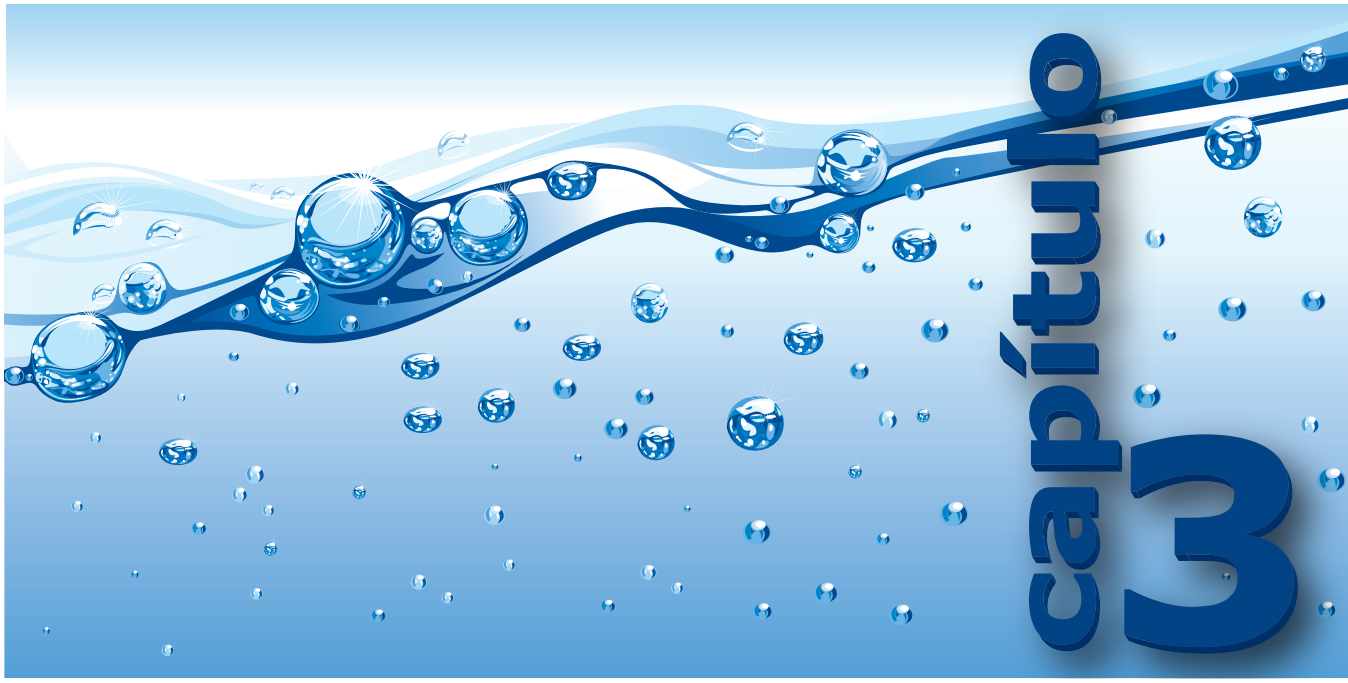

Tipos de Dados Estruturados

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia:
  - ❒ Array
  - ❒ Índice
  - ❒ Zumbi
  - ❒ Estrutura
  - ❒ Fator de escala
  - ❒ Qualificador const
  - ❒ Definição de tipo
  - ❒ Operador sizeof
  - ❒ Rótulo de estrutura
  - ❒ Campo de estrutura
  - ❒ Operador ponto
  - ❒ Operador seta
  - ❒ União
  - ❒ Definidor de tipo
  - ❒ Array unidimensional
  - ❒ Array multidimensional
  - ❒ Acesso sequencial de array
  - ❒ Estrutura com autorreferência
  - ❒ String constante
  - ❒ Classificação de caractere
  - ❒ Bubble Sort
- Descrever as operações aritméticas permitidas sobre ponteiros
- Expressar o endereço e o valor de um elemento de array usando aritmética de ponteiros
- Justificar o fato de indexação de arrays em C começar em zero
- Interpretar declaração de variável ou parâmetro que usa **const**
- Usar array como parâmetro de função
- Implementar um programa que recebe entrada por meio de parâmetros
- Evitar erro de programação causado por zumbi
- Usar as principais funções de processamento de strings da biblioteca padrão de C
- Implementar o algoritmo de ordenação conhecido como Bubble Sort

objetivos

VARIÁVEIS ESTRUTURADAS constituem o foco principal deste capítulo, que explora os seguintes tópicos:

- Arrays (v. **Seção 3.1**)
- Aritmética de ponteiros (v. **Seção 3.2**)
- Uso de **const** (v. **Seção 3.4**)
- Strings e caracteres (v. **Seção 3.7**)
- Função **main()** com parâmetros (v. **Seção 3.8**)
- Tipos definidos pelo programador (v. **Seção 3.9**)
- Estruturas e uniões (v. **Seção 3.10**)
- Operadores de acesso e definidores de tipos (v. **Seção 3.11**)

3.1 Arrays

3.1.1 Definições de Arrays

Array é uma variável estruturada e homogênea (v. **Seção 5.1**) que consiste numa coleção de variáveis do mesmo tipo armazenadas contiguamente em memória. Cada variável que compõe um array é denominada **elemento** e pode ser acessada usando o nome do array e um **índice**. Em C, índices são valores inteiros não negativos e o elemento inicial de um array sempre tem índice igual a zero.

A definição de um array, em sua forma mais simples, consiste em:

tipo nome-do-array [tamanho-do-array];

Nesse esquema de definição, tipo é o tipo de cada elemento do array e tamanho-do-array consiste em uma expressão inteira, constante e positiva que especifica o número de elementos do array. Por exemplo, a seguinte linha de um programa:

```
double notas[50];
```

define a variável `notas[]` como um array de 50 elementos, cada um dos quais é uma variável do tipo **double**[1].

3.1.2 Acesso a Elementos de um Array

Os elementos de um array são **acessados** por meio de **índices** que indicam a posição de cada elemento em relação ao elemento inicial do array. Como a **indexação** dos elementos começa com `0`, o último elemento de um array possui índice igual ao número de elementos do array menos um. Esquematicamente, a notação usada para acesso aos elementos de um array é:

nome-do-array [índice]

Para todos os efeitos, um elemento de um array pode ser considerado uma variável do tipo usado na definição do array, de modo que qualquer operação permitida sobre uma variável do tipo de um elemento de um array também é permitida para o próprio elemento.

[1] A razão pela qual este e muitos outros textos sobre C usam colchetes quando fazem referência a um array (como em `notas[]`, por exemplo) é que o nome de um array considerado isoladamente representa seu endereço (v. **Seção 3.2**).

Compiladores de C não fazem verificação de acesso além dos limites de um array, de modo que o programador pode acidentalmente acessar porções de memória que não foram alocadas para o array e isso pode trazer consequências imprevisíveis.

3.1.3 Iniciações de Arrays

Como ocorre com outros tipos de variáveis, arrays de duração fixa e arrays de duração automática diferem em termos de iniciação. Como padrão, arrays com duração fixa têm todos os seus elementos iniciados automaticamente com zero, mas podem-se iniciar todos ou alguns elementos explicitamente com outros valores. A iniciação de elementos de um array é feita por meio do uso de expressões constantes, separadas por vírgulas e entre chaves, seguindo a definição do array. Uma construção dessa natureza é denominada um **iniciador**. Considere, por exemplo:

```
static  int  meuArray1[5];
static  int  meuArray2[5] = {1, 2, 3.14, 4, 5};
```

Nesse exemplo, todos os elementos de `meuArray1` serão iniciados (implicitamente) com `0`, enquanto `meuArray2[0]` recebe o valor `1`, `meuArray2[1]` recebe `2`, `meuArray2[2]` recebe `3`, `meuArray2[3]` recebe `4` e `meuArray2[4]` recebe `5`.

Não é ilegal incluir numa iniciação um número de valores maior do que o permitido pelo tamanho do array, mas, nesse caso apenas o número de valores igual ao tamanho do array é usado na iniciação. Também não é necessário iniciar todos os elementos de um array e, se houver um número de valores de iniciação menor do que o número de elementos do array, os elementos remanescentes serão iniciados implicitamente com zero independentemente da duração do array.

Quando todos os elementos de um array são iniciados, o tamanho do array pode ser omitido, pois, nesse caso, o compilador deduz o tamanho do array baseado no número de valores de iniciação. Por exemplo:

```
static  int  meuArray2[] = {1, 2, 3.14, 4, 5};
```

é o mesmo que:

```
static  int  meuArray2[5] = {1, 2, 3.14, 4, 5};
```

Essa última característica é válida também para arrays de duração automática e, na prática, é mais usada com strings (v. **Seção 3.7**).

Arrays de duração automática também podem ser iniciados explicitamente. As regras para iniciação explícita de elementos de um array de duração automática são similares àquelas usadas para arrays de duração fixa. Entretanto, arrays de duração automática não são iniciados implicitamente. Ou seja, se não houver nenhuma iniciação explícita, o valor de cada elemento será indefinido; i.e., eles receberão o conteúdo indeterminado encontrado nas posições de memória alocadas para o array.

3.1.4 Tamanho de Array

O tamanho, em bytes, de um array pode ser determinado aplicando-se o operador **sizeof** (v. **Seção 1.16.3**) ao nome do array. Entretanto, nesse caso, não se deve utilizar nenhum índice; caso contrário, o tamanho resultante será o de um único elemento do array (ao invés do tamanho de todo o array). O trecho de programa a seguir demonstra esses fatos.

```
int    ar[] = {-1, 2, -2, 7, -5};
size_t tamanhoArray, tamanhoElemento, nElementos;

tamanhoArray = sizeof(ar);

tamanhoElemento = sizeof(ar[0]);

nElementos = tamanhoArray/tamanhoElemento;
```

A expressão:

```
sizeof(ar[0])
```

resulta no tamanho do elemento de qualquer array `ar[]` porque qualquer array tem pelo menos um elemento, que é o elemento `ar[0]`. Portanto pode-se concluir que a expressão:

```
sizeof(ar)/sizeof(ar[0])
```

resulta sempre no número de elementos de um array `ar[]`, desde que ela seja avaliada dentro do escopo no qual o array é definido (v. **Seção 2.4**).

Conforme foi afirmado na **Seção 1.6**, o resultado do operador **sizeof** é do tipo **size_t**, que é um tipo inteiro sem sinal. Portanto esse tipo não deve ser misturado com tipos inteiros com sinal, a não ser em expressões de atribuição.

3.2 Aritmética de Ponteiros

A linguagem C permite que as operações aritméticas apresentadas na **Tabela 3–1** sejam executadas com ponteiros (supondo que `p`, `p1` e `p2` sejam ponteiros de quaisquer tipos).

Operação	Exemplo
Soma de um valor inteiro a um ponteiro	`p + 2`
Subtração de um valor inteiro de um ponteiro	`p - 3`
Incremento de ponteiro	`++p` ou `p++`
Decremento de ponteiro	`--p` ou `p--`
Subtração entre dois ponteiros do mesmo tipo	`p1 - p2`

Tabela 3–1: Operações Aritméticas sobre Ponteiros

Operações aritméticas sobre ponteiros, entretanto, devem ser interpretadas de modo diferente das operações aritméticas usuais. Por exemplo, se `p` é um ponteiro definido como:

```
int  *p;
```

a expressão:

```
p + 3
```

deve ser interpretada como o endereço do espaço em memória que está três variáveis do tipo **int** adiante do endereço da variável para o qual `p` aponta. Isto é, como `p` é um endereço, `p + 3` também será um endereço. Mas, em vez de adicionar `3` ao valor do endereço armazenado em `p`, o compilador adiciona `3` multiplicado pelo tamanho (i.e., número de bytes) da variável para a qual `p` aponta. Nesse contexto, o tamanho da variável

apontada pelo ponteiro é denominado **fator de escala**. Com exceção da última operação apresentada na **Tabela 3–1**, as demais operações sobre ponteiros envolvem a aplicação de um fator de escala, que corresponde à largura do tipo de variável para o qual o ponteiro aponta.

Suponha, por exemplo, que o endereço correntemente armazenado no ponteiro `p` definido acima seja `e` que variáveis do tipo **int** sejam armazenadas em 4 bytes. Então, `p + 3` significa, após a aplicação do fator de escala, o endereço `e + 3*4`, que é igual ao endereço `e + 12`. A **Figura 3–1** ilustra esse argumento.

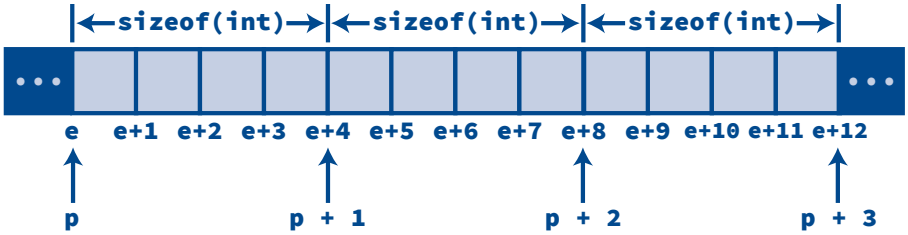

FIGURA 3–1: SOMA DE UM INTEIRO A UM PONTEIRO

Se, no exemplo anterior, o ponteiro `p` tivesse sido definido como `char *p`, então `p + 3` significaria `e + 3` [porque `sizeof(char)` é sempre `1`]. Concluindo, `p + 3` sempre significa o endereço da terceira variável do tipo apontado pelo ponteiro `p` após aquela correntemente apontada por ele.

Subtrair um inteiro de um ponteiro tem uma interpretação semelhante. Por exemplo, `p - 3` representa o endereço da terceira variável do tipo apontado pelo ponteiro `p` que precede a variável correntemente apontada por ele.

Operações de incremento e decremento de ponteiros também são bastante comuns em programação em C. Nesses casos, os operadores de incremento e decremento são utilizados para fazer um ponteiro apontar para a variável posterior e anterior, respectivamente, à posição atual do ponteiro. Em qualquer caso, o fator de escala mencionado é aplicado à operação.

A subtração de dois ponteiros é legal, desde que os ponteiros sejam do mesmo tipo, mas só faz sentido quando eles apontam para elementos de um mesmo array. Essa operação resulta num número inteiro cujo valor absoluto representa o número de elementos do array que se encontram entre os dois ponteiros.

3.3 Relações entre Ponteiros e Arrays

Em geral, se `p` aponta para o início de um array `ar[]`, então a seguinte relação é válida para qualquer inteiro `i`:

`*(p + i)` *é o mesmo que* `ar[i]`

Agora, em C, o nome de um array considerado isoladamente é interpretado como o endereço do array (i.e., o endereço do elemento de índice `0`). Ou seja,

`ar` *é o mesmo que* `&ar[0]`

Combinando-se as duas relações anteriores, obtém-se a seguinte equivalência:

`*(ar + i)` *é o mesmo que* `ar[i]`

Em consequência dessa última relação, quando um compilador de C encontra uma referência com índice a um elemento de um array (p. ex., `ar[2]`), ele adiciona o índice ao endereço do array (p. ex., `ar + 2`) e aplica o operador de indireção a esse endereço [p. ex., `*(ar + 2)`], obtendo, assim, o valor do elemento do array.

Devido às relações apresentadas, ponteiros e nomes de arrays podem ser utilizados de modo equivalente tanto com o operador de indireção, representado por `*`, quanto com colchetes, que também representam um operador,

como será visto na **Seção 3.11**. É importante lembrar, entretanto, que conteúdos de ponteiros podem ser modificados, enquanto o valor atribuído a um nome de array não pode ser modificado porque ele representa o endereço do array (variável) e o endereço de uma variável não pode ser modificado. Em termos práticos, isso significa, por exemplo, que o nome de um array (sem índice) não pode sofrer o efeito de nenhum operador com efeito colateral.

Com o conhecimento adquirido nesta seção, pode-se responder uma questão que intriga muitos iniciantes na linguagem C, que é: Por que a indexação de arrays começa em 0 e não em 1, como seria mais natural? A resposta a essa pergunta é que, em geral, para acessar o elemento de índice `i` do array seria necessária uma operação aritmética a mais (i.e., subtrair `1` de `p + i`) do que seria o caso se `p` apontasse para o elemento de índice `0`. Ocorreria o mesmo se a indexação de arrays começasse em `1`: a cada acesso a um elemento de um array seria necessário subtrair `1`. Portanto, do ponto de vista de eficiência, a indexação de arrays em C é justificável.

3.4 O Qualificador const

A palavra-chave **const**, quando aplicada na definição de uma variável, especifica que a variável não pode ter seu conteúdo alterado após ser iniciada. Por exemplo, após a iniciação:

```
const int varConstante = 0;
```

não seria mais permitido à variável `varConstante` ter seu conteúdo modificado. Mas, não se pode garantir que uma variável constante não seja modificada indiretamente. Por exemplo, considerando a definição de `ptr1` apresentada acima, é permitido que a variável `varConstante` seja modificada por meio de uma instrução como:

```
*ptr1 = 1;
```

Numa definição de ponteiro, a palavra-chave **const** pode aparecer precedida pelo símbolo `*` ou não. Nos dois casos, os significados das definições são diferentes. Por exemplo, na segunda definição a seguir:

```
int          x;
int  *const  ponteiroConstante = &x;
```

a variável `ponteiroConstante` é considerada um ponteiro que deve apontar sempre para a variável `x` (i.e., o valor do ponteiro não deve mudar). Mas, esse ponteiro pode ser utilizado para alterar o conteúdo da variável para a qual ele aponta. Esse uso de **const** é semelhante àquele apresentado antes; afinal, todo ponteiro é uma variável.

Considere, agora, outro tipo de uso de **const** que aparece na segunda definição de variável a seguir:

```
int         x;
int  const *ponteiroParaConstante = &x;
```

Essa última definição é equivalente a:

```
const int *ponteiroParaConstante = &x;
```

Em ambos os formatos, a variável `ponteiroParaConstante` é considerada como um ponteiro para uma variável que não pode ser modificada (indiretamente) por meio desse ponteiro. Mas, obviamente, pode ser modificada diretamente por meio da própria variável. Nesse caso, o valor do ponteiro em si pode ser modificado, de modo que ele possa apontar para outra variável.

A palavra-chave **const** também pode ser usada para qualificar o conteúdo (i.e., os elementos) de um array como, por exemplo:

```
const int ar[] = {1, 2, 3, 4};
```

ou:

```
int const ar[] = {1, 2, 3, 4};
```

As duas últimas definições têm exatamente o mesmo significado e indicam que os elementos do array `ar[]` devem ser considerados constantes.

O principal propósito de **const** é assegurar que dados que não devem ser modificados não serão realmente modificados. Em particular, o uso de **const** é bastante útil quando ponteiros são passados para funções, pois a declaração de um ponteiro utilizado como parâmetro com a palavra **const** garante que o valor apontado pelo ponteiro não será modificado pela função.

3.5 Uso de Arrays com Funções

3.5.1 Declarando Arrays como Parâmetros Formais

Na definição de uma função, um parâmetro formal que representa um array é declarado como um ponteiro para o elemento inicial do array. Existem duas formas alternativas de declarar tal parâmetro:

*tipo-do-elemento *parâmetro*

ou:

tipo-do-elemento parâmetro[]

Como exemplo de parâmetro formal que representa um array considere o seguinte cabeçalho de função:

```
void  MinhaFuncao(double *ar)
```

ou, alternativamente:

```
void  MinhaFuncao(double ar[])
```

Quando o segundo formato de declaração é utilizado, o compilador converte-o no primeiro formato. Assim os dois tipos de declarações são completamente equivalentes. Entretanto, em termos de legibilidade, a segunda declaração é melhor do que a primeira, pois enfatiza que o parâmetro será tratado como um array. Na primeira declaração, não existe, em princípio, uma maneira de se saber se o parâmetro é um ponteiro para uma única variável do tipo **double** ou para um array de elementos do tipo **double**.

A escolha entre declarar um parâmetro de função em forma de array ou de ponteiro não tem nenhum efeito na tradução feita pelo compilador. Para o compilador, o parâmetro `ar` do exemplo anterior é apenas um ponteiro para um espaço em memória contendo um valor **double** e não exatamente um array. Mas, em virtude da relação entre ponteiros e arrays (v. **Seção 3.3**), ainda é possível acessar os elementos do parâmetro `ar` usando colchetes.

Deve-se ressaltar que, em consequência do modo como arrays são tratados numa função, não é possível determinar o tamanho de um array por meio da aplicação do operador **sizeof** sobre um parâmetro formal que representa o array. Devido à impossibilidade de uma função determinar o tamanho de um array cujo endereço é recebido como parâmetro, é muitas vezes necessário incluir o tamanho do array na lista de parâmetros da função. Isso permite à função saber onde o array termina.

3.5.2 Arrays como Parâmetros Reais

Numa chamada de função que possui um parâmetro que representa um array, utiliza-se como parâmetro real o nome de um array compatível com o parâmetro formal correspondente. Esse nome será interpretado como o endereço do array, como mostra o seguinte programa.

```
#include <stdio.h>

void ExibeArrayInts(const int ar[], int tamanho)
{
   int i;

      /* Se o número de elementos for menor do que ou */
      /* igual a zero, escreve as chaves e retorna    */
   if (tamanho <= 0) {
      printf("\n\t{ }\n");
      return;
   }

   printf("\n\t{ "); /* Escreve abre-chaves */

      /* Escreve do primeiro ao penúltimo elemento */
   for (i = 0; i < tamanho - 1; ++i)
      printf("%d, ", ar[i]);

      /* Escreve o último elemento separadamente para */
      /* que não haja uma vírgula sobrando ao final   */
   printf("%d }\n",  ar[tamanho - 1]);
}

int main(void)
{
   int array[] = {5, -1, 0, 4, 5, -8, 9};

   printf("\n\t   ****** Array ******\n");
   ExibeArrayInts(array, sizeof(array)/sizeof(array[0]));

   return 0;
}
```

O programa acima possui uma função que recebe como parâmetro um array de elementos do tipo **int**. Observe o uso de **const** nessa função. Nesse caso, o array é um parâmetro de entrada e o uso de **const** garante que ele não é acidentalmente modificado pela função.

A função `ExibeArrayInts()` é chamada como:

```
ExibeArrayInts(array, sizeof(array)/sizeof(array[0]));
```

Note que o primeiro parâmetro na chamada de `ExibeArrayInts()` no corpo da função **main()** é `array`, que é o nome do array definido em **main()**. O segundo parâmetro na chamada de `ExibeArrayInts()` resulta no tamanho do array `array[]`, conforme visto na **Seção 3.1**.

3.5.3 Retorno de Arrays e Zumbis

Do mesmo modo que uma função nunca recebe um array completo como parâmetro, também não é permitido a uma função retornar um array completo. Mas é permitido a uma função retornar o endereço de um array (ou de qualquer outro tipo de variável). Todavia, o programador deve tomar cuidado para não retornar um endereço de um array de duração automática (i.e., definido no corpo da função sem o uso de **static**), pois tal

array é liberado quando a função retorna. Aliás, esse conselho não se aplica apenas no caso de arrays; ele é mais abrangente e deve ser sempre seguido.

É importante salientar ainda que parâmetros são tratados como variáveis de duração automática no sentido de que eles são alocados quando a função que os usa é chamada e são liberados quando a mesma função é encerrada. Assim retornar o endereço de um parâmetro também produz efeito zumbi. Portanto nunca retorne o endereço de um parâmetro.

3.5.4 Qualificação de Parâmetros com const

O uso mais pragmático e efetivo de **const** é a qualificação de parâmetros formais e reais, notadamente quando eles representam estruturas (v. **Seção 3.10**) e arrays.

Com relação à qualificação com **const** de parâmetros reais e formais, a seguinte recomendação deve ser seguida: se uma variável for qualificada com **const**, seu endereço só deverá ser passado como parâmetro real se o respectivo parâmetro formal for um ponteiro para o tipo da variável e for qualificado com **const**. Se uma variável não for qualificada com **const**, essa restrição não se aplica. Além disso, se o conteúdo de um array for qualificado com **const**, seu endereço só deverá ser passado como parâmetro real se o respectivo parâmetro formal for um ponteiro para o tipo do elemento do array e for qualificado com **const**.

3.6 Arrays Multidimensionais

Array multidimensional é um array cujos elementos também são arrays. Os arrays vistos até aqui são, em contrapartida, denominados **unidimensionais**. Um **array bidimensional** é aquele cujos elementos são arrays unidimensionais, um **array tridimensional** é aquele cujos elementos são arrays bidimensionais e assim por diante. Embora o padrão ISO determine que um compilador de C deve suportar pelo menos seis dimensões para arrays multidimensionais, raramente, mais de três ou quatro dimensões são utilizadas em aplicações práticas.

Um array multidimensional é definido com pares consecutivos de colchetes, cada um dos quais contendo o tamanho de cada dimensão:

$$\textit{tipo-do-elemento nome-do-array } [\textit{tamanho}_1][\textit{tamanho}_2]...[\textit{tamanho}_N]$$

No exemplo abaixo, um array tridimensional de caracteres é definido:

```
char arrayDeCaracteres[3][4][5];
```

A variável `arrayDeCaracteres` desse exemplo é interpretada como um array com três elementos, sendo cada um dos quais um array com quatro elementos, cada um dos quais é um array com cinco elementos do tipo **char**.

3.6.1 Iniciações de Arrays Multidimensionais

Para iniciar um array multidimensional, devem-se colocar os valores dos elementos de cada dimensão do array entre chaves. Se, para uma dada dimensão, houver um número de valores menor do que o número especificado na definição do array, os elementos remanescentes receberão zero como valor. A seguir, é apresentado um exemplo de iniciação de um array bidimensional:

```
int  arBi[5][3] = { {1, 2, 3},
                    {4},
                    {5, 6, 7}
                  };
```

Frequentemente, a primeira e a segunda dimensões de um array bidimensional são denominadas, respectivamente, **linha** e **coluna**. Assim, no último exemplo, `arBi[]` é definido como um array com 5 linhas e 3 colunas, mas apenas suas três primeiras linhas são iniciadas explicitamente e apenas o primeiro elemento da sua segunda linha é iniciado explicitamente.

Observe como a endentação no exemplo acima torna a iniciação mais legível. Em forma de tabela, esse array poderia ser visualizado como:

1	2	3
4	0	0
5	6	7
0	0	0
0	0	0

Se as chaves internas não tivessem sido incluídas na iniciação do array, como no exemplo a seguir:

```
int  arBi[5][3] = { 1, 2, 3,
                    4,
                    5, 6, 7 };
```

o resultado da iniciação seria visualizado como:

1	2	3
4	5	6
7	0	0
0	0	0
0	0	0

Se a especificação de tamanho da primeira dimensão de um array multidimensional for omitida, o compilador automaticamente deduz o tamanho dela baseado no número de valores de iniciação presentes. Neste caso, os especificadores de tamanho das outras dimensões devem estar presentes. Por exemplo,

```
int  arA[][3][2] = { { {1, 2}, {0, 0}, {1, 1} },
                     { {0, 1}, {1, 0}, {0, 0} } };
```

resulta num array *2×3×2*, pois existem *12* valores e cada elemento do array `arA[]` é um array *3×2*. Por outro lado, a definição a seguir:

```
int  arB[][] = { 1, 2, 3, 4, 5, 6 };   /* ILEGAL */
```

é ilegal porque o compilador não consegue determinar os tamanhos das dimensões do array (i.e., ele não seria capaz de decidir se o array é *2×3* ou *3×2*). Entretanto, se o tamanho da segunda dimensão for especificado, a definição deixa de ser ambígua e torna-se legal.

3.6.2 Acesso a Elementos de Arrays Multidimensionais

Para acessar um elemento de um array multidimensional, utilizam-se tantos índices quanto for'em as dimensões do array e cada índice deve ser envolvido por um par de colchetes. Por exemplo:

```
int  ar[2][3][4];
int  x;
x = ar[1][1][2];
```

3.6.3 Arrays Multidimensionais como Parâmetros de Funções

Para declarar um array multidimensional como parâmetro formal de uma função, devem-se especificar os tamanhos de todas as dimensões, exceto o tamanho da primeira dimensão, como mostra a função `ExibeArrayBi()` a seguir.

```
void ExibeArrayBi(int a[][2], int dim1, int dim2)
{
   int i, j;

   putchar('\n'); /* Embelezamento */

      /* Escreve o índice de cada coluna */
   for (i = 0; i < dim2; ++i)
      printf("\t%d", i);

   putchar('\n'); /* Embelezamento */

   for (i = 0; i < dim1; ++i) {
      printf("\n%d", i); /* Escreve o índice de cada linha */

         /* Escreve o valor de cada elemento */
      for (j = 0; j < dim2; ++j)
         printf("\t%d", a[i][j]);
   }

   putchar('\n'); /* Embelezamento */
}
```

Para passar um array multidimensional como parâmetro real para uma função, deve-se proceder da mesma forma que com arrays unidimensionais. Isto é, apenas o nome do array deve ser utilizado na chamada.

3.7 Strings e Caracteres

3.7.1 Strings

Em C, um **string** é um array de elementos do tipo **char** terminado pelo **caractere nulo**, representado pela sequência de escape `'\0'`. Em qualquer código de caracteres usado numa implementação de C, o inteiro associado a esse caractere é zero.

Pode-se armazenar um string em memória utilizando-se um array de elementos do tipo **char** iniciado como mostra o seguinte exemplo:

```
char ar1[] = "bola";
```

Devido à onipresença do caractere terminal `'\0'`, quando não é especificado explicitamente, o tamanho de um array iniciado com um string é sempre um a mais do que o número de caracteres visíveis no string. Assim o array `ar1[]` do exemplo acima possui tamanho igual a `5`.

Apesar da aparência, os caracteres entre aspas que aparecem numa iniciação de um array de caracteres não constituem um string constante. Isto é, na **Seção 1.2**, um string constante foi definido como uma sequência de caracteres entre aspas, mas essa definição não vale nesse caso específico de iniciação de array. Quer dizer, nesse caso, caracteres entre aspas constituem uma facilidade oferecida pela linguagem C para isentar os programadores de ter que escrever iniciações de arrays de caracteres do modo convencional (v. **Seção 3.1**), que é bem mais trabalhoso.

Quando o número de elementos do array é especificado e é menor do que ou igual ao número de caracteres presentes na iniciação, o array não conterá um string. Por exemplo, na seguinte iniciação, o array `ar2[]` receberá apenas os caracteres: `'b'`, `'o'`, `'l'` e `'a'` e, portanto, não conterá um string em virtude da ausência do caractere terminal `'\0'`.

```
char ar2[4] = "bola";
```

Quando o número de elementos do array é especificado e é maior do que o número de caracteres presentes na iniciação, os elementos remanescentes no array, se for o caso, serão iniciados com zero.

Um **string constante** é uma sequência de caracteres entre aspas, desde que tal construção não apareça na iniciação de um array de caracteres, conforme foi visto acima. Um string constante é representado pelo endereço de seu primeiro caractere em memória. Portanto o tipo de um string constante é **char** * e pode-se iniciar um ponteiro para **char** com um string constante, como, por exemplo:

```
char *ptr = "Isto e' um string constante."
```

Essa definição de variável é diferente de uma definição que utiliza array. Por exemplo, uma diferença entre essa última definição e a definição:

```
char str[] = "Isto NAO e' um string constante."
```

é que, no primeiro caso, além do espaço reservado para conter o string, também é alocado espaço para conter o ponteiro `ptr`. Além disso, apesar de `ptr` e `str` apontarem para o elemento inicial do string, o valor da variável `ptr` pode ser modificado, enquanto o endereço `str` não pode. Entretanto, se o valor de `ptr` for modificado, o endereço com o qual esse ponteiro foi iniciado será perdido (i.e., o string para o qual `ptr` estava apontando não poderá mais ser acessado).

Uma importante diferença entre as iniciações do array `str[]` e do ponteiro `ptr` acima é o fato de strings constantes poderem ser armazenados numa região de memória cujo conteúdo não pode ser modificado. Em tal situação, qualquer tentativa de modificar o string para o qual o ponteiro `ptr` aponta gera um erro de execução (i.e., aborto) do programa. Portanto strings constantes devem ter seus conteúdos considerados realmente constantes.

3.7.2 Funções de Biblioteca para Processamento de Strings

A biblioteca padrão de C possui várias funções para processamento de strings declaradas no cabeçalho `<string.h>`. Por outro lado, leitura e escrita de strings utilizam funções declaradas em `<stdio.h>`. A **Tabela 3–2** apresenta as principais funções da biblioteca padrão de C para processamento de strings.

Protótipo da função	O que ela realiza
`char *fgets(char *ar, size_t n, FILE *stream)`	*Lê um string no meio de entrada representado pelo terceiro parâmetro; se a leitura for via teclado, esse parâmetro deve ser* **stdin**.
`int puts(const char *s)`	*Escreve o string* `s` *na tela*
`size_t strlen(const char *s)`	*Retorna o comprimento do string* `s`
`char *strcpy(char *ar, const char *s)`	*Copia o string* `s` *para o array* `ar`
`char *strcat(char *ar, const char *s)`	*Copia o string* `s` *ao final do string armazenado no array* `ar`

Tabela 3–2: Funções da Biblioteca Padrão de C para Processamento de Strings

Protótipo da função	O que ela realiza
`int strcmp(const char *s1, const char *s2)`	*Compara os strings* `s1` *e* `s2` *e retorna* `0` *se os strings são iguais, um valor negativo se* `s1` *preceder* `s2` *e um valor positivo se* `s1` *suceder* `s2`
`int strcoll(const char *s1, const char *s2)`	*Semelhante à* **strcmp**(), *mas leva em consideração informações sobre localidade*
`char *strstr(const char *s1, const char *s2)`	*Retorna o endereço da primeira ocorrência do string* `s2` *no string* `s1`*; se não houver nenhuma ocorrência de* `s2` *em* `s1`*, retorna* **NULL**
`char *strchr(const char *s, int c)`	*Retorna o endereço da primeira ocorrência do caractere* `c` *no string* `s`*; se não houver nenhuma ocorrência de* `c` *em* `s`*, retorna* **NULL**
`char *strrchr(const char *s, int c)`	*Retorna o endereço da última ocorrência do caractere* `c` *no string* `s`*; se não houver nenhuma ocorrência de* `c` *em* `s`*, retorna* **NULL**
`char *strtok(char *s, const char *sep)`	*Separa o string* `s` *em tokens; os separadores de tokens são os caracteres presentes no string* `sep`

Tabela 3–2: Funções da Biblioteca Padrão de C para Processamento de Strings

3.7.3 Conversão de Strings Numéricos em Números

A função **strtol**() faz parte do módulo stdlib da biblioteca padrão de C e converte um string num valor do tipo **long int**, desde que ele permita tal conversão. O protótipo dessa função é:

```
long strtol(const char *str, char **final, int base)
```

Nesse protótipo, `str` é o string que se deseja converter em inteiro e o retorno é o valor resultante da conversão. O parâmetro `base` corresponde à base do número a ser convertido, ao passo que `final` é o endereço de um ponteiro que apontará para o caractere do string que encerra a conversão. Se todos os caracteres do string forem usados na conversão, esse ponteiro apontará para o caractere terminal do string. Se o string não puder ser convertido, o retorno é zero. Se o parâmetro `final` não for **NULL**, ao final da conversão ele estará apontando para a porção do string que não foi convertida. Se esse parâmetro for **NULL** e a função retornar `0`, esse valor será ambíguo, pois não se tem como saber se ele é o valor resultante da conversão ou se não houve conversão.

A função **strtod**(), declarada em `<stdlib.h>`, converte strings em números reais do tipo **double** e seu protótipo é:

```
double strtod(const char *str, char **final)
```

Nesse protótipo, `str` é string a ser convertido e `final` tem interpretação idêntica ao parâmetro de mesmo nome da função **strtol**(). Essa função retorna o valor resultante da conversão, se ela for possível ou zero, se nenhuma conversão for possível. Um string válido como primeiro parâmetro de **strtod**() pode incluir: espaços em branco em seu início, sinal (i.e., `+` ou `-`), ponto decimal, `e` ou `E` (indicando o uso de notação científica). Se nenhuma conversão for possível, essa função retorna `0.0`, que obviamente, é um valor válido do tipo **double**. Assim, para verificar se ocorreu erro de conversão, deve-se checar o parâmetro `final`. Isto é, se, ao retorno da função, esse parâmetro estiver apontando para o início do string original, pode-se concluir que não houve nenhuma conversão. Por outro lado, se, ao término da conversão, o parâmetro `final` apontar para o caractere terminal do string, todos os caracteres do string foram usados na conversão.

3.7.4 Classificação de Caracteres: Funções isX()

No cabeçalho `<ctype.h>`, são declaradas funções que classificam caracteres de acordo com diversas categorias, tais como letras, dígitos e espaços em branco. Todas essas **funções de classificação de caracteres** têm nomes começando com is e recebem um parâmetro do tipo **int** representando o caractere que será classificado. Cada uma delas verifica se o caractere recebido como parâmetro satisfaz uma determinada propriedade e retorna um valor diferente de zero, se esse for o caso ou zero, se o caractere não satisfaz a propriedade a que se refere a função. As funções de classificação de caracteres mais comumente utilizadas são brevemente descritas na **Tabela 3–3**.

Função	Retorna um valor diferente de zero se o parâmetro representar...
isalnum()	um caractere alfanumérico (i.e., dígito ou letra)
isalpha()	uma letra
isblank()	`' '` ou `'\t'`
isdigit()	um dígito
islower()	uma letra minúscula
ispunct()	um símbolo de pontuação
isspace()	um espaço em branco qualquer, incluindo quebra de linha (`'\n'`)
isupper()	uma letra maiúscula

Tabela 3–3: Funções de Classificação de Caracteres mais Comuns

3.7.5 Transformação de Caracteres: tolower() e toupper()

Além das funções de classificação de caracteres apresentadas na **Tabela 3–3**, o módulo ctype também provê duas funções de **transformação de caracteres** (assim denominadas pelo padrão ISO de C). A função **tolower()** retorna a letra minúscula correspondente a um dado caractere e a função **toupper()** retorna a letra maiúscula correspondente a um dado caractere. Se o único parâmetro que cada uma dessas funções recebe não for letra, o retorno é o próprio caractere recebido como parâmetro.

3.8 Função main() com Parâmetros

A função **main()** é uma função com características especiais. Sua presença num programa em C é obrigatória em programas hospedados; i.e., programas que são executados sob intermediação de um sistema operacional. Essa função é sempre a primeira função a ser executada no programa e, quando ela retorna, o programa é encerrado.

A função **main()** possui dois protótipos, sendo que o mais conhecido deles é:

```
int main(void)
```

A função **main()** pode receber dois parâmetros do sistema operacional no qual o programa é executado. Esses parâmetros estão associados a strings que acompanham a invocação do programa via console e são denominados argumentos de linha de comando. O protótipo da função **main()** que incorpora esses dois parâmetros é:

```
int main(int argc, char *argv[])
```

O primeiro parâmetro recebido pela função **main()** quando o programa que a contém é executado é tradicionalmente denominado *argc* e representa o número de **argumentos** presentes na **linha de comando** do sistema operacional quando esse programa é invocado. O segundo parâmetro fornecido pelo sistema operacional,

tradicionalmente denominado *argv*, consiste em um array de strings que armazena os argumentos presentes na linha de comando quando o programa é invocado.

Os argumentos passados para um programa incluem seu nome e cada argumento (string) que constitui o comando de execução do programa deve ser separado de outro por meio de um ou mais espaços em branco:

***nome-do-programa argumento*$_1$ *argumento*$_2$ *... argumento*$_n$**

Quando a execução de um programa é iniciada, seu nome é armazenado como primeiro elemento no array `argv[]` e os demais strings presentes na linha de comando serão armazenados consecutivamente nesse array. Ao parâmetro `argc` será automaticamente atribuído o número de elementos do array `argv[]`. Por exemplo, se houver três argumentos na linha de comando, além do nome do programa, `argv[]` terá quatro elementos e `argc` assumirá `4` como valor.

É importante salientar que, nos dois protótipos da função **main()** aceitos pelo padrão ISO de C, o tipo de retorno é **int**. Isso quer dizer que, apesar de alguns compiladores permitirem que **main()** seja definida com tipo de retorno **void**, usar este tipo de retorno não é portável.

3.9 Tipos Definidos pelo Programador

A linguagem C permite que o programador crie seus próprios tipos de dados com o uso da palavra-chave **typedef**, que tem a seguinte sintaxe:

typedef *tipo nome-do-tipo;*

A sintaxe de uma definição de tipo é semelhante àquela utilizada na definição de variáveis. Entretanto, ao contrário de uma declaração de variável, uma definição de tipo não causa a alocação de nenhum espaço em memória, ela apenas assegura que nome-do-tipo é um sinônimo de tipo. Por exemplo, a declaração de tipo abaixo:

```
typedef char tCPF[11];
```

define um tipo de dado, denominado `tCPF`, que representa variáveis que são arrays com 11 elementos do tipo **char**. Em consequência dessa definição, a declaração de variável a seguir:

```
tCPF meuCPF;
```

é idêntica a:

```
char meuCPF[11];
```

A notação adotada neste livro para identificadores de tipos preconiza que eles comecem com *t* (p. ex., `tCPF`), mas alguns programadores preferem que, em vez disso, esses identificadores sejam terminados por *_t* (p. ex., `CPF_t`).

Definições de tipos são usadas frequentemente não apenas para identificar tipos estruturados construídos pelo programador, como também para atribuir novos nomes a tipos primitivos. Nesse último caso, a finalidade é primar pela legibilidade e, principalmente, pela portabilidade de programas.

3.10 Estruturas e Uniões

Estruturas são variáveis estruturadas semelhantes a arrays que diferem destes por permitirem que seus elementos, denominados **campos** ou **membros**, sejam de tipos diferentes. Por causa dessa característica, estruturas constituem variáveis **heterogêneas**.

Uma estrutura serve para conter dados de tipos diferentes relacionados entre si. Cada membro de uma estrutura deve possuir um nome, que segue as regras de construção de identificadores de C.

3.10.1 Definições de Estruturas

Existem várias formas permitidas para a definição de uma estrutura em C, mas o uso de **typedef** constitui a melhor elas. Usando **typedef**, antes de definir uma estrutura (variável), define-se seu tipo como:

```
typedef struct rótulo-da-estrutura {
            tipo1  campo1;
            tipo2  campo2;
            ...
            tipoN campoN;
        } nome-do-tipo;
```

Aqui, define-se um tipo que é sinônimo daquilo que precede sua definição (v. **Seção 3.9**), incluindo a palavra **struct**. O rótulo da estrutura é um identificador opcional que é necessário apenas quando se declara uma estrutura com autorreferência, como será visto na **Seção 10.2.2**, ou um tipo de estrutura incompleto, como se verá na **Seção 5.4**.

Mais de um tipo pode ser definido simultaneamente com um único uso de **typedef**, como, por exemplo:

```
typedef  struct   {
            char nome[30];
            int  dia, mes, ano;
         } tRegistro, *tPtrParaRegistro;

tRegistro         registroDaPessoa;
tPtrParaRegistro  ptrParaRegistro;
```

Dois campos que fazem parte de estruturas diferentes podem possuir o mesmo nome, pois, devido à forma como os campos de uma estrutura são acessados, não existe chance de colisão dos identificadores de campos nesse exemplo.

3.10.2 Estrutura com Autorreferência

Uma estrutura não pode conter um campo que seja do tipo da própria estrutura, mas é permitido que um campo de uma estrutura seja um ponteiro para ela mesma. Por exemplo:

```
typedef struct E {
            int       a, b;
            struct E *ptr;
        } tEstruturaAutoRef;
```

Nesse exemplo, o campo `ptr` de estruturas do tipo `tEstruturaAutoRef` é um ponteiro para estruturas desse mesmo tipo.

Estruturas como aquelas do tipo `tEstruturaAutoRef` são denominadas estruturas com autorreferência e são bastante úteis para a criação de listas encadeadas (v. **Seção 10.2.2**) bem como outras construções mais complexas usadas em programação.

Autorreferências não são permitidas com o uso de definições de tipo sem rótulos (v. **Seção 3.10**). Por exemplo, a seguinte tentativa de autorreferência não é permitida:

```
typedef struct  {
           int  a, b;
           tE  *ptrParaE; /* ILEGAL */
        } tE;
```

Nesse exemplo, o compilador considera ilegal a declaração do campo **ptrParaE**, porque ele julga que o tipo **tE** é desconhecido no local onde esse campo é declarado.

3.10.3 Iniciações de Estruturas

Uma estrutura pode ser iniciada de modo similar a um array. Isto é, uma estrutura a ser iniciada deve ser seguida pelo sinal de igualdade e de uma lista de valores entre chaves. O número de valores de iniciação não deve exceder o número de campos da estrutura e cada um deles deve ser compatível com o respectivo campo. Por exemplo, considerando a definição do tipo **tRegistro**:

```
typedef  struct {
            char nome[30];
            int  dia, mes, ano;
         } tRegistro;
```

a variável **registroDaPessoa** poderia ser iniciada como:

```
tRegistro registroDaPessoa = {"Jose da Silva", 12, 10, 1960};
```

3.10.4 Atribuições entre Estruturas

Uma estrutura pode ser atribuída a outra, desde que ambas sejam do mesmo tipo. Por exemplo, considerando o tipo **tRegistro** apresentado acima e a seguinte definição de variáveis:

```
tRegistro e1, e2, *ptr;
```

as seguintes atribuições são perfeitamente válidas:

```
e1 = e2;
e2 = e1;
ptr = &e1;
e2 = *ptr;
```

3.10.5 Acesso a Campos de Estruturas

Existem duas formas de acesso aos campos de uma estrutura, dependendo do fato de se estar lidando com uma estrutura (variável) ou com um ponteiro para uma estrutura:

- Caso uma estrutura esteja sendo empregada, utiliza-se o nome da estrutura seguida pelo **operador ponto** (**.**) seguido pelo nome do campo.
- Caso um ponteiro para estrutura esteja sendo empregado, usa-se o nome do ponteiro seguido pelo **operador seta** (**->**) seguido pelo nome do campo.

Considere, como exemplo, as seguintes definições:

```
typedef struct {
           char  nome[30];
           int   dia, mes, ano;
       } tRegistro, *tPtrParaRegistro;

tRegistro registroDaPessoa  = { "Jose da Silva", 12, 10, 1960 };

tPtrParaRegistro  ptrParaRegistro = &registroDaPessoa;
```

Como o ponteiro `ptrParaRegistro` foi iniciado com o endereço da variável `registroDaPessoa`, cada campo dessa variável poderia ser acessado de duas maneiras, conforme mostrado na **Tabela 3–4**:

Campo	Acesso com `registroDaPessoa`	Acesso com `ptrParaRegistro`
`nome`	`registroDaPessoa.nome`	`ptrParaRegistro->nome`
`dia`	`registroDaPessoa.dia`	`ptrParaRegistro->dia`
`mes`	`registroDaPessoa.mes`	`ptrParaRegistro->mes`
`ano`	`registroDaPessoa.ano`	`ptrParaRegistro->ano`

Tabela 3–4: Acesso a Campos de Estrutura

O operador `->` é uma abreviação para as operações conjuntas de indireção de ponteiro e acesso a um campo utilizando o operador ponto. Isto é:

```
ptrParaRegistro->ano é o mesmo que: (*ptrParaRegistro).ano
```

3.10.6 Aninho de Estruturas

Um campo de uma estrutura pode ser de qualquer tipo, inclusive de um tipo estrutura. Quando um campo de uma estrutura também é uma estrutura, ele é denominado **estrutura aninhada**. Por exemplo, a definição do tipo `tRegistro` apresentada acima poderia ser reescrita como:

```
typedef struct {
           int  dia, mes, ano;
        } tData;

typedef struct {
           char   nome[30];
           tData  nascimento;
        } tRegistro2;
```

Na última definição de tipo, o campo `nascimento` é uma estrutura aninhada, pois ele é uma estrutura.

Tanto a definição de `tRegistro` quanto a definição de `tRegistro2` apresentadas acima descrevem variáveis que ocupam a mesma quantidade de espaço em memória, mas a definição de `tRegistro2` é mais elegante porque denota uma melhor organização de dados, que facilita a legibilidade e o reúso de código.

Quando uma estrutura contendo uma estrutura aninhada como campo é iniciada, utilizam-se chaves internas na iniciação da estrutura aninhada. Por exemplo:

```
tRegistro2 registroDaPessoa = { "Jose da Silva", {12, 10, 1960} };
```

O acesso a campos de estruturas aninhadas é efetuado da mesma forma que ocorre com estruturas simples. Por exemplo, o campo `nascimento` da estrutura `registroDaPessoa` pode ser acessado como:

```
registroDaPessoa.nascimento
```

Agora, o campo `nascimento` é também uma estrutura. Portanto o campo `dia` dessa estrutura pode ser acessado como:

```
(registroDaPessoa.nascimento).dia
```

que é o mesmo que:

```
registroDaPessoa.nascimento.dia
```

em virtude do fato de a associatividade do operador ponto ser à esquerda (v. **Seção 3.11**).

O acesso por meio de ponteiros é similar. Por exemplo, suponha que `ptrRegistro` é um ponteiro para o tipo `tRegistro`. Então, o campo `dia` do campo `nascimento` da estrutura apontada por `ptrRegistro` pode ser acessado como:

```
ptrRegistro->nascimento.dia
```

3.10.7 Estruturas como Parâmetros de Funções

Estruturas inteiras (e não apenas ponteiros, como ocorre com arrays) podem ser passadas como parâmetros para funções.

Existem duas maneiras de declarar um parâmetro que representa uma estrutura:

[1] O parâmetro é declarado como uma estrutura.

[2] O parâmetro é declarado como um ponteiro para estrutura.

Por exemplo, considerando a definição do tipo `tRegistro`:

```
typedef  struct  {
            char  nome[30];
            int   dia, mes, ano;
         } tRegistro;
```

as funções `ExibeRegistro1()` e `ExibeRegistro2()`, apresentadas a seguir, ilustram as duas formas de declaração de parâmetros que representam estruturas do tipo `tRegistro`:

```
void ExibeRegistro1(tRegistro registro)
{
   printf("Nome: %s\n", registro.nome);
   printf( "Nascimento: %d/%d/%d\n", registro.dia, registro.mes, registro.ano );
}

void ExibeRegistro2(const tRegistro *ptrRegistro)
{
   printf("Nome: %s\n", ptrRegistro->nome);
   printf( "Nascimento: %d/%d/%d\n", ptrRegistro->dia,
           ptrRegistro->mes, ptrRegistro->ano );
}
```

Como, na prática, estruturas ocupam relativamente muito espaço em memória, tipicamente, usam-se ponteiros para representá-las como parâmetros, mesmo quando elas são apenas parâmetros de entrada. Nesse último caso, para garantir que uma estrutura passada como parâmetro real não seja alterada, usa-se **const** na declaração do respectivo parâmetro formal, como é feito na função `ExibeRegistro2()` apresentada acima. Existe apenas uma situação na qual é justificável usar uma estrutura (em vez de um ponteiro para estrutura) como parâmetro de uma função: quando a estrutura é relativamente pequena (i.e., aproximadamente do mesmo tamanho de um ponteiro). Na maioria das situações práticas, parâmetros que representam estruturas são ponteiros.

3.10.8 Funções com Retorno de Estruturas

Uma função pode retornar tanto uma estrutura quanto um endereço de estrutura. Em qualquer situação, o tipo de retorno declarado no cabeçalho da função deve ser compatível com o valor que ela efetivamente retorna. Pela mesma razão que passar o endereço de uma estrutura como parâmetro é mais eficiente que passar a própria estrutura, retornar o endereço de uma estrutura é mais eficiente e, portanto, mais utilizado, do que retornar uma estrutura em si.

3.10.9 Uniões

Uniões são variáveis estruturadas e heterogêneas semelhantes às estruturas, mas diferem de estruturas pelo fato de seus campos serem compartilhados em memória. Quer dizer, todos os campos de uma união iniciam no mesmo endereço em memória. Assim uniões são utilizadas primariamente com o objetivo de economizar memória; i.e., quando os campos não devem coexistir ao mesmo tempo.

Uniões obedecem a regras sintáticas semelhantes àquelas usadas com estruturas, mas para definir-se um tipo de união usa-se **union** em vez de **struct** como mostra o exemplo abaixo:

```
typedef union {
           int    unidades;
           double peso;
        } tQuantidade;

tQuantidade quantidade;
```

O compilador sempre faz com que espaço suficiente seja alocado para conter o membro de maior tamanho de uma união, pois os campos de uma união são mutuamente exclusivos no sentido de que apenas um deles pode ser considerado válido num dado instante. Por exemplo, se forem feitas as seguintes atribuições à variável `quantidade` definida acima:

```
quantidade.unidades = 2;
quantidade.peso = 1.5;
```

ao final da segunda atribuição, o valor `2` será perdido e uma tentativa de acesso ao valor do campo `unidades` de `quantidade` produzirá um resultado sem sentido.

Uma união pode ser iniciada atribuindo-se um valor entre chaves ao seu primeiro campo. Por exemplo, considerando o tipo definido acima, a variável `quantidade2` pode ser iniciada como:

```
tQuantidade quantidade2 = {2};
```

3.10.10 Registros Variantes

Na prática, uniões são usadas em implementações de registros variantes. Um **registro variante** é uma estrutura composta de, pelo menos, uma parte fixa e, pelo menos, uma parte variante. As partes fixas de um tipo que representa registros variantes são campos comuns a todas as variáveis (estruturas) do tipo e as partes variantes são campos alternativos representados por uniões aninhadas. Considere, como exemplo de uso de registros variantes, o seguinte programa:

```
#include <stdio.h>  /* Função printf() */
#include <stdlib.h> /* Função exit()   */

typedef enum {UNIDADE, PESO} tTipoDeQuantidade;

typedef union {
           int    unidades;
           double peso;
        } tQuantidade;

typedef struct {
           char             descricao[20];
           double           precoUnitario;
           tTipoDeQuantidade tipoDeQuantidade;
           tQuantidade      quantidade;
        } tProduto;
```

```
void ExibeProduto(const tProduto* produto)
{
   printf( "\nProduto: %s\tPreco: R$%4.2f\t",
           produto->descricao, produto->precoUnitario );

      /* Verifica qual é o tipo de quantidade (i.e., */
      /* unidades ou peso) antes de apresentá-la     */
   if (produto->tipoDeQuantidade == UNIDADE) {
      printf( "Quantidade: %d unidades\n", produto->quantidade.unidades );
   } else if (produto->tipoDeQuantidade == PESO) {
      printf("Quantidade: %3.2f Kg\n", produto->quantidade.peso);
   } else {
      printf( "\n\nESTE PROGRAMA CONTEM UM ERRO:\n\t"
              "O valor de produto->tipoDeQuantidade\n\t"
              "nao poderia ser %d\n\n", produto->tipoDeQuantidade  );
      exit(1); /* Aborta voluntariamente o programa */
   }
}

int main(void)
{
   tProduto produto1 = { "Arroz Arboreo", 12.99, PESO };
   tProduto produto2 = { "Coco Verde", 2.50, UNIDADE, {2} };
   tProduto produto3 = { "Jaca Dura", 5.49, 3, {1} };

   produto1.quantidade.peso = 1.5;
   ExibeProduto(&produto1);
   ExibeProduto(&produto2);
   ExibeProduto(&produto3);

   return 0;
}
```

As variáveis `produto1`, `produto2` e `produto3` são registros variantes, cujas partes fixas consistem dos campos `descricao`, `precoUnitario` e `tipoDeQuantidade`, enquanto a única parte variante é representada pelo campo `quantidade`. Esse último campo é uma união consistindo dos campos alternativos `unidades` e `peso`. Apenas um dos campos dessa união é válido para uma variável do tipo `tProduto` num dado instante.

Tipicamente, quando se processam registros variantes, utiliza-se um campo fixo, denominado indicador, que informa qual é o campo variante da estrutura válido num dado instante. No último exemplo, o campo `tipoDeQuantidade` faz papel de indicador. Esse campo é uma enumeração (v. **Seção 1.13**) do tipo `tTipoDeQuantidade`, o que significa que ele deve apenas assumir um dos valores que fazem parte da definição de seu tipo.

Se um registro variante não possui campo **indicador**, não existe, em princípio, uma forma de determinar qual campo variante está correntemente em uso. Portanto o programador deve dispor de outro meio para obter essa informação, de modo a não acessar incorretamente um campo variante.

3.11 Operadores de Acesso e Definidores de Tipos

Os operadores representados por `.` e `->`, mais os operadores representados por `[]` e `()`, fazem parte de um mesmo grupo de precedência. Coletivamente, esses operadores e o operador de indireção (v. **Seção 1.18**) são denominados operadores de acesso. Para facilidade de referência, a **Tabela 3–5** apresenta um resumo desses operadores.

Nome	Símbolo	Utilizado em...
Operador de indexação	`[]`	Acesso a elemento de array
Operador de chamada de função	`()`	Chamada de função
Operador ponto	`.`	Acesso a campo de estrutura
Operador seta	`->`	Acesso a campo de estrutura
Operador de indireção	`*`	Acesso indireto a variável

Tabela 3–5: Operadores de Acesso

Excetuando-se o operador de indireção, representado por `*`, que é um operador unário, os demais operadores de acesso possuem a mais alta precedência da linguagem C e a associatividade deles é à esquerda. Isso significa que as seguintes propriedades são válidas:

```
a.b.c é equivalente a (a.b).c
a->b->c é equivalente a (a->b)->c
```

Com exceção dos símbolos utilizados como operadores de acesso a campo de estrutura e união (i.e., `.` e `->`), os demais símbolos de operadores de acesso mais as palavras-chave **struct**, **union** e **enum** são usados como **definidores** (ou **construtores**) **de tipos** da linguagem C. Ou seja, eles são usados em definições de tipos de dados derivados. A **Tabela 3–6** resume esses definidores de tipo.

Definidor	Usado em Definição de...
`*`	Ponteiro
`[]`	Array
`()`	Função
struct	Estrutura
union	União
enum	Enumeração

Tabela 3–6: Definidores de Tipo

Fazer distinção num programa entre definidores e operadores correspondentes é fácil: basta verificar o tipo de construção na qual um desses símbolos aparece. Isto é, se a construção trata-se de uma declaração ou definição, o símbolo está sendo usado como definidor; se o símbolo é utilizado numa expressão, ele representa um operador.

Quando usados como definidores, os símbolos `*`, `[]` e `()` também possuem precedência e associatividade. A **Tabela 3–7** apresenta precedências e associatividades desses definidores. Para interpretar declarações que envolvem o uso de mais um desses definidores, é preciso estar ciente das propriedades de precedência e associatividade apresentadas nessa tabela.

Grupo de Definidores	Precedência	Associatividade
`[]` e `()`	Mais alta	À esquerda
`*`	Mais baixa	À direita

Tabela 3–7: Precedência e Associatividade de Definidores de Tipo

3.12 Exemplos de Programação

3.12.1 Incrementando o Módulo de Leitura Resiliente

Problema: Crie um novo módulo, denominado `leitura2`, baseado no módulo `leitura1` apresentado na **Seção 2.7.1**. Esse novo módulo deve incluir três novas funções cujos protótipos e especificações são apresentadas na **Tabela 3–8**.

Protótipo	Especificação
`LeIntEntre(int inf, int sup)`	*Lê um número inteiro entre os valores especificados pelos parâmetros.*
`int LeOpcao(const char *op)`	*Lê um caractere via teclado limitando a aceitação a um dos caracteres que constam no string recebido como parâmetro.*
`int LeString(char *ar, int n)`	*Lê um string via teclado e armazena-o no array* `ar`*, limitando o número de caracteres a* `n - 1`*. Essa função deve retornar o número de caracteres excedentes que o usuário porventura introduziu.*

Tabela 3–8: Módulo para Leitura Resiliente

As duas funções descritas na tabela acima, devem, antes de retornar, deixar limpo o buffer associado à entrada via teclado.

Solução:

Arquivo de Cabeçalho

O arquivo de cabeçalho do módulo em questão, denominado `leitura2.h`, contém alusões das funções globais disponibilizadas pelo módulo para uso em qualquer outro o módulo. O conteúdo desse arquivo é o seguinte.

```
#ifndef _leitura2_H_
#define _leitura2_H_

extern int LeCaractere(void);
extern int LeInteiro(void);
extern int LeNatural(void);
extern int LeNaturalPositivo(void);
extern double LeReal(void);
extern int LeIntEntre(int inf, int sup);
extern int LeString(char *ar, int nElementos);
extern int LeOpcao(const char *opcoes);

#endif
```

Arquivo de Implementação: Funções Locais

As funções locais do novo módulo são exatamente as mesmas do módulo `leitura1` apresentado na **Seção 2.7.1**.

Arquivo de Implementação: Funções Globais

```
int LeIntEntre(int inf, int sup)
{
   int num;

   /* Volta para cá a cada tentativa frustrada de leitura */
inicioLeitura:
```

```
      /* Apresenta prompt e faz uma tentativa de leitura */
   printf( "\n\t>>> Digite um valor inteiro entre %d e %d\n\t\t> ", inf, sup );
   num = LeInteiro();

      /* Verifica se o valor introduzido satisfaz os critérios especificados. */
      /* Se não for o caso,  apresenta uma mensagem informando o erro ao      */
      /* usuário, e faz uma nova tentativa de leitura.                        */
   if (num < inf || num > sup) {
      printf("\n\t>>> Valor invalido\n");
      goto inicioLeitura; /* Não cria código espaguete */
   }

   return num; /* Seguramente, o número retornado satisfaz as especificações */
}

int LeString(char *ar, int nElementos)
{
   char *p; /* Usado para remover '\n' do string */
   int   nCarDeixados = 0; /* Conta o número de caracteres excedentes no buffer */

      /* Lê no máximo nElementos - 1 caracteres ou até encontrar '\n'. Quando */
      /* fgets() retorna NULL,  ocorreu erro ou tentativa de leitura além do  */
      /* final do arquivo. Nesses casos, o laço continua.                     */
   while ( fgets(ar, nElementos, stdin) == NULL )
      printf( "\a\n\t>>> %s", "Erro de leitura. Tente novamente\n\t> " );

      /* Faz p apontar para o caractere que antecede o caractere terminal */
   p = strchr(ar, '\0') - 1;

      /* Se o caractere '\n' foi lido, remove-o do   */
      /* string. Caso contrário, remove-o do buffer. */
   if (*p == '\n')
         /* Caractere '\n' faz parte do string */
      *p = '\0'; /* Sobrescreve '\n' com '\0' */
   else /* Usuário digitou caracteres demais */
      nCarDeixados = LimpaBuffer();

      /* Retorna o número de caracteres que o usuário */
      /* digitou além do esperado, sem incluir '\n'   */
   return nCarDeixados;
}

int LeOpcao(const char *opcoes)
{
   int op; /* Opção (caractere) escolhida pelo usuário */

      /* Enquanto o usuário não escolher uma opção */
      /* válida, o laço a seguir não encerra       */
   while (1) {
      op = LeCaractere(); /* Lê o caractere digitado */

         /* Verifica se o caractere é válido */
      if (strchr(opcoes, op))
         break; /* É */
      else /* Não é */
         printf( "\a\n\t>>> Opcao incorreta. Tente novamente\n\t> " );
   }

   return op; /* Certamente, a opção retornada é válida */
}
```

Observações:

- A função `LeIntEntre()` lê um número inteiro entre os valores especificados pelos parâmetros. Seus parâmetros são `inf` e `sup`, que representam o menor e o maior valores permitidos e ela retorna o número lido. Essa função assume que realmente o primeiro parâmetro é o menor e o segundo parâmetro é o maior.
- Ler um string significa ler caracteres, armazená-los num array e acrescentar o caractere `'\0'` ao final para que o array contenha um string. A função `LeString()` lê um string via teclado deixando o buffer associado a esse meio de entrada intacto novamente após a leitura. Os parâmetros dessa função são `ar[]`, que é o array que armazenará o string lido e `nElementos`, que é o tamanho desse array. Essa função retorna o número de caracteres que o usuário digitou a mais, sem incluir `'\n'`. A função `LeString()` usa a função **fgets()** declarada em `<stdio.h>`.
- O número máximo de caracteres lidos por `LeString()` é `nElementos - 1`, porque se deve deixar um espaço sobressalente para o caractere `'\0'`. Se o caractere `'\n'` for lido, ele não fará parte do string resultante.
- A função `LimpaBuffer()`, implementada na **Seção 2.7.1**, é chamada por `LeString()` para remover caracteres remanescentes no buffer de entrada.
- A função `LeOpcao()` lê um caractere usando `LeCaractere()`. Então, ela usa a função **strchr()** (v. **Tabela 3–5**) para verificar se o caractere lido faz parte do string recebido como parâmetro. A função `LeOpcao()` é denominada *LeOpcao* porque, tipicamente, ela é usada para ler opções disponíveis em menus.

3.12.2 Ordenação de Arrays pelo Método da Bolha

Preâmbulo: Um algoritmo de ordenação consiste num procedimento que descreve como organizar os elementos de um array segundo certa ordem. Um dos algoritmos de ordenação mais simples (mas também, um dos mais ineficientes) é conhecido como método da bolha (*Bubble Sort* ou *BubbleSort*, em inglês) e consiste no seguinte:

- São efetuados vários acessos sequenciais aos elementos do array a ser ordenado.
- Em cada acesso, comparam-se, dois a dois, os elementos adjacentes do array.
- Se algum par de elementos estiver fora de ordem, os elementos que o compõem são trocados de posição.
- O algoritmo encerra quando se faz um acesso sequencial do primeiro ao último elemento do array sem que ocorra nenhuma troca de posição entre dois elementos quaisquer.

Problema: Escreva uma função que ordena um array de elementos do tipo **int** usando o método da bolha.

Solução: A função `BubbleSort()`, apresentada a seguir, ordena em ordem crescente um array de elementos do tipo **int** utilizando o método de ordenação da bolha. Seus parâmetros são `ar`, que é um ponteiro para array que será ordenado, e `nElementos`, que é o número de elementos desse array. Esse é o método de ordenação mais simples de entender (e explicar), mas ele é um dos mais ineficientes (v. **Seção 6.11.3**).

```
void BubbleSort(int ar[], int nElementos)
{
   int i, aux,
       ordenado = 0; /* Informará se o array está ordenado. Essa variável */
                     /* deve ser iniciada com zero (v. abaixo).          */

   /* Supõe-se que inicialmente o array está ordenado fazendo-se 'ordenado = 1'. */
   /* Então, se forem encontrados dois elementos fora de ordem, atribui-se nova- */
   /* mente zero a 'ordenado', trocam-se os elementos de posição e recomeça-se a */
   /* verificação de ordem do array.                                             */
```

```
      /* O laço encerra quando a variável 'ordenado' for diferente de 0 */
   while (!ordenado) {
      ordenado = 1; /* Supõe que o array está ordenado */

      /* Compara cada elemento do array com o elemento seguinte. Se */
      /* for encontrado um elemento menor do que seu antecessor, os */
      /* dois elementos trocam de posição e o array é considerado   */
      /* fora de ordem (i.e., 'ordenado' assume zero).              */

      for (i = 0; i < nElementos - 1; i++)
            /* Compara cada elemento com o elemento seguinte */
         if (ar[i] > ar[i+1]){
               /* Pelo menos dois elementos estão fora de ordem */
            ordenado = 0;

               /* Troca elementos adjacentes */
            aux = ar[i];
            ar[i] = ar[i+1];
            ar[i+1] = aux;
         }  /* if */

      --nElementos; /* Mais um elemento foi colocado em seu devido lugar */
   } /* while */
}
```

3.12.3 Validando Datas

Problema: Escreva uma função que lê datas consideradas válidas a partir de `1752` (primeiro ano bissexto).

Solução: A função `LeDataValida()`, apresentada a seguir, lê uma data válida e possui um único parâmetro que é um ponteiro para a estrutura contendo a data lida e validada. Essa função retorna o endereço da estrutura que armazena os dados lidos. (O tipo `tData` é definido na page 136.)

```
tData* LeDataValida(tData *data)
{
   int diasNoMes;

      /* Lê o ano */
   printf("\n\t\t>>> Introduza o ano <<<");
   data->ano = LeIntEntre(MENOR_ANO, MAIOR_ANO);

      /* Lê o mês */
   printf("\n\t\t>>> Introduza o mes <<<");
   data->mes = LeIntEntre(1, 12);

      /* Determina o maior valor para o dia */
   diasNoMes = NumeroDeDiasNoMes(data->mes, data->ano);

      /* Lê o dia */
   printf("\n\t\t>>> Introduza o dia <<<");
   data->dia = LeIntEntre(1, diasNoMes);

   return data; /* Serviço completo */
}
```

Observações:

- Parece intuitivo ler uma data na sequência: dia, mês e ano. Acontece que, assim procedendo, a função `LeDataValida()` só seria capaz de determinar a validade do dia após a leitura do mês ou até mesmo do ano, caso o mês seja fevereiro. Por exemplo, suponha que a função solicitasse inicialmente que o usuário introduzisse o dia e o usuário digitasse `31`. Em princípio, esse dia seria válido, mas isso só

poderia ser confirmado se o mês posteriormente introduzido tivesse `31` dias. Pior ainda: suponha que a sequência intuitiva de leitura seja seguida e o usuário digite `29` como dia; então, se o mês digitado em seguida for `2` (fevereiro), a função só saberá se o dia é válido após checar se o ano é bissexto ou não. Com base nessa argumentação, a sequência de leitura da função `LeDataValida()` foi escolhido como: ano, mês e dia.

- A função `LeDataValida()` chama a função `LeIntEntre()`, apresentada na **Seção 3.12.1**, para ler o ano e o mês. É interessante notar que essa função pode ser usada quando não há um limite superior imposto pelo programa para o valor a ser lido, como ocorre no caso da leitura do ano, que só possui limite inferior. Quer dizer, como o programa não limita o valor do maior ano que pode ser introduzido, usa-se o maior valor imposto pela implementação de C para o tipo **int** (i.e., o valor da constante **INT_MAX** definida no cabeçalho `<limits.h>`). Esse valor é usado na definição da constante simbólica `MAIOR_ANO`, que é usada na chamada da função `LeIntEntre()` para leitura do ano.
- Em seguida, a função `LeDataValida()` chama `NumeroDeDiasNoMes()` para determinar o número de dias do mês introduzido pelo usuário. Essa função, apresentada abaixo, também leva em consideração o respectivo ano.

```
int NumeroDeDiasNoMes(int mes, int ano)
{
      /* Se o mês não for válido retorna 0 */
   if (mes < 1 || mes > 12)
      return 0; /* Mês inválido */

      /* Retorna o número de dias do mês. Note que o mês 2 é um caso especial, */
      /* pois o valor retornado depende do fato de o ano ser bissexto ou não.  */
      /* Os demais messes possuem números de dias fixos.                       */
   switch(mes) {
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
         return 31;
      case 2:
         return EhAnoBissexto(ano) ? 29 : 28;
      default:
         return 30;
   }
}
```

A função `NumeroDeDiasNoMes()` chama a função `EhAnoBissexto()`, que verifica se um número é bissexto e pode ser definida como mostrado a seguir. Note que um ano é bissexto quando ele é múltiplo de *400* ou ele é múltiplo de *4*, mas não é múltiplo de *100*.

```
int EhAnoBissexto(int ano)
{
      /* Se o ano for anterior ao primeiro ano considerado */
      /* bissexto, ele não pode ser assim considerado      */
   if (ano < MENOR_ANO)
      return 0;

   return !(ano%400) || (!(ano%4) && ano%100);
}
```

3.12.4 Operações com Vetores Reais

Observação: Este exemplo lida com conceitos com os quais, talvez, você nunca tenha se deparado. O foco aqui continua sendo obviamente programação, mas se você não se sentir confortável com o tema, passe adiante.

Preâmbulo: **Vetores**[2] são entidades matemáticas com os seguintes atributos: magnitude (ou norma), direção e sentido. No espaço bidimensional, vetores são representados por dois valores reais nas direções ortogonais que definem o espaço. Assim vetores bidimensionais podem ser representados em C pelo seguinte tipo:

```
typedef struct {
           double x; /* Componente horizontal */
           double y; /* Componente vertical   */
        } tVetor;
```

No plano, pelo menos, as seguintes operações sobre vetores são definidas:

(a) **Multiplicação de um vetor por uma constante**, resultando num outro vetor. Se um vetor é representado por `(x, y)` e `c` é a constante, essa operação resulta no vetor `(cx, cy)`.

(b) **Soma de dois vetores**, resultando em outro vetor. Se os vetores a serem somados são `(x1, y1)` e `(x2, y2)`, a soma deles é o vetor `(x1 + x2, y1 + y2)`.

(c) **Norma** (ou **magnitude**) **de um vetor**, que resulta num valor real positivo. Se um vetor é representado por `(x, y)`, sua norma é a raiz quadrada de `x2 + y2`.

(d) **Produto interno de dois vetores**, que resulta num valor real. Se os vetores em questão são `(x1, y1)` e `(x2, y2)`, o produto interno deles é dado por `x1*y1 + x2*y2`.

(e) **Orientação** de um vetor, que é o ângulo que o vetor forma com o eixo horizontal medido no sentido anti-horário.

Problema: Escreva funções em C que implementem as operações sobre vetores descritas no preâmbulo.

Solução: As funções que implementam o que foi solicitado utilizam as seguintes definições de constantes simbólicas, além do tipo `tVetor` apresentado acima.

```
#define PI              3.1416
#define GRAU_POR_RADIANO 57.3
```

Solução de (a): A função `VetorVezesConstante()` multiplica um vetor por uma constante e seus parâmetros são:

- `*res` (saída) — resultado da operação
- `*v1` (entrada) — vetor a ser multiplicado
- `cte` (entrada) — a constante

Essa função retorna o endereço da estrutura que contém o resultado da operação.

```
tVetor *VetorVezesConstante(tVetor *res, const tVetor *v, double cte)
{
   res->x = cte*v->x;
   res->y = cte*v->y;

   return res;
}
```

Solução de (b): A função `SomaVetores()` soma dois vetores e seus parâmetros são:

[2] Muitos textos de programação utilizam a denominação *vetor* para o conceito de array. Esses textos, provavelmente, teriam dificuldade em discutir o conceito de vetor.

- `*res` (saída) — resultado da operação
- `*v1` (entrada) — primeiro vetor
- `*v2` (entrada) — segundo vetor

Essa função retorna o endereço da estrutura que armazena o resultado da operação.

```
tVetor *SomaVetores(tVetor *res, const tVetor *v1, const tVetor *v2)
{
   res->x = v1->x + v2->x;
   res->y = v1->y + v2->y;

   return res;
}
```

Solução de (c): A função `Norma()` calcula e retorna a norma de um vetor representado por uma estrutura do tipo `tVetor`:

```
double Norma(const tVetor *v)
{
   return sqrt(v->x*v->x + v->y*v->y);
}
```

Solução de (d): A função `ProdutoInterno()` calcula e retorna o produto interno de dois vetores representados por estruturas do tipo `tVetor`:

```
double ProdutoInterno(const tVetor *v1, const tVetor *v2)
{
   return v1->x*v2->x + v1->y*v2->y;
}
```

Solução de (e): A função `Orientacao()` calcula e retorna a orientação de um vetor representado por uma estrutura do tipo `tVetor`. O valor retornado por essa função é o seguinte

- `0.0`, se o vetor for nulo [i.e., `(0, 0)`].
- Se o vetor não for nulo, o ângulo formado entre o vetor e o eixo horizontal medido no sentido anti-horário em radianos.

```
double Orientacao(const tVetor *v)
{
   double angulo;

      /* Se o vetor for nulo, retorna 0.0 */
   if (!v->x && !v->y)
      return 0.0;

   /* A chamada de função atan2(u, v) calcula o arco cuja tangente é v/u, levando */
   /* em conta o quadrante em que se encontra o vetor. A função atan() também     */
   /* calcula arco tangente, mas é incapaz de considerar o quadrante do vetor.    */
   /* As duas funções são declaradas em <math.h>.                                 */

   angulo = atan2(v->y, v->x);

      /* Se o ângulo for negativo, ele foi medido no sentido horário. Nesse */
      /* caso, soma-se 2*PI ao ângulo para obter o resultado desejado.      */
   return angulo < 0 ? 2*PI + angulo : angulo;
}
```

Observação: Se você nunca estudou Cálculo Vetorial, provavelmente terá dificuldade em entender o significado das operações sobre vetores descritas no presente exemplo. Mas, mesmo assim, as

implementações dessas funções são tão simples que você não terá dificuldade de entender o programa apresentado.

3.12.5 Coordenadas Retangulares e Polares

Preâmbulo: **Sistema polar de coordenadas** é um sistema bidimensional de coordenadas no qual cada ponto é representado por duas coordenadas reais: **(1)** **coordenada radial**, que é a distância do ponto à origem do sistema e **(2)** coordenada angular, que é um ângulo em radianos medido, tipicamente, em relação à direção horizontal em sentido anti-horário.

Um ponto `(r, θ)` no sistema cartesiano pode ser representado em coordenadas polares como:

```
r = sqrt(x*x + y*y);
θ = atan2(y, x);
```

Essas expressões estão escritas usando a sintaxe de C (mas, θ não é identificador válido em C). A função **sqrt()** (declarada em `<math.h>`) calcula a raiz quadrada de um número real e a função **atan2()** (também declarada em `<math.h>`) calcula o arco tangente do ângulo formado pela reta que passa pela origem e pelo ponto de coordenadas `x` e `y`, levando em consideração o quadrante no qual esse ponto se encontra. Quando o ponto encontra-se no terceiro ou quarto quadrante, o valor retornado por essa função é negativo. Portanto, para obter sempre um ângulo positivo (i.e., medido no sentido anti-horário), quando o valor retornado por essa função for negativo, soma-se 2π a esse valor. Isto é, somando-se 2π a esse ângulo quando o valor retornado por **atan2()** é negativo garante-se a consistência na apresentação dos resultados, pois, assim, os ângulos serão sempre medidos no sentido anti-horário.

Um ponto `(x, y)` em coordenadas polares pode ser representado no sistema cartesiano como (novamente, usando a sintaxe de C):

```
x = r*cos(θ);
y = r*sin(θ);
```

As funções **cos()** e **sin()** calculam, respectivamente, o cosseno e o seno dos seus respectivos parâmetros reais e são declaradas no cabeçalho `<math.h>`.

As seguintes definições de tipo podem ser utilizadas na implementação de pontos nos dois sistemas de coordenadas:

```
   /* Categorias de sistemas */
typedef enum {POLAR, CARTESIANO} tSistema;
```

```
   /* Pontos em qualquer sistema */
typedef  struct {
            tSistema sistema; /* Sistema de coordenadas  */
            double   a, b;    /* As coordenadas do ponto */
         } tPonto;
```

Nesse tipo de estrutura, os campos `a` e `b` representam as coordenadas do ponto, que são interpretadas de acordo com sistema indicado pelo campo `sistema`. Ou seja, se o sistema de coordenadas for cartesiano, `a` e `b` representarão um ponto `(x, y)`; se o sistema de coordenadas for polar, `a` e `b` representarão um ponto `(r, θ)` (v. acima).

Problema: (a) Escreva uma função que lê via teclado um ponto em qualquer sistema de coordenadas descrito. (b) Escreva uma função que escreve na tela um ponto em qualquer sistema de coordenadas descrito. (c) Escreva uma função que converte um ponto em coordenadas cartesianas para coordenadas

polares. (d) Escreva uma função que converte um ponto em coordenadas polares para coordenadas cartesianas.

Solução de (a): A função `LePonto()`, definida abaixo, lê valores para os campos de uma estrutura do tipo `tPonto` e armazena-os na variável para a qual seu único parâmetro aponta. Ela retorna o endereço da estrutura que contém o resultado.

```
tPonto *LePonto(tPonto *ponto)
{
   int op; /* Opção de sistema de coordenadas */

      /* Lê a opção de sistema de coordenadas */
   printf("\nSistema de coordenadas (P = Polar, C = cartesiano)? ");
   op = LeOpcao("pPcC");

      /* Lê as coordenadas de acordo com o sistema escolhido pelo usuário */
   if (op == 'P' || op == 'p') { /* Sistema polar */
      ponto->sistema = POLAR;
      printf("\tCoordenada radial: ");
      ponto->a = LeReal();

         /* No sistema polar, este componente não pode ser negativo */
      while (ponto->a < 0) {
         printf( "\a\n\t>>> Esta coordenada nao pode ser negativa <<<\n" );
         printf("\tCoordenada radial: ");
         ponto->a = LeReal();
      }
         /* Lê a coordenada angular em graus e armazena-a em  */
         /* radianos, pois parece mais natural para o usuário */
      printf("\tCoordenada angular (graus): ");
      ponto->b = RADIANO_POR_GRAU * LeReal();
   } else {  /* Sistema cartesiano */
      ponto->sistema = CARTESIANO;

      printf("\tCoordenada horizontal: ");
      ponto->a = LeReal();

      printf("\tCoordenada vertical: ");
      ponto->b = LeReal();
   }
   return ponto;
}
```

A leitura da coordenada angular é feita em graus, pois, assim procedendo, o programa parece ser mais amigável ao usuário. Entretanto, para uso interno, o programa armazena esse ângulo em radianos e, para converter graus em radianos, é usada a constante simbólica `RADIANO_POR_GRAU`.

Solução de (b): A função `ExibePonto()`, apresentada abaixo, escreve na tela uma estrutura do tipo `tPonto`.

```
void ExibePonto(const tPonto *ponto)
{
      /* Escreve o ponto de acordo com o sistema de coordenadas */
   if (ponto->sistema == POLAR) { /* Sistema polar */
      printf("\traio = %3.1f\n", ponto->a);

         /* A coordenada angular é escrita em graus. Se o ângulo for */
         /* negativo, soma-o a 2*PI antes de convertê-lo em graus.   */
      if (ponto->b < 0) {
         printf("\tteta (graus) = %3.1f\n", GRAU_POR_RADIANO*(2*PI + ponto->b));
      } else {
```

```
         printf("\tteta (graus) = %3.1f\n", GRAU_POR_RADIANO*ponto->b);
      }
   } else {  /* Sistema cartesiano */
      printf("\tx = %3.1f\n", ponto->a);
      printf("\ty = %3.1f\n", ponto->b);
   }
}
```

A função `ExibePonto()` usa a constante simbólica `GRAU_POR_RADIANO`, que é necessária para converter um ângulo de radiano para grau.

Solução de (c): A função `CartesianoParaPolar()` converte um ponto de coordenadas cartesianas para coordenadas polares. Seus parâmetros são: `polar` (saída), que é um ponteiro para o ponto em coordenadas polares, `cart` (entrada), que aponta para o ponto em coordenadas cartesianas a ser convertido. Essa função retorna o endereço da estrutura que armazena o resultado da conversão.

```
tPonto *CartesianoParaPolar(tPonto *polar, const tPonto *cart)
{
      /* Se o sistema do ponto já for polar, é preciso apenas  */
      /* copiar o parâmetro de entrada para o parâmetro de     */
      /* saída. Caso contrário, é preciso efetuar a conversão. */

   if (cart->sistema == POLAR) {
      *polar = *cart; /* Conversão é desnecessária */
   } else { /* Efetua a conversão do ponto */
      polar->sistema = POLAR;
      polar->a = sqrt(cart->a*cart->a + cart->b*cart->b);

         /* Se os dois componentes são iguais a 0, o ângulo também é 0. */
         /* Caso contrário, ele será calculado por meio de atan2().     */
      if (!cart->a && !cart->b)
         polar->b = 0.0;
      else
         polar->b = atan2(cart->b, cart->a);
   }

   return polar;
}
```

A função `CartesianoParaPolar()` apenas implementa a fórmula de conversão de coordenadas cartesianas para polares apresentada no preâmbulo. Mas, o seguinte trecho dessa função merece destaque:

```
if (!cart->a && !cart->b)
   polar->b = 0.0;
else
   polar->b = atan2(cart->b, cart->a);
```

Essa instrução **if** é necessária, pois, quando `cart->a` e `cart->b` são iguais a zero, trata-se do ponto `(0, 0)` (em qualquer sistema de coordenadas em discussão). Ou seja, nesse caso, se a função **atan2()** fosse chamada como `atan2(0, 0)`, ela retornaria um valor indefinido.

Solução de (d): A função `PolarParaCartesiano()` converte um ponto de coordenadas polares para coordenadas cartesianas. Seus parâmetros são: `cart` (saída), que é um ponteiro para o ponto em coordenadas cartesianas, `polar` (entrada), que aponta para o ponto em coordenadas polares a ser convertido. Essa função retorna o endereço da estrutura que armazena o resultado da conversão.

```
tPonto *PolarParaCartesiano( tPonto *cart, const tPonto *polar )
{
      /* Se o sistema do ponto já for cartesiano, é necessário apenas  */
      /* copiar o parâmetro de entrada para o parâmetro de saída. Caso */
      /* contrário, é preciso efetuar a conversão.                     */
   if (polar->sistema == CARTESIANO) {
      *cart = *polar; /* Conversão não é necessária */
   } else { /* Efetua a conversão do ponto */
      cart->sistema = CARTESIANO;
      cart->a = polar->a*cos(polar->b);
      cart->b = polar->a*sin(polar->b);
   }

   return cart;
}
```

Essa última função implementa a fórmula de conversão de coordenadas polares para cartesianas apresentada no preâmbulo e não requer considerações complementares.

3.13 Exercícios de Revisão

Arrays (Seção 3.1)

1. O que é um array?
2. Qual é a sintaxe usada na definição de arrays?
3. Quais são as vantagens obtidas ao se declarar o tamanho de um array usando-se uma constante simbólica em vez de usando-se o valor da constante em si?
4. (a) O que é índice de um array? (b) Quais são os valores válidos de um índice de array? (c) O que ocorre quando o programador usa um índice inválido para acesso a um elemento de array?
5. (a) Qual é o problema com o seguinte trecho de programa? (b) Qual é a gravidade desse problema?

```
int i, ar[5];

for (i = 1; i <= 5; ++i) {
   ar[i] = i;
}
```

6. No seguinte fragmento de programa, o conteúdo do array `ar1[]` é copiado para o array `ar2[]`. (a) Explique por que ele não é portável. (b) Sugira uma forma de tornar esse fragmento de programa portável.

```
int i = 0, ar1[5], ar2[5];

...

while (i < 5) {
   ar2[i] = ar1[i++];
}
```

7. (a) Como deve ser escrita a iniciação de um array unidimensional? (b) É obrigatória a iniciação explícita de todos os elementos de um array? (c) Qual é a maneira mais simples de iniciar com zero todos os elementos de um array de duração automática? (d) Isso é necessário quando o array é de duração fixa?
8. Qual é a diferença em termos de iniciação entre arrays de duração fixa e arrays de duração automática?
9. É ilegal incluir numa iniciação um número de valores maior do que o tamanho do array? Explique.
10. Considere um array `ar[]`. (a) Como se determina o número de bytes ocupados pelo array `ar[]`? (b) Como se determina o número de bytes ocupados por um elemento do array `ar[]`? (c) Como o número de elementos do array `ar[]` pode ser determinado sem que se tenha que recorrer à sua definição?

11. Por que a aplicação do operador **sizeof** num parâmetro formal que representa um array não resulta no tamanho em bytes do array?

Aritmética de Ponteiros (Seção 3.2)

12. (a) Quais são as operações aritméticas permitidas sobre ponteiros? (b) Quando um inteiro é adicionado ou subtraído a um ponteiro, como a operação é interpretada?

13. (a) O que é fator de escala em aritmética de ponteiros? (b) Que operação aritmética sobre ponteiros não é influenciada por fator de escala, independente do tipo de ponteiro envolvido?

14. Suponha que um ponteiro `p` possui num determinado instante o valor `1240`. Se o valor de `p + 1` for `1241`, o que se pode concluir em relação ao tipo de `p`?

15. Suponha que um ponteiro `p` possui num determinado instante o valor `1240`. Se o valor de `p + 1` for `1244`, é possível inferir qual é o tipo de `p`?

16. Suponha que `p1` e `p2` são ponteiros. (a) Quando a operação `p1 - p2` ou `p2 - p1` é ilegal? (b) Quando ela é legal, mas não faz sentido?

17. Considerando as seguintes definições de variáveis:

```
double *p1, *p2;
int     j;
char   *p3;
```

quais das seguintes expressões são válidas?

(a) `p2 = p1 + 4;`
(b) `j = p2 - p1;`
(c) `j = p1 - p2;`
(d) `p1 = p2 - 2;`
(e) `p3 = p1 - 1;`
(f) `j = p1 - p3;`

18. Suponha que `p` seja um ponteiro para **int**. (a) Interprete cada uma das expressões a seguir consideradas legais. (b) Quais delas são ilegais?

(a) `*p`
(b) `++p`
(c) `++*p`
(d) `*++p`
(e) `*p++`
(f) `p++*`
(g) `p*++`

Relações entre Ponteiros e Arrays (Seção 3.3)

19. Seja `ar[]` um array unidimensional. (a) Descreva duas formas diferentes de especificar o endereço do elemento de índice `i` desse array. (b) Descreva duas formas diferentes de acessar o valor do elemento de índice `i` desse array.

20. Um programa em C contém a seguinte definição de array:

```
int  a[8] = {10, 20, 30, 40, 50, 60, 70, 80}
```

(a) O que representa `a`?
(b) O que representa `a + 2`?
(c) Qual é o valor de `*a`?
(d) Qual é o valor de `*a + 2`?

(e) Qual é o valor de `*(a + 2)`?

21. Dadas as seguintes iniciações:

```
int  ar[] = {10, 15, 4, 25, 3, -4};
int *p = &ar[2];
```

quais são os resultados das avaliações das seguintes expressões:

(a) `*(p + 1)`
(b) `p[-1]`
(c) `ar - p`
(d) `ar[*p++]`
(e) `*(ar + ar[2])`

22. Considere a seguinte iniciação do array `ar[]`:

```
int ar[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
```

(a) Escreva um trecho de programa contendo um laço **for** responsável pela apresentação na tela dos valores do array `ar[]` utilizando índices.

(b) Repita a tarefa do item anterior utilizando aritmética de ponteiros, em vez de índices.

23. O que justifica o fato de indexação de arrays em C começar em zero, e não em um, como seria mais natural?

O Qualificador const (Seção 3.4)

24. (a) Para que serve a palavra-chave **const**? (b) Uma variável qualificada com **const** pode ser alterada? (c) Em que situações práticas a palavra-chave **const** deve ser utilizada?

25. Qual é a diferença entre usar **const** e **#define** para definir uma constante?

26. Interprete cada uma das seguintes definições de ponteiros.

(a) `const int *p;`
(b) `int const *p;`
(c) `int *const p;`
(d) `const int *const p;`
(e) `int const *const p;`

27. Considerando os protótipos de função a seguir, é possível dizer se `x` é um parâmetro de entrada, de saída ou de entrada e saída?

(a) `void F(int x)`
(b) `void F(int *x)`
(c) `void F(const int *x)`

28. Por que não faz sentido usar **const** com um parâmetro que não é ponteiro?

Uso de Arrays com Funções (Seção 3.5)

29. (a) Como um parâmetro formal que representa um array deve ser declarado? (b) Como deve ser passado um parâmetro real correspondente a um parâmetro formal que representa um array? (c) Como o nome de um array passado como parâmetro real para uma função é interpretado?

30. (a) Se um array é passado para uma função e um de seus elementos é alterado, essa alteração é reconhecida na porção do programa que chamou a função? (b) Se esse for o caso, como se poderia garantir que os elementos de um array não são modificados por uma função?

31. O tipo de retorno de uma função pode ser um array?

32. (a) Por que um array de duração automática cujo endereço é retornado por uma função é denominado zumbi? (b) Por que erros causados por zumbis são difíceis de detectar? (c) Por que arrays de duração fixa nunca são zumbis?

33. (a) Quais das seguintes funções retornam zumbis? (b) Qual delas retorna um valor incompatível com tipo de retorno da função?

(i)
```
int *F1(int n)
{
   ...
   return &n;
}
```

(ii)
```
int *F2(int *p)
{
   ...
   return &p;
}
```

(iii)
```
int *F3(int *p)
{
   ...
   return p;
}
```

(iv)
```
int F4(int *p)
{
   ...
   return *p;
}
```

34. (a) O que há de errado com o seguinte fragmento de programa? (b) Por que as chances de este programa ser abortado são maiores se o parâmetro `p` da função `F()` representar um array do que se ele representar um ponteiro para uma única variável?

```
int F(double *p, int n)
{
   ...
   return 0;
}

int main(void)
{
   int     x;
   double *ptr;

   x = F(ptr, 10);
   ...
}
```

35. Na **Seção 3.5**, recomenda-se que não se deve retornar o endereço de um parâmetro. Considerando essa recomendação, se uma função recebe um endereço como parâmetro, é seguro retorná-lo, como faz a seguinte função?

```
int *F(int ar[])
{
   ...
   return ar;
}
```

36. O parâmetro `ar` no protótipo de função abaixo necessariamente representa um array?

```
int F(double ar[], int n)
```

37. O parâmetro `p` no protótipo de função a seguir pode representar um array?

```
int F(double *p, int n)
```

Arrays Multidimensionais (Seção 3.6)

38. (a) O que é um array bidimensional? (b) O que é um array tridimensional?

39. Por que o uso de arrays com mais de quatro dimensões deve ser evitado?

40. Considerando arrays bidimensionais, (a) o que é linha? (b) O que é coluna?

41. (a) Como deve ser escrita a iniciação de um array multidimensional? (b) É obrigatória a iniciação de todos os componentes do array?

42. Quando se atribuem valores a elementos de um array multidimensional, qual é a vantagem de se incluírem chaves em torno de grupos de valores?

43. Quando um array multidimensional é utilizado como parâmetro de uma função, como ele deve ser declarado?

44. Como os elementos do array `arBi[][]` serão iniciados com a iniciação a seguir?

```
int arBi[5][3] = { 1, 2, 3, 4, 5, 6, 7 };
```

Strings e Caracteres (Seção 3.7)

45. (a) O que é um string? (b) Todo array de caracteres é um string? (c) Todo string é um array de caracteres?

46. (a) O que é caractere nulo? (b) Qual é o valor inteiro associado ao caractere nulo em qualquer código de caracteres usado em C? (c) Para que serve o caractere nulo?

47. Se a iniciação do array `ar[]`:

```
char ar[] = {'b', 'o', 'l', 'a', '\0'};
```

é o mesmo que:

```
char  ar[] = "bola";
```

por que existe essa segunda notação para iniciação de arrays de caracteres?

48. A iniciação do array `ar[]` a seguir é legal? Explique.

```
char  ar[3] = "bola";
```

49. Na iniciação a seguir, como os elementos do array `ar[]` são iniciados?

```
char  ar[10] = "bola";
```

50. (a) A seguinte iniciação do array `ar[]` é legal? (b) Se esse for o caso, qual será o conteúdo do array `ar[]` após essa iniciação?

```
char ar[4] = "bola";
```

51. (a) Interprete a definição de variável a seguir. (b) Supondo que um ponteiro ocupe 4 bytes, quantos bytes serão alocados em decorrência dessa definição?

```
char *ar[] = {"azul", "vermelho", "branco"};
```

52. Por que um programa pode ser abortado ao tentar alterar o conteúdo de um string constante?

53. (a) Por que é recomendado o uso de **const** na definição de ponteiros para strings constantes? (b) Dê exemplo do uso preventivo de **const** na definição de tal ponteiro.

54. O que escreve na tela cada uma das seguintes chamadas de **printf()**? Justifique suas respostas.

(a) `printf("\nValor de '0': %d", '0');`
(b) `printf("\nValor de '\0': %d", '\0');`
(c) `printf("\nValor de '0': %c", '0');`
(d) `printf("\nValor de '\\0': %c", '\0');`

55. Qual deve ser o terceiro parâmetro da função **fgets()** quando essa função é invocada para ler strings via teclado?

56. Por que o seguinte programa será provavelmente abortado quando for executado?

```
#include <stdio.h>

int main(void)
{
   char *p = "otorrinolaringologista";

   printf("\nDigite no maximo 5 caracteres: ");
   scanf("%6s", p);

   printf("String lido: %s", p);

   return 0;
}
```

57. Um aluno de programação escreveu a seguinte função com o objetivo de concatenar dois strings, mas ela está incorreta. Descubra e corrija os erros de programação apresentados por esta função.

```
char *Concatena(char *str, const char *ptr)
{
   while (*str++)
      str++;

   while (*str++ = *ptr++)
      ; /* Instrução vazia */

   return str;
}
```

58. Que cuidado deve ser tomado quando se usa o valor retornado por **strlen()**?

59. Se função **strcpy()** armazena o resultado da cópia do string recebido como segundo parâmetro no array recebido como primeiro parâmetro, para que serve o valor retornado por essa função?

60. (a) A função **strcmp()** é útil em ordenação de strings? Explique. (b) Se a função **strcmp()** não é muito útil em ordenação de strings, para que ela serve afinal?

61. Para que serve a função **strstr()**?

62. Em que diferem as funções **strchr()** e **strrchr()**?

63. Como se faz um ponteiro `p` do tipo **char** * apontar para o caractere terminal de um string `str` utilizando apenas uma instrução?

64. Como se pode calcular o comprimento de um string representado pelo ponteiro `str` com uma única expressão usando **strchr()** [ou **strrchr()**]?

65. Que cuidado se deve tomar para evitar corrupção de memória quando se chama a função **strcat()**?

66. Que função declarada no cabeçalho `<ctype.h>` você utilizaria num programa para testar se um caractere é classificado como:

(a) Alfanumérico
(b) Letra
(c) Dígito
(d) Letra maiúscula

67. Que função declarada em `<ctype.h>` é usada para converter letras maiúsculas em minúsculas?

Função main() com Parâmetros (Seção 3.8)

68. (a) Considerando qualquer padrão da linguagem C, a função **main()** pode ter tipo de retorno **void**? (b) Se a resposta for negativa, por que existem programas que definem o tipo de retorno de **main()** como **void**?

69. (a) Quais são os significados dos parâmetros `argc` e `argv` usados pela função **main()**? (b) Quais são as origens das denominações *argc* e *argv*? (c) Esses parâmetros precisam ser realmente denominados assim?

70. Como um programa pode obter seu nome de arquivo executável?

71. Suponha que um programa precisa processar dois argumentos de linha de comando. Que teste ele deve efetuar ao início de sua execução para verificar se foi invocado corretamente?

Tipos Definidos pelo Programador (Seção 3.9)

72. Qual é a sintaxe utilizada numa definição de tipo em C?

Estruturas e Uniões (Seção 3.10)

73. (a) O que é uma estrutura? (b) Qual é a diferença conceitual entre estruturas e arrays?

74. Uma estrutura pode possuir um campo com o mesmo nome do campo de outra estrutura?

75. (a) O que é um rótulo de estrutura? (b) Quando o uso de rótulo de estrutura é estritamente necessário?

76. (a) O que é uma estrutura com autorreferência? (b) Qual é a utilidade de estruturas com autorreferência?

77. (a) Por que a seguinte definição de tipo não é permitida? (b) Como corrigi-la?

```
typedef struct {
           int   conteudo;
           tNo  *proximo;
         } tNo, *tLista;
```

78. (a) Como os membros de uma estrutura podem ser iniciados? (b) Pode-se incluir iniciação numa definição de um tipo de estrutura?

79. Como são acessados os campos de uma estrutura?

80. Para que servem os operadores `.` e `->`?

81. O que é uma estrutura aninhada?

82. Como uma estrutura pode ser passada como parâmetro para uma função?

83. Em que situações deve-se usar um ponteiro para estrutura como parâmetro de uma função?

84. Por que, tipicamente, é mais eficiente ter como parâmetro de função um ponteiro para estrutura do que uma estrutura?

85. Quando se deve usar **const** na definição de um ponteiro para estrutura como parâmetro de uma função?

86. Uma função pode retornar uma estrutura?

87. (a) Por que é mais eficiente para uma função retornar o endereço de uma estrutura do que uma estrutura inteira? (b) Que cuidado deve ser tomado nesse caso?

88. (a) O que é uma união? (b) Quais são as semelhanças entre uniões e estruturas? (c) Qual é a principal diferença entre uniões e estruturas? (d) Em que situações uniões são úteis?

89. (a) Como se pode iniciar um membro de uma união? (b) Qual é a diferença entre iniciações de uniões e estruturas?

90. O que é um registro variante?

91. Para que serve um campo indicador de um registro variante?

Operadores de Acesso e Definidores de Tipos (Seção 3.11)

92. (a) O que são operadores de acesso? (b) Quais são os operadores de acesso de C? (c) Em que situações são empregados operadores de acesso?

93. (a) Que operador de acesso não faz parte do mesmo grupo de precedência dos demais operadores de acesso? (b) Qual é a precedência dos operadores [], (), `.` (ponto) e `->` com relação aos demais operadores da linguagem C? (c) Qual é a associatividade desses operadores?

94. Suponha que um tipo de estrutura seja definido como:

```
typedef struct {
           int    a;
           double b;
} tEstrutura;
```

Suponha ainda que as variáveis `e` e `p` sejam definidas como a seguir:

```
tEstrutura e, *p = &e;
```

(a) Por que a seguinte instrução é ilegal? (b) Apresente duas maneiras de corrigir o erro apresentado por essa instrução.

```
*p.b = 2.5;
```

95. Suponha que um tipo de estrutura seja definido como:

```
typedef struct {
           int     a;
           double *b;
} tEstrutura;
```

e que se tenha uma estrutura `e` do tipo `tEstrutura` definida e iniciada como:

```
tEstrutura e = {10};
```

(a) Por que a seguinte instrução é legal (do ponto de vista sintático)? (b) Essa instrução poderá causar o mau funcionamento de um programa que a execute? Explique.

```
*e.b = 2.5;
```

96. (a) O que é um definidor de tipo? (b) Quais são os definidores de tipos disponíveis em C?

Exemplos de Programação (Seção 3.12)

97. (a) O que é um algoritmo de ordenação? (b) Descreva o algoritmo de ordenação conhecido como *Bubble Sort*.

3.14 Exercícios de Programação

EP3.1 Escreva uma macro que recebe o nome de um array como parâmetro e, quando expandida, resulta no número de elementos do array.

EP3.2 (a) Escreva uma função que exibe um array de inteiros na tela, de tal modo que elementos repetidos sejam escritos uma única vez. (b) Escreva um programa que testa a função requerida no item (a).

EP3.3 (a) Escreva uma função que retorna `1` se um array de elementos do tipo **int** estiver ordenado em ordem decrescente ou zero, em caso contrário. (b) Escreva uma função **main()** que lê via teclado valores do tipo **int** até um limite máximo estipulado por uma constante simbólica, armazena esses valores num array na ordem em que eles são introduzidos e usa a função especificada no item (a) para testar se o array está ordenado em ordem decrescente.

EP3.4 Escreva uma função que verifica se um array `ar[]` de elementos do tipo **int** possui um elemento `ar[i]` que é a soma de dois elementos que o antecedem no array. Ou seja, `ar[i]` deve ser a igual a `ar[j] + ar[k]`, sendo `j < i` e `k < i`.

EP3.5 Escreva uma função que retorna `1` quando um string possui apenas letras e dígitos ou `0`, em caso contrário.

EP3.6 Escreva uma função que substitui cada caractere de tabulação de um string por um espaço em branco.

EP3.7 Escreva uma função, denominada `TransformaStr()`, que recebe um string como primeiro parâmetro e um caractere como segundo parâmetro. Quando o segundo parâmetro for o caractere `'M'`, essa função deve transformar o string de tal modo que todas as suas letras passem a ser maiúsculas. Quando o segundo parâmetro for `'m'`, a função deve transformar o string de tal modo que todas as suas letras

sejam minúsculas. Quando o segundo parâmetro não for `'M'` ou `'m'` essa função não deve promover nenhuma transformação no string. A função deverá retornar o endereço do string transformado.

EP3.8 Implemente uma função, denominada `ComparaStrings()`, funcionalmente equivalente à função **strcmp**().

EP3.9 Implemente uma função, denominada `EncontraPrimeiroChar()`, funcionalmente equivalente à função **strchr**().

EP3.10 Escreva uma função, denominada `OcorrenciasCar()`, que conta o número de ocorrências de um caractere num string.

EP3.11 (a) Escreva uma função que substitui todas as ocorrências de um dado caractere num string por outro caractere. O protótipo dessa função deve ser:

```
char *SubstituiCaracteres( char *str, int substituir, int novo )
```

Os parâmetros dessa função são interpretados como:

- `str` é o string no qual serão feitas as substituições
- `substituir` é o caractere que será substituído
- `novo` é o caractere que substituirá as ocorrências do segundo parâmetro

O retorno dessa função deve ser o endereço do string recebido como parâmetro.

EP3.12 (a) Escreva uma função que retorna o token de ordem `n` de um string, se este existir; caso contrário, a função deve retornar **NULL**. O protótipo dessa função deve ser:

```
char *EnesimoToken( char *str, const char *separadores, int n )
```

Nesse protótipo, os parâmetros têm os seguintes significados:

- `str` é o string no qual o enésimo token será procurado
- `separadores` é um string contendo os possíveis separadores de tokens
- `n` é o número de ordem do token desejado

(b) Escreva um programa para testar a função `EnesimoToken()`.

EP3.13 (a) Escreva uma função que copia os `n` caracteres iniciais de um string. O protótipo dessa função deve ser:

```
char *CopiaInicio( char *destino, const char *origem, int n )
```

Os parâmetros dessa função são interpretados como:

- `destino` é o array que receberá a cópia
- `origem` é o string que doará os caracteres
- `n` é o número de caracteres iniciais que serão copiados.

O retorno dessa função deve ser o endereço do array recebido como primeiro parâmetro. (b) Escreva um programa para testar a função `CopiaInicio()`.

EP3.14 Escreva um programa que soma números reais passados para o programa como argumentos de linha de comando.

EP3.15 Escreva um programa que recebe um valor inteiro positivo `N` como argumento de linha de comando e informa se `N` faz parte de alguma sequência de Fibonacci. Se `N` não for um valor válido para o programa ou estiver ausente, o programa deve responder adequadamente.

EP3.16 (a) Escreva uma função, denominada `InverteString()`, que copia um string invertido (segundo parâmetro) para um array (primeiro parâmetro). (b) Escreva um programa que lê strings via teclado e apresenta-os invertidos na tela.

EP3.17 (a) Escreva uma função, denominada `OcorrenciasStr()`, que conta o número de ocorrências de um string (primeiro parâmetro) em outro string (segundo parâmetro). (b) Escreva um programa para testar a função solicitada no item (a).

EP3.18 **Preâmbulo:** **Rotação à direita de ordem** `k` de um array com `n` elementos (`k < n`) consiste em mover cada elemento do array `k` posições adiante, de tal modo que o último elemento passe a ocupar a posição `k - 1`, o penúltimo elemento passe a ocupar a posição `k - 2` e assim por diante. Se `k = n`, a rotação à direita de ordem `k` é equivalente à rotação à direita de ordem `k%n`. Essa equivalência também vale quando `k < n`, pois, nesse caso, `k%n` é igual a `k`. **Rotação à esquerda de ordem** `k` de um array é definida de modo análogo, mas, agora, os elementos são deslocados para trás. **Problema:** (a) Escreva uma função que provoca a rotação dos elementos de um array um número determinado de vezes para a direita (rotação positiva) ou para a esquerda (rotação negativa). (b) Escreva um programa que define um array e, repetidamente, apresenta-o antes e depois sofrer as rotações especificadas pelo usuário.

Exemplo de execução do programa:

```
>>> Estado atual do array:
{ 1, 2, 3, 4, 5 }
>>> Numero de rotacoes (0 encerra o programa): 3
>>> Estado atual do array:
{ 3, 4, 5, 1, 2 }
>>> Numero de rotacoes (0 encerra o programa): -8
>>> Estado atual do array:
{ 1, 2, 3, 4, 5 }
>>> Numero de rotacoes (0 encerra o programa): 23
>>> Estado atual do array:
{ 3, 4, 5, 1, 2 }
>>> Numero de rotacoes (0 encerra o programa): 0
```

[**Sugestões:** (1) Existem diversos algoritmos para rotação de arrays e um dos mais fáceis de entender e implementar usa um array auxiliar para armazenar os elementos do array que está passando por uma rotação. Assim defina um array auxiliar local à função que implementa rotações para copiar os elementos do array original em suas novas posições. Antes de retornar, essa função deve copiar o conteúdo do array auxiliar para o array original. (2) Se o valor de `k` for negativo, representando uma rotação à esquerda, converta-o numa rotação à direita substituindo esse valor por: `n + k%n`.]

EP3.19 Use estruturas do tipo:

```
typedef struct {
           double   valor;
           tUnidade unidade;
        } tMedida;
```

para representar medidas de comprimento, no qual o tipo `tUnidade` representa as unidades de medição: centímetro, metro, polegada e pé. Esse tipo é definido antes do tipo `tMedida` como:

```
typedef enum {CENTIMETRO, METRO, POLEGADA, PE} tUnidade;
```

Escreva uma função, cujo protótipo seja:

```
double SomaMedidas(const tMedida medidas[], int nMedidas, tUnidade unidade)
```

que calcula a soma das medidas armazenadas num array de elementos do tipo `tMedida` na unidade especificada no parâmetro `unidade`.

Dados:

- 1 polegada equivale a 2,54 cm
- 1 pé equivale a 12 polegadas
- 1 metro equivale a 3,28 pés