Capítulo 10

Estruturas, Uniões e Enumerações

Após estudar este capítulo, você deverá ser capaz de:

- Definir os seguintes conceitos:
  - ❒ Estrutura
  - ❒ Definição de tipo
  - ❒ Rótulo de estrutura
  - ❒ Campo de estrutura
  - ❒ Operador ponto
  - ❒ Operador seta
  - ❒ União
  - ❒ Definidor de tipo
  - ❒ Estrutura aninhada
  - ❒ Operador de acesso
  - ❒ Estrutura com autorreferência
  - ❒ Enumeração
  - ❒ Registro variante
  - ❒ Campo indicador
  - ❒ Iniciador designado
- Descrever e usar as seguintes palavras-chave da linguagem C:
  - ❒ **typedef** ❒ **struct** ❒ **union** ❒ **enum**
- Dizer como são iniciados e acessados os membros (campos) de uma estrutura ou união
- Apresentar diferenças entre estruturas e arrays
- Mostrar que é inadequado definir tipos usando **#define** em vez de **typedef**
- Especificar os definidores de tipos disponíveis em C
- Descrever os operadores de acesso `[]`, `()`, `*`, `.` (ponto) e `->` juntamente com suas precedências e associatividades
- Explicar como uma estrutura pode ser declarada e passada como parâmetro para uma função
- Decidir quando se deve usar **const** na definição de um ponteiro para estrutura como parâmetro de uma função
- Enumerar semelhanças e diferenças entre uniões e estruturas
- Identificar situações em que uniões são úteis
- Reforçar a importância de uso de campo indicador em registro variante
- Saber usar enumeração num programa quando a situação assim indicar
- Conhecer as regras de atribuição de valores a constantes de enumerações

OBJETIVOS

10.1 Introdução

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**.

Uniões são variáveis estruturadas e heterogêneas semelhantes a estruturas, mas diferem de estruturas pelo fato de seus campos serem compartilhados em memória.

Apesar das semelhanças com estruturas e uniões na forma como são definidas, **enumerações** não são variáveis estruturadas e foram incluídas neste capítulo apenas por causa dessas semelhanças.

10.2 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 dados, 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`). Qualquer dessas terminologias é inteiramente satisfatória.

Definições de tipos são usadas frequentemente não apenas para identificar tipos estruturados construídos pelo programador, conforme será visto neste capítulo, como também para atribuir novos nomes a tipos primitivos. Nesse último caso, a finalidade é primar pela legibilidade e, principalmente, pela portabilidade de programas. A biblioteca padrão de C contém várias definições de tipo dessa natureza, como, por exemplo, o tipo **size_t**, apresentado na Seção 8.5.

10.3 Estruturas

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 (v. Seção 3.2).

10.3.1 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 {
            tipo1  campo1;
            tipo2  campo2;
            ...
            tipoN campoN;
          } nome-do-tipo;
```

Aqui, define-se um tipo que é sinônimo daquilo que precede sua definição (v. **Seção 10.2**), incluindo a palavra **struct**. O rótulo da estrutura é um identificador opcional que é necessário apenas quando se declara uma estrutura com auto-referência ou opaca, que não é assunto abordado neste livro. Como exemplos de definições de estruturas considere:

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

tRegistro  registroDaPessoa, *ptrParaRegistro;
```

Nesse exemplo, são definidos o tipo `tRegistro` e as variáveis `registroDaPessoa` e `ptrParaRegistro`.

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; /* Não se deve mais usar asterisco aqui */
```

Observe no último exemplo que, na definição da variável `ptrParaRegistro`, não se deve utilizar asterisco, pois o tipo `tPtrParaRegistro` já é um tipo de ponteiro. Caso contrário, estar-se-ia declarando um ponteiro para ponteiro.

Dois campos que fazem parte de estruturas diferentes podem possuir o mesmo nome, como, por exemplo:

```
typedef struct {
      int    a;
      double b;
}  tEstrutura1;

typedef struct {
      int   a;
      char  b;
}  tEstrutura2;
```

Devido à forma como os campos de uma estrutura são acessados (v. **Seção 10.3.4**), não existe chance de colisão dos identificadores de campos nesse exemplo.

Em termos de estilo, é recomendado endentar as declarações de campos de uma estrutura para melhor destacá-los, conforme é demonstrado nos diversos exemplos apresentados neste livro.

10.3.2 Iniciações

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};
```

Não se pode iniciar uma definição de tipo, pois tal definição não aloca espaço em memória. Por exemplo, a seguinte tentativa de iniciação seria inválida:

```
typedef struct {
           char  nome[30];
           int  dia, mes, ano;
        } tRegistro = {"Jose da Silva", 12, 10, 1960};/* ILEGAL */
```

10.3.3 Atribuições

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

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

as seguintes atribuições são perfeitamente válidas:

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

10.3.4 Acesso a Campos

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. Esse operador é representado pelo símbolo de menos seguido pelo símbolo de maior do que e sem espaço entre eles.

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 10–1**.

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 10–1: Acesso a Campos de uma Estrutura

Na realidade, 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
```

Quando acessado, um campo de uma estrutura é considerado como uma variável comum do tipo usado para declará-lo. Em outras palavras, pode-se utilizar um campo acessado em qualquer local onde uma variável do tipo desse campo poderia ser utilizada. Por exemplo, pode-se alterar o campo **ano** da variável **registroDaPessoa** fazendo-lhe uma atribuição como:

```
registroDaPessoa.ano = 1966;
```

ou, equivalentemente:

```
ptrParaRegistro->ano = 1966;
```

10.3.5 Aninho

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**:

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

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. Também, apesar de aparentemente mais complexa, a definição de **tRegistro2** é mais elegante, pois denota uma melhor organização de dados que facilita a

legibilidade e o reúso de código. Por exemplo, o tipo **tData** poderia ser reutilizado na definição de um outro tipo de estrutura, como mostra a definição de tipo a seguir:

```
typedef struct {
           char   codigo[5];
           tData  vencimento;
        } tPromissoria;
```

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 10.4**).

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

```
ptrParaRegistro->nascimento.dia
```

Note, nesse último exemplo, que o operador **->** é utilizado apenas uma vez, visto que o campo **nascimento** é uma estrutura e não um ponteiro para estrutura, como é o caso de **ptrParaRegistro**. O seguinte programa ilustra o que foi exposto:

```
#include <stdio.h>

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

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

int main(void)
{
   tRegistro2 pessoa = { "Jose da Silva", {12, 10, 1960} };
   tRegistro2 *ptrPessoa = &pessoa;

   printf("\n\n***** Usando pessoa *****\n\n");

   printf("Nome: %s\n", pessoa.nome);
   printf( "Nascimento: %d/%d/%d\n", pessoa.nascimento.dia,
```

```
            pessoa.nascimento.mes, pessoa.nascimento.ano );

    printf("\n***** Usando ptrPessoa *****\n\n");

    printf("Nome: %s\n", ptrPessoa->nome);
    printf( "Nascimento: %d/%d/%d\n", ptrPessoa->nascimento.dia,
            ptrPessoa->nascimento.mes, ptrPessoa->nascimento.ano );

    return 0;
}
```

Quando executado, o programa acima produz o seguinte resultado:

```
***** Usando pessoa *****

Nome: Jose da Silva
Nascimento: 12/10/1960

***** Usando ptrPessoa *****

Nome: Jose da Silva
Nascimento: 12/10/1960
```

Em princípio, não existe limite quanto ao número de níveis de aninho de estruturas. No entanto, o uso de muitos níveis de aninho pode prejudicar a legibilidade não apenas da definição da estrutura como também das expressões utilizadas para acessar os campos mais internos.

10.4 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 5.2.3**) são denominados **operadores de acesso**. Para facilidade de referência, a **Tabela 10–2** 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 10–2: 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 (i.e., `.` e `->`), os demais símbolos de operadores de acesso mais as palavras-chave **struct**, **union** e **enum** (v. **Seção 10.9**) 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 10–3** resume esses definidores de tipo.

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

Tabela 10–3: Definidores de Tipo

10.5 Uso de Estruturas em Funções

10.5.1 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:

- O parâmetro é declarado como uma estrutura.
- 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 );
}
```

Considerando as definições de `ExibeRegistro1()` e `ExibeRegistro2()`, se a variável `pessoa` fosse declarada como:

```
tRegistro  pessoa;
```

então, as funções `ExibeRegistro1()` e `ExibeRegistro2()` poderiam ser chamadas como:

```
ExibeRegistro1(pessoa);
```

e

```
ExibeRegistro2(&pessoa);
```

Por outro lado, considerando a definição de ponteiro abaixo:

```
tRegistro *ptrPessoa = &pessoa;
```

as chamadas de `ExibeRegistro1()` e `ExibeRegistro2()` deveriam ser:

```
ExibeRegistro1(*ptrPessoa);
```

e

```
ExibeRegistro2(ptrPessoa);
```

Em ambas as situações ilustradas, `ExibeRegistro1()` sempre recebe como parâmetro real uma estrutura e `ExibeRegistro2()` sempre recebe um endereço de estrutura como parâmetro real.

Quando uma estrutura é passada como parâmetro para uma função, qualquer alteração desse parâmetro incide sobre uma cópia da estrutura, de modo que a estrutura usada como parâmetro real permanece inalterada após a chamada. Por outro lado, quando o endereço de uma estrutura é passado como parâmetro para uma função, qualquer alteração de valor de algum campo da estrutura ocorrida no corpo da função incide sobre a própria estrutura.

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 é 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.

O método escolhido para declaração de uma estrutura como parâmetro de uma função determina o tipo de operador que será utilizado no corpo da função para acesso aos campos da estrutura. Ou seja, se uma estrutura for declarada como parâmetro de função, o operador ponto deve ser utilizado no acesso aos seus campos no corpo da função. Por outro lado, se um ponteiro para estrutura for declarado como parâmetro de função, o operador seta deve ser usado em acessos aos campos da estrutura para a qual o ponteiro aponta.

10.5.2 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. Por exemplo, a função `LeRegistro1()` apresentada a seguir retorna uma estrutura do tipo `tRegistro` definido anteriormente (`MAX_NOME` é uma constante simbólica previamente definida):

```
tRegistro LeRegistro1(void)
{
   tRegistro registro;

   printf("\nNome (max = %d letras): ", MAX_NOME - 1);
   LeString(registro.nome, MAX_NOME);

   printf("\nDia de nascimento: ");
   registro.dia = LeInteiro();
```

```
   printf("\nMes de nascimento: ");
   registro.mes = LeInteiro();

   printf("\nAno de nascimento: ");
   registro.ano = LeInteiro();

   return registro;
}
```

Por outro lado, função `LeRegistro2()` abaixo retorna o endereço de uma estrutura do tipo `tRegistro`:

```
tRegistro *LeRegistro2(tRegistro *ptrRegistro)
{
   printf("\nNome (max = %d letras): ", MAX_NOME - 1);
   LeString(ptrRegistro->nome, MAX_NOME);

   printf("\nDia de nascimento: ");
   ptrRegistro->dia = LeInteiro();
   printf("\nMes de nascimento: ");
   ptrRegistro->mes = LeInteiro();

   printf("\nAno de nascimento: ");
   ptrRegistro->ano = LeInteiro();

   return ptrRegistro;
}
```

Aparentemente, a função `LeRegistro2()` poderia ser definida de modo semelhante à função `LeRegistro1()` como:

```
tRegistro *LeRegistro3(void)
{
   tRegistro registroZumbi;

   printf("\nNome (max = %d letras): ", MAX_NOME - 1);
   LeString(registroZumbi.nome, MAX_NOME);

   printf("\nDia de nascimento: ");
   registroZumbi.dia = LeInteiro();

   printf("\nMes de nascimento: ");
   registroZumbi.mes = LeInteiro();

   printf("\nAno de nascimento: ");
   registroZumbi.ano = LeInteiro();

   return &registroZumbi; /* Retorno de zumbi */
}
```

Ocorre, porém, que, se a função `LeRegistro2()` tivesse sido definida dessa maneira, ao final ela estaria retornando um zumbi, conforme foi discutido na Seção 8.9.4. Portanto, se a intenção do programador for retornar o endereço de uma variável local a uma função, essa variável deverá ter duração fixa, como mostrado a seguir:

```
tRegistro *LeRegistro4(void)
{
   static tRegistro registro;

   printf("\nNome (max = %d letras): ", MAX_NOME - 1);
   LeString(registro.nome, MAX_NOME);

   printf("\nDia de nascimento: ");
   registro.dia = LeInteiro();

   printf("\nMes de nascimento: ");
```

```
   registro.mes = LeInteiro();

   printf("\nAno de nascimento: ");
   registro.ano = LeInteiro();

   return &registro;
}
```

Essa última função resolve parcialmente o problema da função anterior, mas ainda não representa uma solução adequada porque o conteúdo da variável cujo endereço é retornado será sobrescrito a cada chamada subsequente da função. Por exemplo, se essa função for chamada como na função **main()** a seguir:

```
int main(void)
{
   tRegistro *ptrPessoa1, *ptrPessoa2;

   ptrPessoa1 = LeRegistro4();

   printf("\n\n***** Dados da primeira pessoa *****\n\n");
   ExibeRegistro(ptrPessoa1);

   ptrPessoa2 = LeRegistro4();

   printf("\n\n***** Dados da segunda pessoa *****\n\n");
   ExibeRegistro(ptrPessoa2);

   return 0;
}
```

os resultados apresentados pelo programa seriam corretos. Mas, uma ligeira alteração efetuada nessa função **main()** produziria um resultado indesejado:

```
int main(void)
{
   tRegistro *ptrPessoa1, *ptrPessoa2;

   ptrPessoa1 = LeRegistro4();
   ptrPessoa2 = LeRegistro4();

   printf("\n\n***** Dados da primeira pessoa *****\n\n");
   ExibeRegistro(ptrPessoa1);

   printf("\n\n***** Dados da segunda pessoa *****\n\n");
   ExibeRegistro(ptrPessoa2);

   return 0;
}
```

Nesse último exemplo, o programa exibiria os dados da segunda estrutura lida duas vezes. Isso ocorre porque os dados da última estrutura lida sobrescrevem os dados da leitura anterior.

Resumindo, a melhor versão apresentada acima de função que lê dados para os campos de uma estrutura e retorna o endereço da estrutura lida é `LeRegistro2()`. Note, entretanto, que qualquer das funções apresentadas nesta seção é destinada a usuários *muito bem comportados*. Isto é, elas não testam os dados introduzidos pelo usuário, de modo que ele pode introduzir valores sem sentido. Essa verificação de dados de entrada não foi realizada aqui para não desviar o foco da discussão.

Pela mesma razão que passar o endereço de uma estrutura como parâmetro é mais eficiente que passar a própria estrutura (v. **Seção 10.5.1**), retornar o endereço de uma estrutura é mais eficiente e, portanto, mais utilizado do que retornar uma estrutura em si.

10.6 Uniões

Uniões são tipos de dados similares às estruturas, mas os campos de uma união compartilham o mesmo espaço 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. Por exemplo, se os tipos **int** e **double** ocuparem, respectivamente, dois e quatro bytes, a alocação da variável `quantidade` contendo os campos `unidades` e `peso` do último exemplo poderia ser representada esquematicamente como na Figura 10–1. Cada retângulo nessa figura representa um byte.

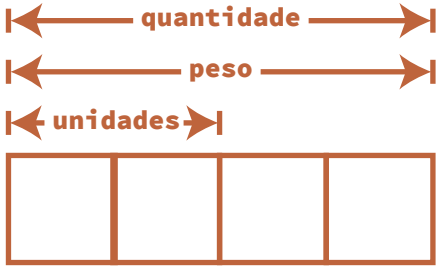

FIGURA 10–1: CAMPOS COMPARTILHADOS DE UMA UNIÃO

Para efeito de comparação, se a variável `quantidade2` for definida como uma estrutura com os mesmos campos que a variável `quantidade`:

```
typedef struct {
           int    unidades;
           double peso;
        } tQuantidade2;

tQuantidade2 quantidade2;
```

ela pode ser representada esquematicamente como na Figura 10–2.

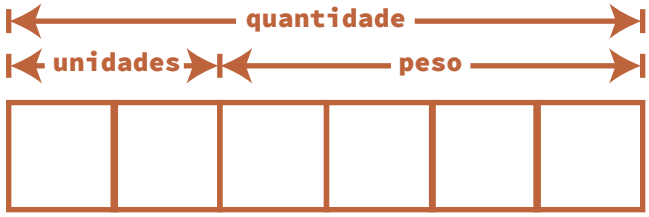

FIGURA 10–2: CAMPOS EXCLUSIVOS DE UMA ESTRUTURA

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 como uma união:

```
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, como mostra o programa a seguir:

```
#include <stdio.h>

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

int main(void)
{
   tQuantidade quantidade;

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

   printf("\nQuantidade: %d unidades\n", quantidade.unidades);

   return 0;
}
```

Quando executado, esse programa produz o seguinte resultado, que é desprovido de significado:

```
Quantidade: 0 unidades
```

Uma união pode ser iniciada atribuindo-se um valor entre chaves ao seu primeiro campo. Por exemplo:

```
typedef union {
           int    unidades;
           double peso;
        } tQuantidade;
tQuantidade  quantidade = {2};
```

Iniciar uma união com um valor de um tipo diferente do tipo de seu primeiro campo produz um resultando inesperado no programa, como mostra o seguinte programa:

```
#include <stdio.h>

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

int main(void)
{
   tQuantidade quantidade = {1.5};

   printf("\nQuantidade: %3.2f Kg", quantidade.peso);
   printf("\nQuantidade: %d unidades\n", quantidade.unidades);

   return 0;
}
```

O resultado produzido por esse programa não faz sentido:

```
Quantidade: 0.00 Kg
Quantidade: 1 unidades
```

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

typedef struct {
           char        descricao[20];
           double      precoUnitario;
           int         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 == 1) { /* Unidade */
      printf( "Quantidade: %d unidades\n", produto->quantidade.unidades );
   } else if (produto->tipoDeQuantidade == 2) { /* Peso */
      printf("Quantidade: %3.2f Kg\n", produto->quantidade.peso);
   } else { /* Desconhecida */
      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, 2 };
   tProduto produto2 = { "Coco Verde", 2.50, 1, {2} };
   tProduto produto3 = { "Jaca Dura", 5.49, 3, {1} };

   produto1.quantidade.peso = 1.5;

   ExibeProduto(&produto1);
   ExibeProduto(&produto2);
   ExibeProduto(&produto3);

   return 0;
}
```

O resultado apresentado pelo programa acima, quando ele é executado, é:

```
Produto: Arroz Arboreo Preco: R$12.99 Quantidade: 1.50 Kg
Produto: Coco Verde    Preco: R$1.50  Quantidade: 2 unidades
Produto: Jaca Dura     Preco: R$5.49
ESTE PROGRAMA CONTEM UM ERRO:
        O valor de produto->tipoDeQuantidade
        nao poderia ser 3
```

As últimas três linhas exibidas pelo último programa foram escritas pela função `ExibeProduto()` logo antes de abortar voluntariamente o programa. Essa função assim procede porque o campo `tipoDeQuantidade` da variável `produto3`, que ela recebe como parâmetro, assume um valor que não deveria assumir. Pode parece estranho à primeira vista que um programa seja abortado voluntariamente, mas tomar essa atitude aparentemente radical é melhor do que permitir que a execução do programa prossiga e, consequentemente, que o erro se alastre.

É importante observar que, se o último programa precisasse ser abortado no corpo da função **main**(), bastaria usar uma instrução **return**. Mas, para abortar um programa durante a execução de outra função, como é o caso aqui, é necessário chamar **exit**(), que é uma função da biblioteca padrão de C declarada em `<stdlib.h>`. O parâmetro recebido por essa função tem a mesma interpretação do valor retornado por **main**(). Isto é, ele indica se o programa foi bem sucedido ou não. Normalmente, esse parâmetro é um valor diferente de zero, porque, na maioria das vezes, essa função é chamada quando ocorre algum tipo de irregularidade detectada pelo programa que impede seu funcionamento normal.

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, mas seu uso é um tanto complicado de decifrar. Isto é, esse campo assume o valor `1` quando o primeiro campo da união `quantidade` (`unidades`) é usado e `2` quando o segundo campo dessa união (`peso`) está em vigor. Definir esse campo como uma enumeração tornaria o papel desempenhado por ele mais fácil de entender, como mostra o programa a seguir.

```
#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 ExibeProduto2(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;

   ExibeProduto2(&produto1);
   ExibeProduto2(&produto2);
   ExibeProduto2(&produto3);

   return 0;
}
```

No programa acima, o campo `tipoDeQuantidade` é uma enumeração do tipo `tTipoDeQuantidade`. Isso significa que essa variável deve apenas assumir um dos valores que fazem parte da definição de seu tipo. (Enumerações serão precisamente descritas na **Seção 10.9**.) A execução desse programa produz exatamente o mesmo resultado do programa anterior, mas é bem mais fácil de entender.

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.

10.8 Iniciadores Designados

Os padrões mais recentes de C (i.e., ISO C99 e C11) permitem o uso de iniciadores designados para estruturas e uniões. Um **iniciador designado** de uma estrutura permite especificar (i.e., designar) quais campos se desejam iniciar. No caso de uniões, como os campos são mutuamente exclusivos, faz sentido especificar apenas um campo para iniciação. Um iniciador designado de estrutura ou união é escrito começando-se por `.` (ponto), seguido do nome do campo que se deseja iniciar, seguido do símbolo de igualdade e, finalmente, seguido do valor que se deseja atribuir ao campo. Ou seja, esquematicamente, um iniciador designado de estrutura ou união é escrito assim:

.nome-do-campo = *valor*

Iniciadores designados devem ser envolvidos entre chaves e podem aparecer em conjunto com iniciadores convencionais (i.e., aqueles vistos nas **Seções 10.3.2** e **10.6**), como mostra o programa abaixo:

```
#include <stdio.h>

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

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

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

int main(void)
{
   tRegistro2 pessoa = { "Jose da Silva",
                         .nascimento.ano = 1960 };
   tQuantidade quantidade = {.peso = 1.5};

   pessoa.nascimento.dia = 12;
   pessoa.nascimento.mes = 10;

   printf("\n\n***** Usando pessoa *****\n\n");

   printf("Nome: %s\n", pessoa.nome);
   printf( "Nascimento: %d/%d/%d\n", pessoa.nascimento.dia,
           pessoa.nascimento.mes, pessoa.nascimento.ano );

   printf("\n\n***** Usando quantidade *****\n\n");
   printf("Quantidade: %3.2f Kg\n", quantidade.peso);

   return 0;
}
```

O campo **ano** da estrutura aninhada **nascimento** foi iniciado como:

```
tRegistro2 pessoa = {"Jose da Silva", .nascimento.ano = 1960};
```

mas, nesse mesmo exemplo, ele também poderia ter sido iniciado assim:

```
tRegistro2 pessoa = { "Jose da Silva", {.ano = 1960} };
```

Contudo, essa última iniciação só é possível porque não há nenhum outro campo entre **nome** e **nascimento**. Em qualquer caso, o formato de iniciação visto na **Seção 10.3.2** é preferível, não apenas porque é sempre válido, como também porque é mais legível.

Arrays também aceitam o uso de iniciadores designados, que permite que elementos específicos de um array sejam iniciados, conforme mostrado no seguinte exemplo:

```
int ar[20] = {[4] = -2, [9] = 5, [12] = 1, [17] = -5};
```

Nesse exemplo, os elementos **ar[4]**, **ar[9]**, **ar[12]** e **ar[17]** são explicitamente iniciados, enquanto os elementos remanescentes são iniciados implicitamente com **0**, conforme foi visto na **Seção 8.4**.

Deve-se ainda ressaltar que a ordem com que os elementos são iniciados é irrelevante. Por exemplo, a iniciação a seguir:

```
int ar[20] = {[12] = 1, [9] = 5, [17] = -5, [4] = -2};
```

tem o mesmo efeito daquela apresentada no exemplo anterior. Em termos de legibilidade, entretanto, a primeira iniciação é mais recomendada.

10.9 Enumerações

Uma variável de um tipo **enumeração** deve assumir apenas um dos valores constantes especificados na definição do seu tipo. Em outras palavras, um tipo enumeração limita os valores que uma variável desse tipo pode assumir a um conjunto de constantes com alguma afinidade entre si. Quando se tenta atribuir a uma variável de um tipo enumeração um valor que não faz parte desse conjunto de constantes, o compilador pode emitir uma mensagem de advertência.

Esquematicamente, uma definição de tipo enumeração tem o seguinte formato:

typedef enum *{lista-de-nomes-de-constantes}* ***tipo;***

Considere o seguinte fragmento de programa como exemplo de uso de enumeração:

```
typedef enum  { AZUL, VERMELHO, PRETO, BRANCO } tCores;
...
tCores  umaCor = VERMELHO,
        outraCor = BRANCO;
```

Note que a mesma recomendação de nomenclatura para constantes simbólicas (v. **Seção 6.5**) é seguida para constantes que fazem parte de uma enumeração; i.e., elas são escritas usando apenas letras maiúsculas.

Apesar de apresentarem alguma semelhança com estruturas e uniões com relação ao modo como seus tipos são definidos, variáveis de um tipo enumeração *não são estruturadas*, como mostra o último exemplo. Ou seja, as variáveis `umaCor` e `outraCor` do último exemplo podem assumir um dos valores constantes `AZUL`, `VERMELHO`, `PRETO` ou `BRANCO`, mas não podem assumir dois ou mais desses valores simultaneamente.

As constantes simbólicas que fazem parte de uma enumeração são consideradas do tipo **int** e valores inteiros são associados a elas. Caso não haja indicação em contrário, esses valores são baseados nas posições das constantes na lista, sendo que, como padrão, a primeira constante vale `0`, a segunda constante vale `1` e assim por diante. No último exemplo, `AZUL` vale `0`, `VERMELHO` vale `1`, `PRETO` vale `2` e `BRANCO` vale `3`.

Tipicamente, os valores atribuídos aos identificadores de constantes de uma enumeração não são importantes, mas permite-se que eles sejam explicitamente especificados. Além disso, se o valor de uma determinada constante de enumeração não for definido explicitamente, ele assumirá o valor da constante anterior na sequência acrescido de `1`. Como exemplo, considere:

```
typedef enum  { AZUL = -3, VERMELHO, PRETO = 20, BRANCO } tCores;
```

Nesse exemplo, as constantes `VERMELHO` e `BRANCO` terão valores `-2` e `21`, respectivamente. Note que não se requer que as atribuições de constantes sejam feitas em ordem crescente, como no exemplo acima, mas, em nome da boa legibilidade, isso é recomendado. Além disso, duas ou mais constantes de uma enumeração podem assumir um mesmo valor.

É importante salientar que constantes de enumerações têm escopo de arquivo. Portanto não é permitido declarar nenhum identificador que tenha o mesmo nome de uma constante de enumeração com esse tipo de escopo. O exemplo abaixo ilustra esse ponto:

```
#include <stdio.h>

typedef enum {VERMELHO, BRANCO} tColorido;
// typedef enum {AZUL, BRANCO} tCores; /* Não compila */

// int BRANCO = 10; /* Não compila */

int main(void)
```

```
{
   int BRANCO = 5; /* Oculta a constante BRANCO */
   printf("BRANCO = %d", BRANCO); /* Escreve BRANCO = 5 */
   return 0;
}
```

Nesse exemplo, se forem removidos os comentários delimitados com //, o programa não conseguirá ser compilado. Ou seja, mais precisamente, se o comentário // da linha:

```
// typedef enum {AZUL, BRANCO} tCores; /* Não compila */
```

for removido, haverá colisão entre as duas constantes que usam o mesmo identificador **BRANCO**. Por outro lado, se for removido o comentário // da linha:

```
// int BRANCO = 10; /* Não compila */
```

haverá colisão entre a variável e a constante da enumeração que usam o mesmo identificador **BRANCO**. Entretanto, não ocorre colisão de identificadores na definição da variável **BRANCO** no corpo da função **main**():

```
int BRANCO = 5;
```

pois o identificador dessa variável e a constante **BRANCO** da enumeração fazem parte de escopos diferentes. O que ocorre nesse caso é sobreposição de escopos, de maneira que o identificador usado com a variável oculta aquele da constante (v. Seção 5.10.5).

O exemplo a seguir, demonstra um uso correto de enumerações:

```
#include <stdio.h>

typedef enum {AZUL = -3, VERDE, PRETO = 20, BRANCO} tCores;

int main(void)
{
   tCores umaCor = VERDE;
   int    outraCor = BRANCO;

   printf("\numaCor = %d\noutraCor = %d\n", umaCor, outraCor);

   return 0;
}
```

Esse programa apresenta o seguinte resultado na tela:

```
umaCor = -2
outraCor = 21
```

Um parâmetro de função pode ser de um tipo enumeração e uma função também pode retornar um valor de um tipo enumeração, como mostra o seguinte programa:

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

typedef enum  { AZUL, VERDE, PRETO, BRANCO } tCores;
typedef enum  { ALCOOL, GASOLINA, DIESEL } tCombustivel;

typedef struct {
           char         modelo[20];
           double        potencia;
           int          portas;
           tCores       cor;
           tCombustivel combustivel;
       } tAutomovel;
```

```
void ExibeCor(tCores cor)
{
      /* Exibe a cor correspondente a cada constante */
      /* da enumeração recebida como parâmetro       */
   switch (cor) {
      case AZUL:
         puts("azul");
         break;
      case VERDE:
         puts("verde");
         break;
      case PRETO:
         puts("preta");
         break;
      case BRANCO:
         puts("branca");
         break;
      default:
            /* Aborta o programa se a execução chegar a este ponto */
         puts("Este programa contém um erro!");
         exit(1);
         break;
   }
}

tCores CorAutomovel(const tAutomovel *carro)
{
   return carro->cor;
}

int main(void)
{
   tAutomovel seuCarro = {"Fusca", 40.0, 2, VERDE, ALCOOL};
   tAutomovel meuCarro = {"Civic Si", 193.0, 2, PRETO, GASOLINA};
   tCores     umaCor;

   printf("Cor do seu carro: ");

      /* Obtém um valor associado a uma constante da enumeração tCores */
   umaCor = CorAutomovel(&seuCarro);

      /* Exibe na tela o string associado a uma constante da enumeração tCores */
   ExibeCor(umaCor);

   printf("Cor do meu carro: ");

      /* Exibe na tela o string associado a uma constante da enumeração tCores */
   ExibeCor(meuCarro.cor);

   return 0;
}
```

O resultado da execução do último programa é:

```
Cor do seu carro: verde
Cor do meu carro: preta
```

Às vezes, enumerações são utilizadas para indexar arrays, como mostra o exemplo a seguir:

```
#include <stdio.h>
```

```
typedef enum  { JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO,
                SET, OTU, NOV, DEZ } tMeses;

int main(void)
{
   tMeses      mes;
   const char *nomeMes[] = { "Janeiro", "Fevereiro", "Marco",
                             "Abril", "Maio", "Junho", "Julho",
                             "Agosto", "Setembro", "Outubro",
                             "Novembro", "Dezembro" };

   for (mes = JAN; mes <= DEZ; mes++) {
      printf("%2.2d - %s\n", mes + 1, nomeMes[mes]);
   }

   return 0;
}
```

Quando o último programa é executado, ele produz como resultado:

```
01 - Janeiro
02 - Fevereiro
03 - Marco
04 - Abril
05 - Maio
06 - Junho
07 - Julho
08 - Agosto
09 - Setembro
10 - Outubro
11 - Novembro
12 - Dezembro
```

Enumerações são excelentes quando usadas para melhorar a legibilidade de um programa, mas sua utilidade prática é limitada a esse âmbito. Isto é, o padrão ISO de C não requer que um compilador sequer emita mensagens de advertência sobre usos indevidos de enumerações, como mostra o programa a seguir:

```
#include <stdio.h>

typedef enum  { AZUL, VERDE, PRETO, BRANCO } tCores;

int main(void)
{
   tCores umaCor = VERDE;
   int    i = BRANCO;

   umaCor = 12;

   printf("\numaCor = %d\ni = %d\n", umaCor, i);

   return 0;
}
```

Observe nesse programa que há duas violações do conceito de enumeração:

[1] A variável `i` é do tipo **int** (e não do tipo `tCores`), mas, mesmo assim, lhe é permitido assumir um valor do tipo `tCores`. Isso ocorre porque constantes de enumerações são inteiras e, portanto, compatíveis com o tipo da variável `i`.

[2] A variável `umaCor`, que é do tipo `tCores`, tem permissão para assumir `12` como valor, que sequer coincide com o valor de qualquer das constantes simbólicas que constituem o tipo `tCores`. Esse problema é bem mais sério do que o anterior e compiladores não deveriam negligenciá-lo.

Quando compilado usando o compilador GCC 4.6.1, que segue quase integralmente o padrão ISO C99, esse último programa não origina nenhuma mensagem de erro ou advertência. Portanto o compilador deixa a tarefa de uso correto de enumerações a cargo único e exclusivo do programador. A execução desse programa também ocorre normalmente e escreve o seguinte na tela:

```
umaCor = 12
i = 3
```

Concluindo, o conselho a ser seguido com respeito ao uso de enumerações é:

Recomendação *Use enumerações conforme recomendado e com bastante cuidado, pois, caso contrário, um compilador não irá acudi-lo.*

10.10 Exemplos de Programação

10.10.1 Processando Dados de um Aluno

Problema: Suponha que, num programa escrito em C, dados de alunos são armazenados em estruturas do seguinte tipo:

```
typedef struct {
           char   nome[MAX_NOME + 1]; /* Nome do aluno */
           char   matr[TAM_MATR + 1]; /* Matrícula     */
           double n1, n2;             /* Notas 1 e 2   */
       } tAluno;
```

Nessa definição de tipo, `MAX_NOME` e `TAM_MATR` são constantes simbólicas previamente definidas. (a) Escreva uma função que lê dados de um aluno via teclado, de tal modo que um dos parâmetros informe se a matrícula deve ser lida ou não. (b) Escreva uma função que apresenta na tela dados de um aluno. (c) Escreva um programa que teste as duas funções solicitadas nos itens (a) e (b).

Solução de (a): Uma função para leitura dos dados de uma estrutura do tipo `tAluno` deve realizar as seguintes tarefas (não necessariamente nesta ordem):

- Ler um nome, como faz a função `LeNome()` apresentada na Seção 9.10.1.
- Ler uma matrícula, como faz a função `LeIdentidade()` apresentada na Seção 9.10.2.
- Ler duas notas. A função `LeNota()`, apresentada na Seção 7.6.1, realiza leitura de notas.

Portanto a escrita de uma função para leitura dos dados de uma estrutura do tipo `tAluno` é trivial, desde que você não tenha que reinventar a roda; caso contrário, sua tarefa será bastante penosa. Isto é, se as funções descritas acima estiverem disponíveis, como é o caso, a tarefa será restrita a praticamente incluir essas funções e chamá-las para ler os campos da estrutura.

Usando as funções mencionadas acima, a função `LeDadosAluno()`, que efetua a leitura requerida, pode ser escrita como:

```
/****
 * LeDadosAluno(): Lê os dados de um aluno
 *
 * Parâmetros:
 *     aluno (saída) - ponteiro para a estrutura que receberá os
 *                     dados introduzidos pelo usuário
 *     fMatr (entrada) - indica se o campo 'matr' será lido ou não
 *
```

```
 * Retorno: Endereço da estrutura que armazena os dados lidos
 *
 * Observação: O parâmetro 'fMatr' é uma flag que determina se a matrícula deve ser
 *             lida ('fMatr' diferente de zero) ou não ('fMatr' igual a zero). A
 *             justificativa para o uso desse parâmetro é que, algumas vezes, quando
 *             esta função é chamada, a matrícula já terá sido lida. Essa alterna-
 *             tiva é melhor do que solicitar novamente a matrícula ao usuário
 *             (inconveniente) ou escrever duas funções que diferem apenas pelo
 *             fato de solicitarem matrícula ou não.
 ****/
tAluno *LeDadosAluno(tAluno *aluno, int fMatr)
{
   LeNome(aluno->nome, MAX_NOME + 1); /* Lê o nome */

      /* Se indicado pelo parâmetro 'fMatr', lê a matrícula */
   if (fMatr) {
      LeIdentidade(aluno->matr, TAM_MATR+1, "Digite a matricula");
   }

      /* Lê a primeira nota */
   printf("\n\t>>> 1a. nota:\n\t    > ");
   aluno->n1 = LeNota();

      /* Lê a segunda nota */
   printf("\n\t>>> 2a. nota:\n\t    > ");
   aluno->n2 = LeNota();

   return aluno;
}
```

Análise: O parâmetro `fMatr` da função `LeDadosAluno()` indica se a matrícula deve ser lida (`fMatr` diferente de zero) ou não (`fMatr` igual a zero). A justificativa para o uso desse parâmetro é que essa função será usada em diversos exemplos mais adiante e, algumas vezes, quando essa função é chamada, a matrícula já terá sido lida.

Solução de (b): A função `ExibeDadosAluno()`, apresentada a seguir, implementa a exibição de dados de um aluno na tela.

```
/****
 * ExibeDadosAluno(): Exibe na tela os campos de uma estrutura do tipo tAluno
 *
 * Parâmetros:
 *     dados (entrada) - ponteiro para a estrutura que será apresentada na tela
 *
 * Retorno: Nada
 ****/
void ExibeDadosAluno( const tAluno *dados )
{
   printf("\n>>> Nome:\t%s", dados->nome);
   printf("\n>>> Matricula:\t%s", dados->matr);
   printf("\n>>> Nota 1:\t%3.1f", dados->n1);
   printf("\n>>> Nota 2:\t%3.1f\n", dados->n2);
}
```

Análise: Apesar de o parâmetro da função `ExibeDadosAluno()` ser de entrada, ele é declarado como ponteiro por uma questão de eficiência (v. **Seção 10.5.1**). Para garantir que não ocorre alteração acidental de nenhum campo da estrutura cujo endereço é recebido como parâmetro real, usa-se **const** na declaração do parâmetro formal.

Solução de (c):

```
/****
 * main(): Testa as funções LeDadosAluno() e ExibeDadosAluno()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   tAluno umAluno;

   printf( "\n\t>>> Este programa le e apresenta dados de um aluno\n" );

      /* Lê dados de um aluno, incluindo sua matrícula */
   LeDadosAluno(&umAluno, 1);

      /* Apresenta os dados do aluno */
   printf("\n\t>>> Dados do aluno <<<\n");
   ExibeDadosAluno(&umAluno);

   return 0;
}
```

Para obter um programa completo, você precisará incluir as definições das funções `LeNome()`, `LeIdentidade()` e `LeNota()` e as seguintes linhas em seu início:

```
/*********************** Includes ***********************/

#include <stdio.h>   /* Entrada e saída             */
#include <string.h>  /* Processamento de strings    */
#include <ctype.h>   /* Classificação de caracteres */
#include "leitura.h" /* LeituraFacil                */

/****************** Constantes Simbólicas *****************/

#define MAX_NOME 20 /* Número máximo de caracteres em nome */
#define TAM_MATR  4 /* Número de dígitos numa matrícula */

/******************* Definições de Tipos ******************/

   /* Tipo dos dados de um aluno */
typedef struct {
           char   nome[MAX_NOME + 1]; /* Nome do aluno */
           char   matr[TAM_MATR + 1]; /* Matrícula     */
           double n1, n2;             /* Notas 1 e 2   */
       } tAluno;

/*********************** Alusões ************************/

extern int LeNome(char *nome, int tam);
extern char *LeIdentidade( char *id, int tamArray, const char *prompt );
extern double LeNota(void);
extern tAluno *LeDadosAluno(tAluno *aluno, int fMatr);
extern void ExibeDadosAluno( const tAluno *dados );
```

Exemplo de execução do programa:

```
        >>> Este programa le e apresenta dados de um aluno
Digite o nome (max = 20 letras):
        > Henrique VIII
Digite a matricula com exatamente 4 digitos:
        > 1029
        >>> 1a. nota:
            > 9.5
        >>> 2a. nota:
            > 9.0
        >>> Dados do aluno <<<
>>> Nome:       Henrique VIII
>>> Matricula:  1029
>>> Nota 1:     9.5
>>> Nota 2:     9.0
```

10.10.2 Arte Moderna na Tela (do Computador)

Problema: Escreva um programa que desenha aleatoriamente na tela caracteres que representam as cores: branco, azul, verde e vermelho.

Solução:

```
/************************ Includes ************************/

#include <stdlib.h> /* srand() e rand() */
#include <stdio.h>  /* putchar()        */
#include <time.h>   /* time()           */

/****************** Constantes Simbólicas *****************/

#define HORIZONTAL 18 /* Tamanho horizontal do quadro */
#define VERTICAL   12 /* Tamanho vertical do quadro   */

/******************* Definições de Tipos ******************/

   /* Tipo de enumeração que define as cores que serão usadas */
typedef enum {BRANCO = ' ', AZUL = '+', VERDE = '.', VERMELHO = '\''} tPaleta;

/****
 * EscolheCor(): Escolhe a cor da próxima pincelada
 *
 * Parâmetros: Nenhum
 *
 * Retorno: A primeira cor sorteada por rand() que coincide
 *          com uma das constantes do tipo tPaleta
 ****/
tPaleta EscolheCor(void)
{
   tPaleta cor;

      /* Chama a função rand() repetidamente até que o valor retornado por essa */
      /* função coincida com um dos valores da enumeração do tipo tPaleta       */
   do {
      cor = rand();
   } while ( cor != VERMELHO && cor != AZUL && cor != VERDE && cor != BRANCO );

   return cor; /* Retorna uma cor que consta na paleta */
}
```

```
/****
 *
 * main(): Desenha aleatoriamente na tela caracteres que representam cores
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   int i, j;

      /* Alimenta o gerador de números aleatórios */
   srand(time(NULL));

   putchar('\n'); /* Primeira pincelada */

      /* Pinta o quadro linha a linha */
   for (i = 0; i < VERTICAL; i++) {
      for (j = 0; j < HORIZONTAL; j++) {
         putchar(EscolheCor());
      }

      putchar('\n'); /* Passa para a próxima linha */
   }

   putchar('\n'); /* Toque final */

   return 0;
}
```

Análise: O mais interessante nesse programa é o uso de uma enumeração para representar as cores da *paleta virtual*.

Exemplo de execução do programa:

```
+''.+'..++.  ...''
.  +..'  .'+.  .'+.  +
'...+'+.+  .'+  +'.+
 .'.'....+.+''+'.'
'..'  .'.++'  +.'+  +
+   '..'  '+  '...   '
 .+  .  ..'  ++''  .++
 .'+   ''   .  +.'  ...
  +....'+...   '.''
'+''' '+.+..  .'+'
++ +..   ++.+ +'  ..
'..''++  '  '.+   ..+
.+.  .'  ..'.'.''  .
```

10.10.3 Alinhamento de Strings

Problema: Escreva um programa contendo uma função que alinha um string à esquerda, à direita ou ao centro de um array e preenche o espaço restante com um caractere recebido como parâmetro.

Solução:

```
#include <string.h> /* strlen() */
#include <stdio.h>  /* printf() */
```

```
   /* Tamanho do array que armazena os strings */
#define TAM_ARRAY 30

   /* Tipos de alinhamento */
typedef enum {ESQUERDA, CENTRO, DIREITA} tAlinhamento;

/****
 * AlinhaString(): alinha um string à esquerda, à direita ou ao centro de um array,
 *                 preenchendo o espaço restante com um caractere especificado
 *
 * Parâmetros:
 *      ar[] (saída) - array que conterá o string alinhado
 *      largura (entrada) - tamanho do array ar[]
 *      preenchimento (entrada) - caractere usado no preenchimento do espaço restante
 *      alinhamento (entrada) - o tipo de alinhamento
 *      str (entrada) - o string que será alinhado
 *
 * Retorno: Endereço do array contendo o resultado da operação
 *
 * Nota: Se o comprimento do string str for maior do que o
 *       tamanho do array, ele será truncado.
 ****/
char *AlinhaString( char ar[], int largura, char preenchimento,
                    tAlinhamento alinhamento, const char *str )
{
   char *p;
   int   i,
         menorLargura,/* O menor valor entre o tamanho do */
                      /* array e o comprimento do string  */
         tamStr; /* Comprimento do string */

      /* Calcula o tamanho do string */
   tamStr = strlen(str);

      /* Determina a menor largura */
   menorLargura = largura < tamStr ? largura : tamStr;

      /* Preenche todo o array com o caractere de preenchimento */
   for (i = 0; i < largura - 1; ++i) {
      ar[i] = preenchimento;
   }

      /* Transforma o conteúdo do array num string */
   ar[largura - 1] = '\0';

      /* Determina a posição inicial do string dentro do array */
   if (alinhamento == ESQUERDA) {
      p = ar;
   } else if (alinhamento == CENTRO) {
      p = ar + (largura - menorLargura)/2;
   } else { /* alinhamento == DIREITA */
      p = ar + largura - menorLargura - 1;
   }

         /* Neste ponto, p aponta para o elemento do array */
         /* que receberá o primeiro caractere do string    */

      /* Insere o string em sua posição */
   for (i = 0; i < menorLargura; ++i) {
      p[i] = str[i];
   }
```

```
      /* Retorna o endereço do array que contém o string alinhado */
   return ar;
}

/****
 *
 * main(): Testa a função AlinhaString()  *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   char  ar[TAM_ARRAY];
   char *str = "Um string";

      /* Apresenta o string original */
   printf("\nString original: \"%s\"\n", str);

         /* Apresenta o string alinhado à esquerda, à direita e centralizado */

   AlinhaString(ar, TAM_ARRAY, '*', ESQUERDA, str);
   printf( "\nString alinhado a esquerda:\n\t> \"%s\"\n",
           ar );

   AlinhaString(ar, TAM_ARRAY, '*', DIREITA, str);
   printf( "\nString alinhado a direita:\n\t> \"%s\"\n", ar );

   AlinhaString(ar, TAM_ARRAY, '*', CENTRO, str);
   printf("\nString centralizado:\n\t> \"%s\"\n", ar);

   return 0;
}
```

Análise: Esse programa é relativamente trivial e constitui mais um uso prático de enumeração.

Resultado de execução do programa:

```
String original: "Um string"

String alinhado a esquerda:
        > "Um string********************"

String alinhado a direita:
        > "********************Um string"

String centralizado:
        > "**********Um string**********"
```

10.10.4 Distâncias entre Cidades

Problema: Escreva um programa que armazena distâncias entre a cidade do Rio de Janeiro e algumas cidades do mundo num array de estruturas do tipo:

```
typedef struct {
           char *cidade;
           int   distancia;
       } tCidadeDistancia;
```

O programa deve receber do usuário o nome de uma cidade e, se essa cidade for encontrada no array, ele deve informar qual é a referida distância.

Solução:

```
/************************ Includes ************************/

#include <stdio.h>   /* printf() e putchar() */
#include <string.h>  /* strcmp()             */
#include "leitura.h" /* LeString()           */

/****************** Constantes Simbólicas *****************/
   /* Tamanho do array que armazena os strings lidos */
#define TAMANHO_ARRAY 50

/******************* Definições de Tipos ******************/

typedef struct {
           char *cidade;
           int   distancia;
        } tCidadeDistancia;

/****************** Definições de Funções *****************/

/****
 *
 * ApresentaCidades(): Apresenta na tela o campo 'cidade' de cada estrutura do tipo
 *                     tCidadeDistancia de um array de elementos desse tipo
 *
 * Parâmetros:
 *      ar[] (entrada) - array que contém as estruturas
 *      tam (entrada) - número de elementos do array
 *
 * Retorno: Nada
 *
 ****/
void ApresentaCidades(const tCidadeDistancia ar[], int tam)
{
   int i;

   printf("\n\n\t        >>> Cidades Disponiveis <<<\n\n");

      /* Exibe na tela quatro cidades por linha */
   for (i = 0; i < tam; ++i) {
      printf("%-15s\t", ar[i].cidade);

         /* Quebra linha a cada 4 cidades escritas */
      if (!((i + 1)%4)) {
         putchar('\n');
      }
   }
}

/****
 *
 * main(): Apresenta a distância de uma cidade até o Rio de Janeiro
 *         (se essa informação estiver disponível, obviamente)
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
```

```
int main(void)
{
   char                   cidade[TAMANHO_ARRAY];
   int                    i, nCidades;
   const tCidadeDistancia distancias[] =
     {
       {"Amsterdam",      9795}, {"Bagdad",        11168},
       {"Berlim",        10004}, {"Beirute",       10429},
       {"Jerusalem",     10306}, {"Joao Pessoa",    2448},
       {"Lisboa",         7711}, {"Londres",        9486},
       {"Moscou",        11542}, {"Nova Deli",     14068},
       {"Nova Iorque",    7763}, {"Oslo",          10415},
       {"Otawa",          8283}, {"Paris",          9162},
       {"Viena",          9871}, {"Zurich",         9610}
     };

   printf("\n\t>>> Este programa apresenta a distancia entre o "
          "Rio\n\t>>> de Janeiro e uma cidade escolhida pelo "
          "usuario.\n\t>>> Digite apenas [ENTER] para encerrar.");

      /* Calcula o número de elementos do array */
   nCidades = sizeof(distancias)/sizeof(distancias[0]);

      /* O laço encerra quando o usuário digitar apenas [ENTER] */
   while (1) {
         /* Apresenta ao usuário as cidades disponíveis */
      ApresentaCidades(distancias, nCidades);

         /* Lê o nome da cidade */
      printf("\n>>> Escolha uma cidade para conhecer sua"
             " distancia ate' o\n>>> Rio de Janeiro > ");
      LeString(cidade, TAMANHO_ARRAY);

      if (!*cidade) {
         break; /* Usuário digitou apenas [ENTER] */
      }

         /* Procura a cidade */
      for (i = 0; i < nCidades; ++i) {
         if (!strcmp(cidade, distancias[i].cidade)) {
            break; /* A cidade foi encontrada */
         }
      }

         /* Se o valor de i na saída do laço for igual */
         /* a 'nCidades', a cidade não foi encontrada  */
      if (i == nCidades) {
         printf("\n>>> A cidade \"%s\" nao foi encontrada", cidade);
      } else {
         printf( "\n>>> A distancia do Rio de Janeiro ate' %s "
                 "e' %dKm", cidade, distancias[i].distancia );
      }
   } /* while */

   printf( "\n>>> Obrigado por usar este programa.\n");

  return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa apresenta a distancia entre o Rio
        >>> de Janeiro e uma cidade escolhida pelo usuario.
        >>> Digite apenas [ENTER] para encerrar.
                >>> Cidades Disponiveis <<<

Amsterdam       Bagdad          Berlim          Beirute
Jerusalem       Joao Pessoa     Lisboa          Londres
Moscou          Nova Deli       Nova Iorque     Oslo
Otawa           Paris           Viena           Zurich

>>> Escolha uma cidade para conhecer sua distancia ate' o
>>> Rio de Janeiro > Campina Grande

>>> A cidade "Campina Grande" nao foi encontrada

                >>> Cidades Disponiveis <<<

Amsterdam       Bagdad          Berlim          Beirute
Jerusalem       Joao Pessoa     Lisboa          Londres
Moscou          Nova Deli       Nova Iorque     Oslo
Otawa           Paris           Viena           Zurich

>>> Escolha uma cidade para conhecer sua distancia ate' o
>>> Rio de Janeiro > Joao Pessoa

>>> A distancia do Rio de Janeiro ate' Joao Pessoa e' 2448Km

                >>> Cidades Disponiveis <<<

Amsterdam       Bagdad          Berlim          Beirute
Jerusalem       Joao Pessoa     Lisboa          Londres
Moscou          Nova Deli       Nova Iorque     Oslo
Otawa           Paris           Viena           Zurich

>>> Escolha uma cidade para conhecer sua distancia ate' o
>>> Rio de Janeiro > [ENTER]

>>> Obrigado por usar este programa.
```

10.10.5 Traduzindo Expressões Telefônicas

Preâmbulo: Telefones convencionais modernos incluem em cada tecla, além de um dígito, uma sequência de três letras que permitem ao usuário memorizar facilmente alguns números. Por exemplo, para ligar para uma determinada farmácia, o usuário poderia digitar as teclas que correspondem às letras `F`, `A`, `R`, `M`, `A`, `C`, `I` e `A`, que é bem mais fácil de memorizar do que o número correspondente de telefone (nesse caso, `32762242`) que será efetivamente chamado.

Problema: Escreva um programa que apresenta números de telefones correspondentes a expressões contendo letras, `'0'`, `'1'` ou `'-'`.

Solução:

```
/*********************** Includes *************************/

#include <stdio.h>   /* putchar() e printf()  */
#include <string.h>  /* strchr()              */
#include <ctype.h>   /* isalpha() e toupper() */
#include <stdlib.h>  /* exit()                */
#include "leitura.h" /* LeString()            */

/***************** Constantes Simbólicas ******************/

   /* Tamanho máximo de um string que representa uma expressão telefônica */
#define MAX_EXPRESSAO 30
```

```
   /* Número máximo de caracteres associados a um dígito */
#define MAX_CAR_POR_DIGITO 4

/****************** Definições de Tipos ******************/

   /* Tipo de estrutura que representa uma correspondência */
   /* entre as letras de um string e um dígito de telefone */
typedef struct {
           char letras[MAX_CAR_POR_DIGITO + 1];
           int  digito;    /* Dígito correspondente a  */
                           /* qualquer letra do string */
        } tCorrespondencia;
/************************ Alusões ************************/

extern void ApresentaEspacos(int nEspacos);
extern void ApresentaErro(const char *str,const char *erro);
extern int ChecaExpressao(const char *str);
extern void EscreveTelefone(const char *s);

/***************** Definições de Funções *****************/

/****
 * ApresentaEspacos(): Escreve o número especificado de espaços em branco na tela
 *
 * Parâmetros:
 *      nEspacos (entrada) - número de espaços em branco que serão escritos
 *
 * Retorno: Nenhum
 ****/
void ApresentaEspacos(int nEspacos)
{
   int i;

      /* Escreve o número especificado de espaços na tela */
   for (i = 1; i <= nEspacos; ++i) {
      putchar(' '); /* Escreve um espaço na tela */
   }
}

/****
 * ApresentaErro(): Apresenta uma mensagem de erro apontando para o caractere num
 *                  string que causou o erro
 *
 * Parâmetros:
 *      str (entrada) - o string contendo o erro
 *      erro (entrada) - ponteiro para o caractere causador do erro
 *
 * Retorno: Nenhum
 ****/
void ApresentaErro(const char *str, const char *erro)
{
   int posicao;

   posicao = erro - str; /* Determina o índice do caractere que causou erro */

      /* O índice do caractere não pode ser negativo */
   if (posicao < 0) {
         /* Apresenta uma mensagem de erro padrão */
      printf("\a\n\t>>> Erro nao indentificado!\n");
```

```
      return; /* Nada mais pode ser feito */
   }

      /* Apresenta o string com embelezamento gráfico e sonoro */
   printf("\a\n>>> Erro <<<\n\n%s\n", str);

      /*************************/
      /* Indica o local do erro */
      /*************************/

      /* Escreve a ponta da seta para cima */
   ApresentaEspacos(posicao);
   printf("^\n");

      /* Escreve a haste da seta */
   ApresentaEspacos(posicao);
   printf("|\n");

      /* Escreve a mensagem de erro */
   ApresentaEspacos(posicao);
   printf("-- Caractere espurio\n");
}
/****
 * ChecaExpressao(): Verifica se uma expressão (string) que representa um número
 *                   telefônico é válida e, quando for o caso, indica na tela onde
 *                   se encontra o primeiro erro detectado
 *
 * Parâmetros: str (entrada) - string que representa a expressão que será verificada
 *
 * Retorno: 1, se a expressão estiver OK; 0, em caso contrário
 *
 * Observação: Uma expressão válida contém apenas letras, '0', '1' e '-'
 ****/
int ChecaExpressao(const char *str)
{
   int i;

      /* Examina cada caractere do string e verifica se ele é válido */
   for (i = 0; str[i]; ++i){
      if (!isalpha(str[i])) {
         if ( str[i] != '0' && str[i] != '1'&&
              str[i] != '-') {  /* Foi encontrado um caractere inválido. */
               /* Apresenta mensagem de erro informando onde ele se encontra. */
            ApresentaErro(str, &str[i]);

            return 0; /* Caractere inválido encontrado */
         } /* Segundo if */
      } /* Primeiro if */
   } /* for */
   return 1; /* Não foi encontrado nenhum erro */
}
/****
 * EscreveTelefone(): Escreve na tela o número de telefone
 *                    correspondente a um string válido
 *
 * Parâmetros: s (entrada) - string que representa a expressão cujo
 *                           telefone correspondente será escrito
 *
 * Retorno: Nenhum
 *
```

```
 * Observações:
 *     1.  Esta função assume que o string é válido
 *     2.  Se o correspondente a algum caractere não for
 *         escrito, o programa será abortado
 ****/
void EscreveTelefone(const char *s)
{
      /* O array a seguir contém as correspondências entre letras e dígitos. */
      /* O uso de static garante uma única iniciação da variável e o uso de  */
      /* const garante que ela não será alterada                              */
   static const tCorrespondencia tabela[] = { {"ABC",  '2'},
                                              {"DEF",  '3'},
                                              {"GHI",  '4'},
                                              {"JKL",  '5'},
                                              {"MNO",  '6'},
                                              {"PQRS", '7'},
                                              {"TUV",  '8'},
                                              {"WXYZ", '9'} };
   int i,
       tamTabela, /* Tamanho da tabela de correspondência */
       carEscrito; /* Indicará se o correspondente a um */
                   /* caractere do string foi escrito   */

      /* Calcula o tamanho da tabela de correspondência */
   tamTabela = sizeof(tabela)/sizeof(tabela[0]);

   printf("\n\t>>> Telefone: "); /* Embelezamento */

   for (; *s; ++s) {
         /* O caractere corrente ainda não foi escrito */
      carEscrito = 0;

      if (isalpha(*s)) {
            /* O caractere corrente é uma letra cujo   */
            /* valor numérico será procurado na tabela */
         for (i = 0; i < tamTabela; ++i) {
               /* Procura na tabela o caractere corrente */
               /* convertido em letra maiúscula          */
            if ( strchr(tabela[i].letras, toupper(*s)) ) {
                  /* A letra foi encontrada e o número correspondente será escrito */
               putchar(tabela[i].digito);

               carEscrito = 1; /* O caractere foi escrito */
            }
         }
      } else if (*s == '0' || *s == '1' || *s == '-'){
            /* Se o caractere for '0', '1' ou '-', ele é escrito como ele é */
         putchar(*s);

         carEscrito = 1; /* O caractere foi escrito */
      }

         /* Se não foi escrita a correspondência de   */
         /* algum caractere, o programa será abortado */
      if (!carEscrito) {
         printf("\nERRO: O caractere %c nao foi escrito\n", *s);
         exit(1);
      }
   }
}
```

```
/****
 * main(): Escreve na tela números de telefones correspondentes a
 *         expressões contendo letras, '0', '1' ou '-'
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   char str[MAX_EXPRESSAO + 1];
   int  excesso;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa apresenta numeros de"
           "\n\t>>> telefone correspondentes a expressoes"
           "\n\t>>> contendo letras, '0', '1' ou '-'." );

      /* O laço termina quando o usuário digitar apenas [ENTER] */
   while (1) {
         /* Lê a expressão */
      printf( "\n\nIntroduza uma expressao ou [ENTER] para encerrar:\n\t> " );
      excesso = LeString(str, MAX_EXPRESSAO + 1);

      if (!*str) { /* Checa se o string é vazio */
         break; /* Usuário digitou apenas [ENTER] */
      }
         /* Verifica se foram digitados caracteres demais */
      if (excesso) {
         printf( "\n\t>>> Voce digitou caracteres demais."
                 "\nO maximo permitido sao %d caracteres\n", MAX_EXPRESSAO );

         continue; /* Salta o restante do corpo do laço */
      }

         /* Verifica se a expressão é válida */
      if (!ChecaExpressao(str)) {
            /* A expressão não é válida */
         continue; /* Salta o resto do laço */
      }

         /* Apresenta na tela o número de telefone */
         /* correspondente à expressão introduzida */
      EscreveTelefone(str);
   } /* while */

   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Análise:

- A função **main()** usa um laço de repetição **while** para ler strings que possam representar números telefônicos, conforme a descrição apresentada no preâmbulo. Esse laço encerra quando o usuário digita um string vazio.
- Quando o usuário digita caracteres além do limite permitido, o programa ignora essa entrada de dados e faz uma nova leitura. Caso contrário, o programa verifica se a expressão é válida (v. preâmbulo) utilizando a função `ChecaExpressao()`.

- A função `ChecaExpressao()` examina cada caractere do string recebido como parâmetro e verifica se ele pode fazer parte de uma expressão que representa um número de telefone. Quando um caractere inválido é encontrado, essa função exibe na tela uma mensagem de erro informando o local onde esse caractere se encontra (v. exemplo de execução). Essa mensagem de erro é apresentada pela função `ApresentaErro()`, que, por sua vez, chama a função auxiliar `ApresentaEspacos()`.
- A função `EscreveTelefone()` escreve na tela o número de telefone correspondente a cada expressão considerada válida. Essa função utiliza uma tabela que associa letras a dígitos, sendo que cada dígito é associado a um string contendo todas as letras associadas a ele. Essa tabela é representada por um array de elementos do tipo `tCorrespondencia`, definido como:

```
typedef struct {
           char letras[MAX_CAR_POR_DIGITO + 1];
           int  digito;
        } tCorrespondencia;
```

  A definição do array mencionado é acompanhada pelas palavras chave **static** e **const**, sendo que **static** faz com que o array seja iniciado apenas uma vez, enquanto **const** garante que essa iniciação não é alterada.
- A função `EscreveTelefone()` examina cada caractere do string recebido como parâmetro. Se um caractere desse string for letra, a função encontra o dígito correspondente na tabela mencionada e o escreve na tela; caso contrário, se o caractere for permitido, ele será escrito exatamente como ele é. Se for encontrado um caractere inesperado no string, a função em discussão aborta o programa, já que, nesse caso, terá sido detectada uma inconsistência do programa. Ou seja, a inconsistência reside no fato de a função ter recebido um string que não representa uma expressão válida. Mas, como essa função é chamada apenas após o programa ter checado a expressão, isso não deveria nunca acontecer.

Exemplo de execução do programa:

```
        >>> Este programa apresenta numeros de telefone que
        >>> correspondem a expressoes com letras, '0', '1' ou '-'

Introduza uma expressao ou [ENTER] para encerrar:
        > farmacia

        >>> Telefone: 32762242

Introduza uma expressao ou [ENTER] para encerrar:
        > [ENTER]

        >>> Obrigado por usar este programa.
```

10.11 Exercícios de Revisão

Introdução (Seção 10.1)

1. O que é uma estrutura?
2. Qual é a diferença conceitual entre estruturas e arrays?

Tipos Definidos pelo Programador (Seção 10.2)

3. Cite uma situação que mostre que é inadequado definir tipos usando **#define** em vez de **typedef**.
4. O que é um definidor de tipo? (b) Quais são os definidores de tipos disponíveis em C?

Estruturas (Seção 10.3)

5. Uma estrutura pode possuir um campo com o mesmo nome do campo de outra estrutura?
6. Uma estrutura cujo tipo é definido como:

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

poderia ser definida como um array, já que todos os seus campos são de um mesmo tipo. Explique por que, mesmo levando esse argumento em consideração, definir a referida variável como array não é aceitável do ponto de vista de estilo.

7. Suponha que um valor do tipo **int** ocupe 4 bytes e um endereço também ocupe 4 bytes. Quantos bytes seriam alocados para o conjunto de definições a seguir?

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

tRegistro        registroDaPessoa;
tPtrParaRegistro ptrParaRegistro;
```

8. (a) Como os membros de uma estrutura podem ser iniciados? (b) Pode-se incluir iniciação numa definição de um tipo de estrutura?
9. Como são acessados os campos de uma estrutura?
10. Para que servem os operadores `.` e `->`?
11. O que é uma estrutura aninhada?
12. Suponha que os tipos `tEst1` e `tEst2` sejam definidos como a seguir:

```
typedef struct {
           char  ar[5];
           int   i;
} tEst1;

typedef struct {
           tEst1  *p;
           double  d;
} tEst2;
```

(a) Apresente um exemplo de iniciação de uma variável do tipo `tEst2`. (b) Apresente um exemplo de iniciação de um ponteiro para variável do tipo `tEst2`. (c) Mostre como acessar o elemento de índice `3` do array que faz parte do campo apontado por `p` da variável apresentada como exemplo no item (a). (d) Mostre como acessar o elemento de índice `3` do array que faz parte do campo apontado por `p` usando o ponteiro apresentado como exemplo no item (b).

Operadores de Acesso e Definidores de Tipos (Seção 10.4)

13. (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?
14. Que operador de acesso não faz parte do mesmo grupo de precedência dos demais operadores de acesso?
15. (a) Qual é a precedência dos operadores [], (), `.` (ponto) e `->` com relação aos demais operadores da linguagem C? (b) Qual é a associatividade desses operadores?
16. 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;
```

17. 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;
```

Uso de Estruturas em Funções (Seção 10.5)

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

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

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

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

22. Uma função pode retornar uma estrutura?

23. (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?

24. Suponha que uma função deve separar as partes inteira e fracionária de um número real e armazená-las numa estrutura do tipo definido a seguir:

```
typedef struct {
           int    pInt;  /* Parte inteira     */
           double pFrac; /* Parte fracionária */
        } tPartes;
```

Para simplificar, suponha ainda que o número a ser convertido não requeira correção para evitar erro de truncamento (v. **Seção 7.5**). A referida função deve retornar uma estrutura que contém o resultado da operação ou o endereço dessa estrutura. (a) Qual das funções abaixo não consegue ser compilada? (b) Quais dessas funções retornam zumbis? (c) Quais dessas funções estão corretamente implementadas? (d) Qual delas é a mais eficiente? (e) Qual delas é a mais ineficiente?

(i)

```
tPartes *SeparaPartesReal(tPartes partes, double x)
{
   partes.pInt = (int)x;
   partes.pFrac = x - (double)partes.pInt;
   return &partes;
}
```

(ii)

```
tPartes *SeparaPartesReal(tPartes *partes, double x)
{
   partes.pInt = (int)x;
   partes.pFrac = x - (double)partes.pInt;
   return partes;
}
```

(iii)
```
tPartes *SeparaPartesReal(tPartes *partes, double x)
{
   partes->pInt = (int)x;
   partes->pFrac = x - (double)partes->pInt;
   return partes;
}
```

(iv)
```
tPartes *SeparaPartesReal(double x)
{
   tPartes partes;

   partes.pInt = (int)x;
   partes.pFrac = x - (double)partes.pInt;
   return &partes;
}
```

(v)
```
tPartes *SeparaPartesReal(double x)
{
   static tPartes partes;

   partes.pInt = (int)x;
   partes.pFrac = x - (double)partes.pInt;
   return &partes;
}
```

(vi)
```
tPartes SeparaPartesReal(double x)
{
   tPartes partes;

   partes.pInt = (int)x;
   partes.pFrac = x - (double)partes.pInt;
   return partes;
}
```

(vii)
```
tPartes SeparaPartesReal(tPartes *partes, double x)
{
   partes->pInt = (int)x;
   partes->pFrac = x - (double)partes->pInt;
   return *partes;
}
```

25. Considere a seguinte definição de função na qual o tipo `tPartes` é aquele definido no exercício anterior:

```
tPartes *SeparaPartesReal(tPartes *partes, double x)
{
   partes->pInt = (int)x;
   partes->pFrac = x - (double)partes->pInt;

   return partes;
}
```

(a) Que chamadas da função `SeparaPartesReal()` que aparecem nas funções **main()** a seguir são corretas? (b) O que ocorre com cada chamada incorreta: erro de compilação ou erro de execução?

(i)
```
int main(void)
{
   tPartes partes, *ptrPartes;

   ptrPartes = SeparaPartesReal(&partes, 3.14);

   printf( "Parte inteira: %d\tParte fracionaria: %f\n",
           ptrPartes->pInt, ptrPartes->pFrac );

   return 0;
}
```

(ii)

```
int main(void)
{
   tPartes partes;

   SeparaPartesReal(&partes, 3.14);

   printf( "Parte inteira: %d\tParte fracionaria: %f\n",
           partes.pInt, partes.pFrac );

   return 0;
}
```

(iii)

```
int main(void)
{
   tPartes *ptrPartes;

   SeparaPartesReal(*ptrPartes, 3.14);

   printf( "Parte inteira: %d\tParte fracionaria: %f\n",
           ptrPartes->pInt, ptrPartes->pFrac );

   return 0;
}
```

(iv)

```
int main(void)
{
   tPartes partes;

   partes = *SeparaPartesReal(&partes, 3.14);

   printf( "Parte inteira: %d\tParte fracionaria: %f\n",
           partes.pInt, partes.pFrac );

   return 0;
}
```

(v)

```
int main(void)
{
   tPartes partes, *ptrPartes;

   partes = *SeparaPartesReal(ptrPartes, 3.14);

   printf( "Parte inteira: %d\tParte fracionaria: %f\n",
           partes.pInt, partes.pFrac );

   return 0;
}
```

Uniões (Seção 10.6)

26. (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?

27. Em que situações uniões são úteis?

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

29. Suponha que o tipo **int** ocupe 4 bytes e o tipo **double** ocupe 8 bytes numa dada implementação de C. Considerando as definições das variáveis `uniao` e `registro` abaixo:

```
typedef union {
           char   a;
           int    b;
           double c;
        } tUniao;

typedef struct {
           char   a;
           int    b;
           double c;
        } tRegistro;

tUniao    uniao;
tRegistro registro;
```

(a) Qual é o espaço ocupado pela variável **registro**? (b) Qual é o espaço ocupado pela variável **uniao**?

Registros Variantes (Seção 10.7)

30. O que é um registro variante?

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

32. Quando o seguinte programa é executado, ele apresenta um estranho resultado. Qual é esse resultado e por que ele ocorre?

```
#include <stdio.h>

typedef union {
           int    x;
           double y;
        } tNumero;

int main( void )
{
   tNumero numero;

   numero.x = 10;

   printf( "\nnumero.x = %d\nnumero.y = %f\n",
           numero.x, numero.y );

   numero.y = 10.0;

   printf( "\nnumero.x = %d\nnumero.y = %f\n",
           numero.x, numero.y );

   return 0;
}
```

Iniciadores Designados (Seção 10.8)

33. (a) O que é um iniciador designado de estrutura? (b) Qual é a sintaxe utilizada para iniciar uma estrutura por meio de um iniciador designado?

34. Suponha que se tenha a seguinte definição de tipo:

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

Como se poderia iniciar com `5` o campo `b` de uma variável do tipo `tEstrutura` utilizando um iniciador designado?

35. Suponha que se tenham as seguintes definições de tipo:

```
typedef struct {
           int    x;
           double y;
        } tInterna;

typedef struct {
           char     a;
           tInterna b;
        } tExterna;
```

Como se poderia iniciar com `5` o campo `x` do campo `b` de uma variável do tipo `tExterna` utilizando um iniciador designado?

36. Como se poderia iniciar com `-2` o elemento de índice `3` do array `ar[]` abaixo utilizando um iniciador designado?

```
int ar[10];
```

Enumerações (Seção 10.9)

37. (a) O que é uma enumeração? (b) Para que servem enumerações? (c) Como uma enumeração é definida? (d) Que cuidados o programador deve tomar quando usa enumerações?

38. (a) O que são constantes de enumeração? (b) Como elas são definidas?

39. Quais são as regras de atribuição de valores a constantes de enumerações?

40. Por que, muitas vezes, os valores atribuídos a constantes de uma enumeração não são importantes?

41. Dois tipos de enumerações diferentes podem ter constantes com o mesmo nome num programa monoarquivo?

42. Uma variável de um tipo enumeração é estruturada? Explique.

43. Suponha que um programa contém as seguintes definições:

```
typedef enum {AZUL, VERMELHO, BRANCO} tCores;
...
tCores umaCor;
```

a seguinte atribuição é válida?

```
umaCor = -5;
```

44. Qual é a vantagem obtida com o uso de enumerações num programa?

45. Por que se diz que o uso correto de enumerações depende apenas do programador, e não do compilador?

46. Qual é o valor atribuído a cada constante na seguinte definição de tipo enumeração?

```
typedef enum {C1 = -1, C2, C3 = 0, C4} tEnumeracao;
```

47. (a) Qual é a semelhança entre estruturas e enumerações? (b) Qual é a principal diferença entre enumerações e estruturas?

10.12 Exercícios de Programação

10.12.1 Fácil

EP10.1 Define-se o tipo `tComplexo`, a ser utilizado na representação de números complexos, da seguinte forma:

```
typedef  struct {
             double  parteReal;
             double  parteImaginaria;
         } tComplexo;
```

(a) Considerando essa definição de tipo, escreva funções em C que implementem as seguintes operações sobre números complexos:

(i) Leitura de um número complexo via teclado, com protótipo:

```
tComplexo *LeComplexo(tComplexo *complexo)
```

Essa função deve retornar o valor do endereço recebido como parâmetro.

(ii) Exibição de um número complexo na tela numa forma legível, com protótipo:

```
void ExibeComplexo(const tComplexo *complexo)
```

(iii) Soma de dois números complexos, com protótipo:

```
tComplexo *SomaComplexos(tComplexo *resultado,
                         const tComplexo *complexo1,
                         const tComplexo *complexo2)
```

Essa função deve retornar o endereço da estrutura que armazena o resultado.

(iv) Subtração de dois números complexos, com protótipo:

```
tComplexo *SubComplexos(tComplexo *resultado,
                        const tComplexo *complexo1,
                        const tComplexo *complexo2)
```

Essa função deve retornar o endereço da estrutura que armazena o resultado.

(v) Multiplicação de dois números complexos, com protótipo:

```
tComplexo *MultiplicaComplexos(tComplexo *resultado,
                               const tComplexo *c1,
                               const tComplexo *c2)
```

Essa função deve retornar o endereço da estrutura que armazena o resultado.

(b) Escreva um programa em C que, repetidamente, lê dois números complexos introduzidos via teclado e oferece ao usuário as opções de soma, subtração e multiplicação de complexos, além da opção de encerramento do programa. Após a execução da operação escolhida pelo usuário, o programa deve exibir o resultado na tela.

EP10.2 Considere o tipo `tPonto`, que representa pontos no plano euclidiano, definido como:

```
typedef struct {
           double x, y;
        } tPonto;
```

(a) Escreva uma função que lê um ponto do plano cartesiano e armazena-o numa estrutura do tipo `tPonto`. O parâmetro único dessa função deve ser um ponteiro para essa estrutura. [**Sugestão**: Use `LeReal()` da biblioteca LeituraFacil para ler cada coordenada.] (b) Escreva uma função que recebe dois parâmetros do tipo `tPonto` que representam pontos do plano cartesiano e apresenta na tela a equação da reta que passa pelos dois pontos. [**Sugestão**: Consulte um bom texto sobre Geometria Analítica elementar para saber como se determina a equação da reta que passa por dois pontos.] (c) Escreva um programa que lê valores para duas variáveis do tipo `tPonto` usando a função solicitada no item (a) e chama a função solicitada no item (b) para escrever a equação da reta que passa pelos dois pontos representados por essas variáveis.

EP10.3 (a) Considerando o tipo `tPonto` definido no exercício **EP10.2**, escreva uma função que recebe dois parâmetros desse tipo, que representam pontos do plano cartesiano e retorna a distância entre esses pontos. [**Sugestão**: Consulte um bom texto sobre Geometria Analítica elementar para saber como se calcula a distância entre dois pontos no plano cartesiano.] (b) Escreva um programa que lê valores para duas variáveis do tipo `tPonto`, chama a função solicitada no item (a) para calcular a distância entre os dois pontos representados por essas variáveis e exibe o resultado na tela. [**Sugestão**: Use a função solicitada no item (a) do exercício **EP10.2** para leitura dos pontos.]

EP10.4 (a) Considerando o tipo `tPonto` definido no exercício **EP10.2**, escreva uma função que recebe três parâmetros desse tipo, que representam pontos do plano cartesiano e retorna `1` se eles forem colineares ou `0`, em caso contrário. [**Sugestões**: (1) Consulte um livro sobre Geometria Analítica elementar para saber como se verifica se três pontos do plano cartesiano são colineares. (2) Você precisará usar a função `ComparaDoubles()` definida na **Seção 5.11.6**.] (b) Escreva um programa que lê valores para três variáveis do tipo `tPonto`, chama a função solicitada no item (a) para verificar se os três pontos representados por essas variáveis são colineares e exibe o resultado na tela. [**Sugestão**: Use a função solicitada no item (a) do exercício **EP10.2** para leitura dos pontos.]

EP10.5 Suponha que intervalos de tempo sejam representados num programa em C por estruturas do tipo definido como:

```
typedef struct {
           int hora;
           int min;
        } tTempo;
```

(a) Escreva uma função que calcula a soma de dois intervalos de tempo. [**Sugestão**: Some primeiro os minutos e, se o resultado exceder `60`, subtraia esse valor dos minutos e acrescente `1` à soma das horas.] (b) Escreva uma função que subtrai dois intervalos de tempo. [**Sugestão**: Subtraia primeiro os minutos e, se o resultado for negativo, some `60` ao resultado e subtraia `1` do total de horas.] (c) Escreva uma função que lê um intervalo de tempo. [**Sugestão**: Use a função `LeIntEntre()`, definida no exemplo da **Seção 8.11.8**.] (d) Escreva uma função que exibe na tela um intervalo de tempo. (e) Escreva um programa que testa as funções propostas nos itens de (a) a (d).

10.12.2 Moderado

EP10.6 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

EP10.7 (a) Escreva uma função que recebe um array de elementos do tipo `tComplexo` definido no exercício **EP10.1**, calcula a soma dos elementos do array e retorna o endereço do resultado, que também é armazenado no terceiro parâmetro da função, cujo protótipo é:

```
tComplexo *SomaArrayComplexos( const tComplexo ar[], int nElementos,
                               tComplexo *soma )
```

(b) Escreva um programa que lê um array de elementos do tipo `tComplexo` usando a função `LeComplexo()` (v. **EP10.1**), soma os elementos do referido array usando a função solicitada no item (a) e apresenta o resultado na tela utilizando a função `ExibeComplexo()` solicitada no exercício **EP10.1**.

EP10.8 Escreva um programa em C que exibe na tela um calendário anual para o ano escolhido pelo usuário, a partir de 1899 e sabendo que o dia 1° de janeiro de 1899 foi um dia de domingo. [**Sugestão**: Utilize as sugestões apresentadas para o exercício **EP5.24**.]