para a função bubble, fazendo com que o array seja classificado em ordem decrescente.
- A saída do programa aparece na Figura 7.27.

Ponteiros para funções

- O parâmetro a seguir aparece no cabeçalho da função para bubble (linha 52)
  - `int (*compare)( int a, int b )`
- Isso diz a bubble que espere por um parâmetro (compare) que é um ponteiro para uma função que recebe dois parâmetros inteiros e retorna um resultado inteiro.

Ponteiros para funções

- Os parênteses são necessários em torno de *compare para agrupar* com compare a fim de indicar que compare é um ponteiro.
- Se não tivéssemos incluído os parênteses, a declaração seria
  - `int *compare( int a, int b )`

que declara uma função que recebe dois inteiros como parâmetros e retorna um ponteiro para um inteiro.

Ponteiros para funções

- O protótipo de função para bubble aparece na linha 7.
- O protótipo poderia ter sido escrito como
  - `int (*)( int, int );`
- sem o nome do ponteiro para função e os nomes de parâmetros.
- A função passada a bubble é chamada em uma estrutura if (linha 66) da seguinte forma:
  - `if ((*compare)( work[contador], work[contador + 1]))`
- Assim como um ponteiro para uma variável é desreferenciado para acessar o valor da variável, um ponteiro para uma função é desreferenciado para usar a função.

Ponteiros para funções

- A chamada para a função poderia ter sido feita sem desreferenciar o ponteiro, como em
  - `if (compare(work[contador], work[contador + 1]))`
- que usa o ponteiro diretamente como o nome da função.
- Preferimos o primeiro método, ou seja, chamar uma função por meio de um ponteiro, pois isso ilustra explicitamente que compare é um ponteiro para uma função que é desreferenciada para chamar a função.
- O segundo método de chamada de uma função por meio de um ponteiro faz parecer que compare é uma função real.
- Isso pode ser confuso para um usuário do programa que gostaria de ver a definição da função compare e acaba descobrindo que ela nunca é definida no arquivo.

Ponteiros para funções

- Os ponteiros para função são comumente usados nos sistemas controlados por menu de texto.
- Um usuário precisa selecionar uma opção a partir de um menu (possivelmente, de 1 a 5), digitando o número do item de menu.
- Cada opção é atendida por uma função diferente.
- Os ponteiros para cada função são armazenados em um array de ponteiros para funções.
- A escolha do usuário é utilizada como um subscrito no array, e o ponteiro no array é usado para chamar a função.

Ponteiros para funções

```c
/* Fig. 7.28: fig07_28.c
   Demonstrando um array de ponteiros para funções */
#include <stdio.h>

/* protótipos */
void function1( int a );
void function2( int b );
void function3( int c );

int main( void )
{
   /* inicializa array de 3 ponteiros para funções que usam um
   argumento int e retornam void */
   void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice; /* variável para manter escolha do usuário */

   printf( "Digite um número entre 0 e 2, 3 para sair: " );
   scanf( "%d", &choice );

   /* processa escolha do usuário */
   while ( choice >= 0 && choice < 3 ) {

      /* chama a função para o local selecionado do array f e passa
         choice como argumento */
      (*f[ choice ])( choice );

      printf( "Digite um número entre 0 e 2, 3 para terminar: ");
      scanf( "%d", &choice );
   } /* fim do while */

   printf( "Execução do programa concluída.\n" );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */

```

Ponteiros para funções

```
36  void function1( int a )
37  {
38     printf( "Você digitou %d, de modo que function1 foi chamada\n\n", a );
39  } /* fim de function1 */
40
41  void function2( int b )
42  {
43     printf( "Você digitou %d, de modo que function2 foi chamada\n\n", b );
44  } /* fim de function2 */
45
46  void function3( int c )
47  {
48     printf( "Você digitou %d, de modo que function3 foi chamada\n\n", c );
49  } /* fim de function3 */
```

Figura 7.28 ■ Demonstração de um array de ponteiros para funções. (Parte 1 de 2.)

Ponteiros para funções

```
Digite um número entre 0 e 2, 3 para sair: 0
Você digitou 0, de modo que function1 foi chamada

Digite um número entre 0 e 2, 3 para sair: 1
Você digitou 1, de modo que function2 foi chamada

Digite um número entre 0 e 2, 3 para sair: 2
Você digitou 2, de modo que function3 foi chamada

Digite um número entre 0 e 2, 3 para sair: 3
Execução do programa concluída.
```

Figura 7.28 ■ Demonstração de um array de ponteiros para funções. (Parte 2 de 2.)

Ponteiros para funções

- A Figura 7.28 oferece um exemplo genérico da mecânica de definição e de uso de um array de ponteiros para funções.
- Definimos três funções — function1, function2 e function3 — que usam um argumento inteiro e não retornam nada. Armazenamos
- ponteiros para essas três funções no array f, que é definido na linha 14.

Ponteiros para funções

- A definição é lida a partir do conjunto de parênteses mais à esquerda, ‘f é um array três ponteiros ponteiros para funções, cada um usando um int como argumento e retornando void’. O array é inicializado com os nomes das três funções.
- Quando o usuário digita um valor entre 0 e 2, o valor é usado como subscrito para o array de ponteiros para funções.
- Na chamada de função (linha 26), f[choice] seleciona o ponteiro no local definido por choice no array.
- O ponteiro é desreferenciado para chamar a função, e choice é passado como argumento para a função.
- Cada função imprime o valor de seu argumento e seu nome de função para demonstrar que a função foi chamada corretamente.

Lembre-se

“ Simplicidade é a alma da eficiência. ”

Austin Freeman