Capítulo 9

ESTRUTURAS, UNIÕES E ENUMERAÇÕES

9.1 Introdução

Estruturas, uniões e enumerações representam o tema central deste capítulo. **Estruturas** são variáveis estruturadas similares a arrays. Entretanto, estruturas diferem de arrays por permitirem que seus elementos, aqui denominados de **campos** (ou **membros**), sejam de tipos diferentes. Por causa desta característica, estruturas constituem tipos de dados **heterogêneos**. Em algumas linguagens de programação, estruturas são conhecidas como **registros**.

Uniões são variáveis estruturadas e heterogêneas semelhantes a estruturas, mas diferem destas pelo fato de ter seus campos compartilhados em memória. Apesar das semelhanças com estruturas e uniões na forma como são definidas, **enumerações** não são tipos estruturados e foram incluídas neste capítulo apenas devido a estas semelhanças.

9.2 Definições e Iniciações de 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. Existem várias formas sintáticas permitidas para a definição de uma estrutura em C, mas elas podem ser resumidas em dois esquemas genéricos: definições de estruturas usando (1) declarações de ***rótulos*** e (2) declarações de ***tipos***.

9.2.1 Definições de Estruturas Usando Declarações de Rótulos

Utilizando declaração de rótulo, uma estrutura pode ser definida conforme esquematizado a seguir:

```
struct rótulo-da-estrutura {
      tipo1  campo1;
      tipo2  campo2;
           ...
      tipoN  campoN;
} lista-de-variáveis1;
struct  rótulo-da-estrutura  lista-de-variáveis2;
...
struct  rótulo-da-estrutura  lista-de-variáveisN;
```

Neste formato, tanto ***rótulo-da-estrutura*** quanto ***lista-de-variáveis1*** são opcionais, mas, na prática, se estes identificadores forem simultaneamente abandonados, não se poderá ter nenhuma variável e a declaração de estrutura não terá nenhuma utilidade. Utilizando este esquema, na prática, raramente o rótulo é abandonado, pois isto implicaria declarar todas as variáveis do tipo de estrutura especificado no local de declaração do próprio tipo de estrutura.

Neste esquema, uma estrutura (variável), denominada `registroDaPessoa`, utilizada para conter o nome e a data de nascimento (dia, mês e ano) de uma pessoa, poderia ser declarada assim:

```
struct  registro {
       char   nome[30];
       short  dia, mes, ano;
};

struct   registro  registroDaPessoa;
```

No exemplo, `registro` é o rótulo da estrutura. Ele é utilizado tanto na declaração do tipo de estrutura (primeira declaração) quanto na definição da variável (segunda declaração). Observe que campos de um mesmo tipo podem ser declarados juntos e separados por vírgulas, como em declarações de variáveis de um mesmo tipo.

Uma declaração de rótulo de estrutura é similar a uma definição de tipo e, portanto, não causa alocação de memória[1]. Também, do mesmo modo que uma declaração de tipo, um rótulo de estrutura pode ser utilizado para declarar quaisquer outras variáveis. O formato de declaração apresentado aqui difere daquele que usa declaração de tipo porque ele requer o uso da palavra **struct** na declaração de variáveis. Por exemplo, suponha que, além da variável `registroDaPessoa`, deseje-se um ponteiro para uma estrutura cujo rótulo é `registro`. Então, poder-se-ia definir este ponteiro como:

```
struct   registro  *ptrParaRegistro;
```

Observe que a variável `registroDaPessoa` do exemplo acima poderia também ser declarada como:

```
struct  {
       char   nome[30];
       short  dia, mes, ano;
}  registroDaPessoa;
```

Entretanto, neste último exemplo, como a estrutura não tem rótulo, quaisquer outras variáveis teriam que ser definidas no mesmo local onde a variável `registroDaPessoa` é definida. Conseqüentemente, este formato de declaração é recomendado em situações nas quais as estruturas precisam ser definidas em apenas uma parte do programa (usualmente, num único arquivo). Por exemplo, para definir a variável `ptrParaRegistro` sem o uso de rótulo, poder-se-ia escrever:

```
struct   {
     char   nome[30];
     short  dia, mes, ano;
} registroDaPessoa, *ptrParaRegistro;
```

Neste último exemplo, se o ponteiro `ptrParaRegistro` precisasse ser definido em outro arquivo e não no mesmo arquivo onde a variável

[1] Em C++, declaração de rótulo de estrutura é exatamente o mesmo que uma definição de tipo estrutura.

registroDaPessoa é definida, seria necessário repetir a declaração de toda a estrutura no arquivo onde o ponteiro ptrParaRegistro tivesse de ser definido, como:

```
struct  {
    char   nome[30];
    short  dia, mes, ano;
} *ptrParaRegistro;
```

O esquema de definição de estruturas usando rótulos é considerado obsoleto, mas ainda é fácil encontrar muitos programas que o utilizam.

9.2.2 Definições de Estruturas Usando Definições de Tipos

O uso de **typedef** constitui a melhor forma de definição de estruturas. Ele é esquematizado como:

typedef struct *rótulo-da-estrutura* {
    tipo1 campo1;
    tipo2 campo2;
    ...
    tipoN campoN;
} *nome-do-tipo*;
nome-do-tipo lista-de-nomes-de-variáveis;

Aqui, define-se um novo tipo, que é sinônimo de tudo o que precede sua definição (v. **Seção 4.6**), incluindo a palavra **struct**. O rótulo da estrutura, que aparentemente é desnecessário, torna-se indispensável quando se declara uma estrutura com auto-referência, como será visto na **Seção 9.5**.

Utilizando este esquema de declaração de estrutura, o tipo tRegistro e as variáveis registroDaPessoa e ptrParaRegistro poderiam ser declaradas como:

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

tRegistro  registroDaPessoa, *ptrParaRegistro;
```

Note, neste último exemplo, que o rótulo da estrutura é dispensável, já que o tipo aqui definido é suficiente para declarar quaisquer variáveis. Note, ainda, que mais de um tipo pode ser definido numa única declaração. Por exemplo:

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

tRegistro         registroDaPessoa;
tPtrParaRegistro  ptrParaRegistro;  /* Não se deve mais */
                                    /* usar asterisco aqui */
```

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

Usualmente, quando uma estrutura é utilizada em várias partes de um programa multiarquivo, coloca-se uma definição de tipo (preferível) ou de rótulo da estrutura num arquivo de cabeçalho que pode, então, ser acessada pelos vários arquivos do programa.

Dois campos em estruturas diferentes podem possuir o mesmo nome. Por exemplo:

```
struct {
      int    a;
      float  b;
}  estruturaA;

struct {
      int   a;
      char  b;
}  estruturaB;
```

Devido à forma como os campos são acessados (v. **Seção 9.3**), não existe chance de colisão de identificadores de campos no exemplo acima. É permitido, ainda, o uso de um mesmo nome para rótulo, variável e campo de um tipo estrutura, como, por exemplo:

```
struct E {
      int   E;
}  E;
```

Em termos de estilo, entretanto, o abuso cometido neste último exemplo deve ser evitado.

9.2.3 Iniciações de Estruturas

Uma estrutura pode ser iniciada de modo similar a um array. Isto é, a estrutura a ser iniciada deve ser seguida pelo sinal de igualdade e 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];
            short  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 ou de rótulo de estrutura, pois estas declarações não alocam espaço em memória. Por exemplo, a seguinte tentativa de iniciação seria inválida:

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

9.2.4 Atribuição entre Estruturas

Uma estrutura pode ser atribuída a outra desde que ambas sejam estruturas do mesmo tipo. Por exemplo, dada a seguinte declaração de variáveis:

```
struct {
    int    a;
    float  b;
} e1, e2, *ptr;
```

as seguintes atribuições são perfeitamente válidas:

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

9.3 Acesso a Campos de Estruturas

Existem duas formas de acesso aos campos de uma estrutura, dependendo do fato de se estar lidando com uma variável (estrutura) ou com um ponteiro para uma estrutura. No primeiro caso, utiliza-se o operador **.** (ponto) e, no segundo, utiliza-se o operador **->** (constituído pelo símbolo de menos seguido pelo símbolo de maior do que e sem espaço entre os mesmos). Para acessar um campo de uma estrutura utilizando qualquer destes operadores, deve-se colocar o operador entre o nome da variável que representa a estrutura e o nome do campo que se deseja acessar. Por exemplo, considere as seguintes declarações:

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

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

O ponteiro `ptrParaRegistro` foi iniciado com o endereço da variável `registroDaPessoa`. Assim, cada campo da variável `registroDaPessoa` poderia ser acessado de duas maneiras, conforme mostrado na tabela a seguir:

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

Deve-se notar que, na realidade, o operador **->** é uma abreviação para as operações conjuntas de indireção de ponteiro e acesso a um campo utilizando o operador ponto. Por exemplo:

```
ptrParaRegistro->ano
```

é o mesmo que:

```
(*ptrParaRegistro).ano
```

Um campo acessado de uma estrutura comporta-se como uma variável comum. Em outras palavras, pode-se utilizar um campo acessado em qualquer local onde uma variável do tipo do campo poderia ser utilizada. Por exemplo, pode-se alterar o campo `ano` da variável `registroDaPessoa` fazendo-lhe uma atribuição como:

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

ou, de forma equivalente:

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

Os operadores **.** e **->**, juntamente com os operadores **[]** e **()**, fazem parte de um mesmo grupo de precedência (v. **Seção 10.2**). A precedência deste grupo é a mais alta dentre todos os operadores de C e sua associatividade é da esquerda para a direita. Isto significa que:

`a.b.c` é equivalente a `(a.b).c`

e

`a->b->c` é equivalente a `(a->b)->c`

9.4 Estruturas Aninhadas

Um campo de uma estrutura pode ser de qualquer tipo, inclusive de um tipo estrutura. Quando um campo de uma estrutura é por si mesmo uma estrutura, a estrutura que o contém é denominada **estrutura aninhada**. Por exemplo, a definição do tipo `tRegistro`:

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

poderia ser reescrita como:

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

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

Uma variável do tipo `tRegistro2` é uma estrutura aninhada, pois o segundo campo dela, denominado `data`, também é uma estrutura. Existem outras formas de se definirem estruturas aninhadas, mas o uso de **typedef** ainda é o mais recomendado.

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, o que facilita a legibilidade e o reuso de código. Por exemplo, a estrutura `tData` poderia ser reutilizada na definição de um outro tipo de estrutura.

Quando uma estrutura aninhada é iniciada, utilizam-se chaves do mesmo modo que em iniciações de arrays multidimensionais. 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 `data` da estrutura aninhada `registroDaPessoa` pode ser acessado como:

```
registroDaPessoa.data
```

Mas o campo `data` é também uma estrutura (interna). Portanto, o campo `dia` desta estrutura pode ser acessado como:

```
registroDaPessoa.data.dia
```

que é o mesmo que:

```
(registroDaPessoa.data).dia
```

devido à associatividade do operador ponto.

O acesso por meio de ponteiros é similar. Suponha que `ptrParaRegistro` seja um ponteiro para o tipo `tRegistro`. Então, o campo `dia` do campo `data` da estrutura apontada por `ptrParaRegistro` pode ser acessado como:

```
ptrParaRegistro->data.dia
```

Note, neste último exemplo, que o operador **->** é utilizado apenas uma vez, visto que o campo `data` é uma estrutura e não um ponteiro para estrutura, como é o caso de `ptrParaRegistro`.

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

9.5 Estruturas com Auto-referência

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

```
struct E {
    int     a, b;
    struct  E *ptrParaE;
};
```

Estruturas deste tipo são denominadas **estruturas com auto-referência** e são bastante úteis para a criação de listas e outras estruturas de dados encadeadas (v. **Capítulo 11**).

Ponteiros podem fazer referências a estruturas que ainda não foram definidas e isto permite a criação de **estruturas com referência mútua**, como apresentado no exemplo a seguir:

```
struct  E1 {
   int     a;
   struct  E2 *ptrParaE2;
};

struct  E2 {
   int     a;
   struct  E1 *ptrParaE1;
};
```

Cada estrutura do último exemplo possui um ponteiro para a outra e, por isso, estas estruturas são consideradas estruturas com referência mútua.

Auto-referências ou referências mútuas não são permitidas com o uso de definições de tipo sem rótulos. Por exemplo, a seguinte tentativa de auto-referência não é permitida:

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

O compilador considera ilegal a declaração do campo `ptrParaE` porque ele julga que o tipo `tE` é desconhecido no local onde este campo é declarado.

9.6 Estruturas como Parâmetros e Retornos de Funções

Diferentemente do que ocorre com arrays, estruturas inteiras (e não apenas ponteiros, como ocorre com arrays) podem ser passadas como parâmetros e retornadas de funções.

9.6.1 Estruturas como Parâmetros de Funções

Existem duas maneiras de se passar uma estrutura como argumento para uma função: (1) **passagem da própria estrutura** ou (2) **passagem de um ponteiro para a estrutura**. Ou seja, o parâmetro formal que representa uma estrutura na declaração de uma função pode ser declarado, respectivamente, como (1) uma estrutura ou (2) um ponteiro para estrutura. Por exemplo, considerando a definição do tipo `tRegistro`:

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

as funções `F1()` e `F2()`, apresentadas a seguir, ilustram as duas formas de declaração de argumentos representando estruturas do tipo `tRegistro`:

```
void F1(tRegistro reg)  /* O argumento é uma estrutura */
{
     ...
}

void  F2(tRegistro  *ptrReg)       /* O argumento é um ponteiro */
{
     ...
}
```

Considerando as declarações de `F1()` e `F2()`, se a variável `registro` fosse declarada como:

```
tRegistro  registro;
```

Então, as funções `F1()` e `F2()` poderiam ser chamadas como:

```
F1(registro);
```

e

```
F2(&registro);
```

Por outro lado, considerando a definição de ponteiro a seguir:

```
tRegistro  *prtParaRegistro;
```

as chamadas de `F1()` e `F2()` deveriam ser:

```
F1(*prtParaRegistro);
```

e

```
F2(prtParaRegistro);
```

Note que, em ambas as situações ilustradas acima, `F1()` sempre recebe como argumento uma estrutura e `F2()` sempre recebe um endereço de estrutura.

Na passagem de um ponteiro para uma estrutura [como na função `F2()` acima], qualquer alteração de valor em algum campo da estrutura dentro da função é comunicada à função que fez a chamada. Por outro lado, na passagem de uma estrutura [como na função `F1()` acima], qualquer alteração feita dentro da função incide sobre uma cópia da estrutura; conseqüentemente, a estrutura passada para a função permanece inalterada após a chamada.

Existem duas situações nas quais uma estrutura (e não um ponteiro) deve ser passada para uma função:

1. Quano a estrutura é relativamente pequena (i.e., aproximadamente do mesmo tamanho de um ponteiro).

2. Quando se deseja garantir que a função não modifique a estrutura sendo passada como argumento. Mas, reconhecidamente, este argumento é fraco, pois pode-se garantir que a estrutura não seja modificada pela função utilizando-se um ponteiro acompanhado do qualificador **const**.

Na maioria das situações práticas, estruturas devem ser passadas para funções por meio de ponteiros.

O método escolhido para passagem de uma estrutura determina o tipo de operador que será utilizado no corpo da função para referência aos campos da estrutura. Se uma estrutura for passada para a função, o operador **.** (ponto) deve ser utilizado no acesso aos seus campos. Por outro lado, se um ponteiro para estrutura for passado para a função, o operador **->** deve ser usado em acessos aos campos da estrutura.

O aprendiz de C deve atentar para o fato de passagens de estruturas para funções serem bem diferentes de passagens de arrays. Para passar um array como parâmetro real numa chamada de função, deve-se simplesmente utilizar o nome do array sem índice. O compilador interpreta o nome do array como um ponteiro para o elemento inicial do array. Por outro lado, quando se passa o nome de uma estrutura como parâmetro para uma função, o compilador interpreta o nome da estrutura como representante da estrutura inteira e não como um ponteiro para o primeiro elemento (campo) da estrutura.

Outra diferença entre arrays e estruturas com relação à passagem de parâmetros ocorre na definição de parâmetros formais de funções. Por exemplo, as duas definições de parâmetros representando arrays a seguir são idênticas:

```
void  F1(int ar[]) /* ar[] é convertido num ponteiro para int */
void  F2(int *ar)  /* ar é um ponteiro para int */
```

Entretanto, as seguintes definições de parâmetros que representam estruturas são bem diferentes:

```
     /* estr representa uma estrutura */
void  F3(struct registro estr)

     /* estr é um ponteiro para uma estrutura */
void  F4(struct registro *estr)
```

9.6.2 Funções com Retorno de Estruturas

Uma função em C pode retornar tanto uma estrutura quanto um ponteiro para uma estrutura. Em qualquer das situações, o tipo de retorno declarado no cabeçalho da função deve ser compatível com o valor que ela retorna. Por exemplo, a função `F3()` esquematizada a seguir retorna uma

estrutura do tipo `tRegistro` definido anteriormente:

```
tRegisto F3(void)
{
   tRegistro   registroASerRetornado;
  ...
   return  registroASerRetornado;
}
```

A função `F4()` esquematizada a seguir retorna um ponteiro para uma estrutura do tipo `tRegistro`:

```
tRegisto *F4(void)
{
   static  tRegistro   registroASerRetornado;
  ...
   return  &registroASerRetornado;
}
```

Existem dois detalhes a ser observados na declaração da função `F4()`. Primeiro, a função retorna o endereço de uma estrutura cujo escopo é local à função. Segundo, esta estrutura deve ter duração fixa; caso contrário, esta variável seria extinta no retorno da função e o endereço retornado apontaria para uma posição de memória que não está mais alocada para esta estrutura (v. discussão sobre zumbis na **Seção 7.5.4**). No caso da função `F3()`, onde a própria estrutura é retornada, esta exigência não se faz necessária.

Existe uma outra forma segura de se retornar um ponteiro para uma estrutura diferente daquela utilizada pela função `F4()`. Este método utiliza alocação dinâmica de memória, que será vista no **Capítulo 11**.

O retorno de ponteiros para estruturas é muito mais utilizado do que o retorno de estruturas em si. Isto deve-se principalmente a questões de eficiência, pois, tipicamente, um ponteiro ocupa muito menos espaço do que uma estrutura.

O retorno de uma estrutura ou ponteiro para uma estrutura é, às vezes, utilizado em situações nas quais se deseja o retorno de mais de um valor de uma função. Cada instrução **return** pode retornar apenas um valor para o ponto de chamada da função, mas, se este valor for uma estrutura ou um ponteiro para uma estrutura, pode-se retornar vários valores embutidos nos campos da estrutura.

9.7 Arrays de Estruturas

Um array de estruturas pode ser criado em C precedendo-se o nome do array por um tipo estrutura previamente definido. Por exemplo, considerando a declaração do tipo `tPessoa`:

```
typedef struct  {
           char   nome[30];
           short  dia, mes, ano;
         } tPessoa;
```

pode-se declarar o array `arrayDePessoas` como:

```
tPessoa  arrayDePessoas[20];
```

Utilizando-se arrays de estruturas, pode-se apresentar um exemplo mais completo e prático do uso de estruturas. Suponha que se tenha um array contendo estruturas do tipo `tPessoa` e que os campos `dia`, `mes` e `ano` deste tipo representem a data de nascimento de uma pessoa. Se for desejado contar o número de pessoas nascidas entre dois determinados anos num array de elementos do tipo `tPessoa`, esta contagem poderia ser realizada pela função a seguir:

```
unsigned PessoasEntreAnos( tPessoa pessoas[],
                           unsigned  numeroDePessoas,
                         unsigned anoInicial, unsigned anoFinal
)
{
   unsigned  i, contador = 0;

     if  (anoInicial < anoFinal){
        for (i = 0; i < numeroDePessoas; i++){
           if ( (pessoas[i].ano > anoInicial) &&
                (pessoas[i].ano < anoFinal) ){
             contador++;
        }
        }
     }

     return  contador;
}
```

Uma função **main()** que utilizasse a função `PessoasEntreAnos()` para determinar o número de pessoas nascidas entre 1960 e 1980 poderia ter o seguinte formato:

```
int main(void)
{
   tPessoa  arrayDePessoas[20];

   ...   /* Atribuição de valores aos registros do array */

    printf( "O numero de pessoas nascidas entre 1960 e 1980 é
%d\n",
      PessoasEntreAnos(arrayDePessoas, 20, 1960,1980) );

   return 0;
}
```

A função `PessoasEntreAnos()` definida acima funciona perfeitamente bem, conforme especificado. No entanto, esta função possui um inconveniente em termos de eficiência que poderia ser melhorado: a cada vez que o laço **for** desta função é executado, são feitos dois acessos a elementos do array `pessoas` por meio do índice `i` nas expressões:

```
pessoas[i].ano > anoInicial e pessoas[i].ano < anoFinal
```

Recorde-se que, conforme foi visto na **Seção 7.4**, cada referência a um array por meio de um índice é traduzida pelo compilador como uma referência da soma com aplicação do fator de escala do ponteiro para o elemento inicial do array com este índice. No caso corrente, cada referência `pessoas[i]` é traduzida pelo compilador como:

```
*(pessoas + i)
```

O problema de eficiência ocorre quando a aplicação do fator de escala é efetuada, pois o valor da variável `i` deve ser multiplicado pelo tamanho de cada elemento do array em bytes (neste caso, o tamanho do tipo `tPessoa`) antes de ser somado ao ponteiro `pessoas`. Será mostrado a seguir que a complexidade dessas operações pode ser reduzida por meio do uso de ponteiros, em vez de índices, para acessar os elementos do array.

Para utilizar este método, lembre-se de que o parâmetro `pessoas` é interpretado como um ponteiro para o primeiro elemento do array. Portanto, este elemento pode ser acessado por `*pessoas` e os elementos seguintes podem ser acessados aplicando-se o operador ***** sobre incrementos sucessivos do ponteiro `pessoas`. Uma nova versão da função `PessoasEntreAnos()` poderia, então, ser escrita como:

```
unsigned PessoasEntreAnos( tPessoa pessoas[],
                           unsigned  numeroDePessoas,
                           unsigned anoInicial,
                           unsigned anoFinal )
{
   unsigned  i, contador = 0;

   if  (anoInicial < anoFinal){
   for (i = 0; i < numeroDePessoas; pessoas++, i++){
         if ( (pessoas->ano > anoInicial) &&
              (pessoas->ano < anoFinal) ){
            contador++;
         }
      }
   }

   return  contador;
}
```

Nesta última versão da função `PessoasEntreAnos()`, as operações de multiplicação necessárias para a aplicação do fator de escala são evitadas e o tempo economizado com esta estratégia pode ser substancial, dependendo do tamanho do array. Por exemplo, com um array de 5 milhões de correntistas de um banco, 10 milhões de operações de multiplicação seriam economizadas. Convença-se de que realmente entendeu este exemplo, pois ele contém conhecimentos cruciais para sua evolução como programador em C. As duas versões da função `PessoasEntreAnos()` são capazes de diferenciar um programador novato de um programador experiente em C.

Quando se incrementa um parâmetro que é um ponteiro para um array, como o argumento `pessoas` na função `PessoasEntreAnos()`

do último exemplo, com a finalidade de acessar os elementos do array num laço de repetição, ao final do laço, este parâmetro não estará mais apontando para o início do array. No último exemplo, ao final do laço **for**, o ponteiro `pessoas` não estará apontando nem mesmo para algum elemento do array; i.e., ele estará apontando para o próximo endereço além do último elemento do array. Conseqüentemente, se houver necessidade de a função processar novamente o array a partir de seu início, é necessário que se restabeleça um ponteiro para o início do array. Uma forma de se contornar este problema sem possibilidade de introduzir erro é utilizar um ponteiro auxiliar, em vez do próprio argumento, para percorrer o array. Por exemplo, se a função `PessoasEntreAnos()` do último exemplo precisasse processar novamente o array após o laço **for**, poder-se-ia utilizar a seguinte definição no início da função:

```
tPessoa  *ptrAux = pessoas;  /* Lembre-se de não usar "&" */
                              /* antes de pessoas        */
```

Então, substituem-se todas as ocorrências de `pessoas` no laço **for** por `ptrAux`. Assim, ao final deste laço, apesar de o ponteiro `ptrAux` não mais apontar para o array, o ponteiro `pessoas` continuará apontando para o início do array.

Uma outra estratégia para fazer o ponteiro `pessoas` retornar ao início do array consiste em desfazer as modificações sofridas pelo ponteiro dentro do laço **for**. No caso do último exemplo, isso consistiria em subtrair `i` ou `numeroDePessoas` do ponteiro `pessoas` que foi o total com o qual este ponteiro foi incrementado. Esta estratégia, no entanto, é sensível a erros e deve ser evitada (ou, pelo menos, utilizada com muito cuidado).

9.8 Arrays Flexíveis Como Membros de Estruturas (C99)

Um **array flexível** (C99) é um array que não possui tamanho especificado e que é membro de uma estrutura com mais de um membro. Ele deve ser o último campo de uma estrutura. Por exemplo, o membro `arrayFlexivel` da estrutura `est1` apresentada a seguir é um array flexível.

```
struct {
   int a;
   float b;
   int arrayFlexivel[];
} est1;
```

Estruturas com arrays flexíveis podem ser utilizadas para acessar objetos de tamanho variável, conforme exemplificado no seguinte programa:

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

typedef struct {
   int a;
   int b[];
} tEstrutura;

tEstrutura *F(int N)
{
   tEstrutura *p;
   int        i;

   if (N <= 0)
      return NULL;

    /* A instrução a seguir é equivalente a fazer com que */
    /* p aponte para uma estrutura do tipo tEstrutura, tendo */
    /* como último elemento um array com N elementos do  */
    /* tipo int, ao invés do array de tamanho indefinido. */
   p = malloc( sizeof(tEstrutura) + N*sizeof(int) );

   if (!p)
         return NULL; // Não houve alocação; não é possível
continuar
   p->a = N; // Atribui o número de elementos do array ao campo
"a"

      // Inicia os elementos do array alocado dinamicamente
      // O conteúdo de cada elemento será seu próprio índice
   for (i = 0; i < N; ++i)
      p->b[i] = i;

   return p;
}
```

```
int main()
{
   tEstrutura *p;

   p = F(rand()); // A função rand() gera números aleatórios
                  // (V. Volume II)

   if (!p)
      return 1;

   printf("Numero de elementos processados: %d\n", p->a);

   for (int i = 0; i < p->a; ++i)
      printf("Elemento[%d] = %d\n", i, p->b[i]);

   return 0;
}
```

Uma estrutura contendo um array flexível como membro não pode ser membro de outra estrutura ou união ou ser elemento de um array. Além disso, a aplicação do operador **sizeof** sobre uma estrutura contendo um array flexível considera o tamanho deste array como sendo 0.

9.9 Uniões

Uniões são tipos de dados similares às estruturas, com a diferença de que os campos de uma união compartilham a mesma área de memória. Quer dizer, todos os campos de uma união começam no mesmo endereço em memória. Assim, uniões são utilizadas primariamente com o objetivo de economizar memória quando os campos não coexistem .

Uniões obedecem às regras sintáticas semelhantes àquelas das estruturas. Portanto, para declarar-se uma união, pode-se utilizar qualquer um dos formatos apresentados na **Seção 9.2** para declaração de estruturas, trocando-se apenas a palavra **struct** por **union**. Por exemplo,

```
typedef  union {
            char   campo1;
            double campo2;
            int    campo3;
         } tUniao;

tUniao minhaUniao;
```

O compilador sempre aloca espaço suficiente para conter o membro de maior tamanho de uma união e todos os membros iniciam no mesmo endereço. Os campos de uma união são, portanto, mutuamente exclusivos no sentido de que apenas um deles pode ser considerado válido num dado instante. Por exemplo, se forem feitas as atribuições a seguir à variável `minhaUniao` do último exemplo:

```
minhaUniao.campo1 = 'b';
minhaUniao.campo2 = 3.14;
```

ao final da segunda atribuição, o valor 'b' será perdido e uma tentativa de acesso ao valor de `campo1` ou `campo3` de `minhaUniao` produzirá um resultado sem sentido.

Uma união pode ser iniciada por meio da atribuição de um valor inicial ao primeiro componente da variável[2]. Por exemplo:

```
typedef  union {
            char   campo1;
            double campo2;
            int    campo3;
         } tUniao;

tUniao   minhaUniao = {'a'};
```

Um uso freqüente para uniões é na implementação de **registros variantes**, que consiste em uma estrutura composta de, pelo menos, uma parte fixa e, pelo menos, uma parte variante. Considere, por exemplo, as seguintes definições[3]:

```
typedef       enum    {SOLTEIRO,    CASADO,    DIVORCIADO}
tEstadoCivil;

typedef  struct {
            char   rua[30];
            char   numero[5];
```

[2] De acordo com o padrão C99, pode-se também iniciar qualquer outro membro de uma união usando-se um iniciador designado, conforme será visto na Seção 9.10.

[3] A primeira declaração define um tipo enumeração, que é introduzido na Seção 9.12. Enumerações são simples de assimilar, mas, se não conseguir entender este exemplo, retorne a ele depois de estudar este tipo de dados.

```
            char   cidade[20];
            char   uf[3];
            char   cep[10];
         } tEndereco;

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

typedef struct {
            char          nome[30];
            tEndereco     endereco;
            tEstadoCivil  estadoCivil;
            union {
               char  nomeDoConjuge[30];
               short moraSozinho;
            tData  dataDoDivorcio;
        } complemento;
} tEmpregado;
tEmpregado empregado;
```

A variável `empregado` é um registro variante, cujas partes fixas consistem nos campos `nome`, `endereco` e `estadoCivil`[4], enquanto a única parte variante é representada pelo campo `complemento`. Este último campo é uma união consistindo nos campos `nomeDoConjuge`, `moraSozinho` e `dataDoDivorcio`. Apenas um dos campos desta união é válido para uma variável do tipo `tEmpregado` num dado instante.

Normalmente, em registros variantes, utiliza-se um dos campos fixos como **indicador**. O valor do campo indicador informa qual é o campo variante da estrutura válido num dado instante. No último exemplo, o campo `estadoCivil` serve como indicador. Utilizando um campo indicador, o programador pode acessar corretamente o campo variante, como mostra o seguinte fragmento de programa:

[4] O campo `estadoCivil` é uma enumeração do tipo `tEstadoCivil`. Isto significa que esta variável pode assumir apenas um dos valores enumerados na definição de seu tipo. Enumerações são apresentadas na **Seção 9.12**.

```
if (empregado.estadoCivil == CASADO) {
   printf("%s", empregado.complemento.nomeDoConjuge);
} else if (empregado.estadoCivil == SOLTEIRO) {
   if (empregado.complemento.moraSozinho)
      printf("Mora sozinho");
   else
      printf("Nao mora sozinho");
} else if (empregado.estadoCivil == DIVORCIADO) {
   printf("%d", empregado.complemento.dataDoDivorcio.dia);
   printf("%d", empregado.complemento.dataDoDivorcio.mes);
   printf("%d", empregado.complemento.dataDoDivorcio.ano);
}
```

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 um outro meio para obter esta informação, de modo a não acessar incorretamente um campo variante. Em qualquer situação, é sempre melhor utilizar um campo indicador na declaração de um registro variante.

Uniões também são utilizadas menos freqüentemente para interpretar uma mesma posição de memória de maneiras diferentes. Este uso de uniões, entretanto, pode ser substituído de maneira mais portável e legível pelo uso de conversões explícitas de dados (v. **Seção 1.6.1**).

9.10 Iniciadores Designados de Estruturas e Uniões (C99)

De modo semelhante ao que ocorre com arrays, o padrão C99 também permite o uso de iniciadores designados para estruturas e uniões, conforme mostra o exemplo a seguir:

```
typedef struct {
           int a;
           float b;
           char c;
           char d[10];
        } tEst;

tEst est = {.b = 2.54f, .d[4] = 'c', .a = -2, .d[7] = 'a'};
```

Como exemplo de uso iniciadores designados com uniões, considere:

```
typedef union {
           int a;
           float b;
           char c;
        } tUn;

tUn uniao = {.b = 2.54f};
```

Um campo cujo tipo é uma estrutura também pode ser iniciado usando iniciador designado. Por exemplo:

```
typedef struct {
           int a;
           float b;
        } tEst1;

typedef struct {
           int   x;
           tEst1 y;
        } tEst2;

tEst2 est2 = {.y.b = 2.3f};
```

9.11 Literais Compostos (C99)

Literais compostos, introduzidos pelo padrão C99, permitem a criação de **variáveis anônimas**[5] por meio de uma construção semelhante a uma combinação de conversão implícita e iniciação. Eles podem ser utilizados com qualquer tipo de dados, mas, na prática, são usados principalmente com tipos estruturados (arrays, estruturas e uniões). A sintaxe utilizada para criação de tal variável é:

(tipo do literal composto) iniciador;

[5] Variável anônima é aquela que não possui um identificador associado e, assim, seu conteúdo só pode ser acessado indiretamente por meio de um ponteiro.

Por exemplo:

```
int *p = (int []){ 1, 2 }; // p aponta para um arranjo anônimo
                           // de ints com dois elementos
                           // iniciados com 1 e 2
```

As regras de duração e iniciação para literais compostos são as mesmas vistas anteriormente no **Capítulo 4**. Por exemplo, se o literal composto para o qual o ponteiro `p` do exemplo acima aponta for criado dentro de um bloco, ele terá duração automática; caso contrário, ele terá duração fixa. Em ambos os casos, as regras de iniciação vistas na **Seção 4.2.2** continuam válidas.

Literais compostos podem ser estruturas ou uniões e ainda utilizar iniciadores designados. Por exemplo:

```
typedef struct {
           int   a;
           float b;
           char  c;
           char  d[10];
        } tEst;

tEst *pEst = &(tEst) {.b = 3.5, .d[2] = 'C'};
```

No último exemplo, o ponteiro `pEst` aponta para uma variável do tipo `tEst`, cujo membro `b` é iniciado com `3.5` e cujo terceiro elemento do arranjo que constitui o membro `d` é iniciado com `'C'`.

Literais compostos não precisam ser de tipos estruturados, apesar de fazerem mais sentido quando utilizados com estes. Por exemplo:

```
int *pInt = &(int){-2}; // pInt aponta para uma variável
                           anônima
                        // do tipo int iniciada com -2
```

Literais compostos podem ser bastante práticos em chamadas de funções. Por exemplo:

```
extern int F(int ar[], unsigned int nElementos);

...

int x = F((int []){ -2, 1 }, 2 );
```

Como outro exemplo de uso de literais compostos em chamada de função, considere o trecho de programa a seguir, no qual o endereço de um literal composto é passado para uma função:

```
typedef struct {
           int a;
           float b;
           char c;
         } tEst2;

extern void F2(tEst2 *est);

...

F2(&(tEst2){ -2, 2.2f, 'B' });
```

Dois ou mais literais compostos com o mesmo conteúdo podem representar o mesmo objeto. Por exemplo, os ponteiros `p1` e `p2` no trecho de programa a seguir podem referir-se ao mesmo literal composto:

```
const char *p1 = (const char []){'C', 'B', 'F'};
const char *p2 = (const char []){'C', 'B', 'F'};
```

9.12 Enumerações

Uma enumeração é útil quando se deseja utilizar um conjunto determinado de valores constantes com alguma afinidade entre si que podem estar associados a uma variável. Quando se tenta atribuir a uma variável de um tipo enumeração um valor que não faz parte da própria enumeração, o compilador emite uma mensagem de erro. Uma declaração de variável de um tipo enumeração consiste na palavra-chave **enum** seguida de uma lista de identificadores entre chaves seguidos, finalmente, pelo nome da variável. Além disso, a palavra-chave **enum** pode, opcionalmente, ser seguida por um identificador (rótulo). Mais precisamente, uma declaração de enumeração tem o seguinte formato:

enum *rótulo* {*lista-de-nomes-de-constantes*} *variável1*, ..., *variávelN*;

Quando o rótulo é utilizado, não é necessário declarar nenhuma variável junto com a declaração da própria enumeração. Por exemplo:

```
enum  cores {AZUL, VERMELHO, BRANCO, PRETO};
...
enum cores umaCor;
...
enum cores outraCor;
```

No exemplo acima, `umaCor` e `outraCor` são duas variáveis do tipo enumeração representado pelo rótulo `cores`.

De modo semelhante a estruturas e uniões, pode-se também definir um tipo enumeração usando **typedef**, como:

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

Por exemplo:

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

Observe que, apesar das semelhanças com estruturas e uniões em termos de declarações, enumerações não representam tipos estruturados. Ou seja, as variáveis `umaCor` e `outraCor` dos dois últimos exemplos podem assumir os valores constantes `AZUL`, `VERMELHO`, `BRANCO` ou `PRETO`, mas não podem assumir dois ou mais destes valores simultaneamente.

Valores inteiros são associados aos identificadores de constantes numa enumeração. Caso não haja indicação em contrário, estes valores são baseados simplesmente nas posições das constantes na lista de enumeração, sendo que, como padrão, a primeira constante recebe o valor 0, a segunda constante recebe o valor 1 e assim por diante. No último exemplo, `AZUL` recebe o valor 0, `VERMELHO` recebe o valor 1, `BRANCO` recebe o valor 2 e `PRETO` recebe o valor 3.

Muitas vezes, os valores atribuídos aos identificadores de constantes de uma enumeração não são importantes, mas pode ser que seja necessário

modificar os valores atribuídos como padrão. Por isso, a linguagem C permite que se especifiquem valores de constantes numa enumeração. Se o valor de uma dada constante na enumeração não for definido explicitamente, seu valor será o valor da constante anterior na seqüência acrescido de 1; se o valor da primeira constante não for definido, este será zero. Como exemplo de atribuição explícita de valores aos identificadores de constantes de uma enumeração, considere:

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

No último exemplo, as constantes `VERMELHO` e `PRETO` 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 legibilidade, isto é recomendável.

Um parâmetro de função pode ser de um tipo enumeração e uma função também pode retornar um valor de um tipo enumeração. Por exemplo:

```
typedef enum {AZUL, VERMELHO, BRANCO, PRETO} tCores;

typedef struct {
      char   modelo[20];
           float  potencia 85.4;
           ...
           tCores cor;
        } tCarro;

tCores CorDoCarro(tCarro oCarro)
{
   return oCarro.cor;
}
```

O padrão C99 permite que uma declaração de enumeraçãocontenha uma vírgula sobrando ao final. Em versões anteriores do padrão ISO de C isto não era permitido. Por exemplo, a seguinte declaração de enumeração é legal, de acordo com C99:

```
enum cores {AZUL, PRETO, BRANCO,};
```

9.13 Tipos Incompletos e Compatibilidade de Rótulos

Um **tipo incompleto** pode ser um tipo estrutura ou união cujos

membros ainda não tenham sido pormenorizados, um tipo array cujo número de elementos ainda não tenha sido especificado ou o tipo **void**. Na prática, tipos incompletos servem para fazer alusões a tipos. Considere os seguintes exemplos de tipos incompletos:

```
struct data *pData; /* pData aponta para uma estrutura incompleta
*/

float ar[]; /* O array ar tem tipo incompleto */
```

Pode-se declarar uma variável do tipo **void**, mas nunca defini-la, pois o tipo **void** não pode ser completado. Por exemplo,

```
void umaVarSemUtilidade; /* OK, mas a variável nunca será
alocada */
```

Exceto o tipo **void**, que jamais pode ser completado, um tipo incompleto torna-se completo quando a informação ausente é especificada dentro do escopo onde ele é aludido. No caso de estruturas e uniões incompletas, as informações que devem ser especificadas são os nomes e os tipos dos campos, enquanto no caso de arrays incompletos, deve-se especificar o número de elementos. Por exemplo, no escopo onde aparecem as declarações do último exemplo, devem constar os complementos dos tipos incompletos:

```
struct data {
          short dia, mes,ano;
}

float ar[10];
```

É importante salientar que a completação de um tipo deve ser realmente feita no mesmo escopo onde se encontra o tipo incompleto, visto que tipos não possuem ligação (v. **Seção 4.9**).

De acordo com o padrão C99, dois tipos que representam uma estrutura, união ou enumeração declarados em diferentes unidades de tradução são compatíveis se seus rótulos são iguais. Além disso, se ambos os tipos são completos, os seguintes requisitos adicionais devem ser satisfeitos:

- Deve haver uma correspondência um a um entre seus membros, tal que cada par de membros correspondentes é declarado com tipos compatíveis.
- Se um membro é declarado com um nome, o outro membro correspondente é declarado com o mesmo nome.
- No caso de estruturas, membros correspondentes devem ser declarados na mesma ordem.
- No caso de enumerações, os membros correspondentes devem ter os mesmos valores.

É importante salientar que declarações de estruturas, uniões ou enumerações que se encontram em arquivos diferentes não são consideradas do mesmo tipo, mesmo que sejam provenientes de um mesmo arquivo de cabeçalho incluído nesses arquivos. Isto deve-se ao fato de, em C, unidades de tradução (v. **Seção 5.2**) serem disjuntas entre si. Por isso, o padrão ISO especifica regras de compatibilidade para tais tipos, de modo que as declarações que os utilizam sejam compatíveis se forem suficientemente semelhantes.

De acordo com C99, tipos estruturas e uniões declarados em diferentes unidades de tradução devem possuir rótulos idênticos para serem considerados compatíveis. Em versões anteriores do padrão de C não havia esta exigência.

9.14 Exercícios de Revisão

1. Qual é a principal diferença conceitual entre estruturas e arrays?

2. (a) Descreva os formatos gerais permitidos para a declaração de uma estrutura. (b) Qual desses formatos é mais apropriado e por quê?

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

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

5. Como um array de estruturas é iniciado?

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

7. Como são acessados os campos de uma estrutura?

8. (a) Para que servem os operadores **.** e **->**? (b) Qual é a precedência deles com relação aos demais operadores de C? (c) Qual é a associatividade desses operadores?

9. Como uma estrutura pode ser passada como argumento para uma função?

10. Uma função pode retornar uma estrutura?

11. (a) O que é uma estrutura com auto-referência? (b) Qual é a utilidade de estruturas com auto-referência?

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

13. (a) Como se pode atribuir um valor inicial a um membro de uma união? (b) Qual é a diferença entre iniciações de uniões e estruturas?

14. Suponha que um valor do tipo **short** ocupe 2 bytes e um endereço ocupe 4 bytes. Quantos bytes seriam alocados para o conjunto de declarações a seguir?

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

tRegistro        registroDaPessoa;
tPtrParaRegistro ptrParaRegistro;
```

15. Suponha que o tipo **int** ocupe 4 bytes e o tipo **float** ocupe 8 bytes numa dada implementação. Quais são os espaços ocupados nesta implementação pelas variáveis `uniao` e `registro` a seguir?

```
union {
    char   a;
    int    b;
    float  c;
} uniao;

struct {
    char   a;
    int    b;
    float  c;
} registro;
```

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

17. Suponha que se tenha a seguinte definição de variável:

```
struct {
    char   a;
    int    b;
    float  c;
} estrutura;
```

Como se poderia alterar esta definição de modo a iniciar com o valor 5 apenas o campo `b` da variável `estrutura` utilizando um iniciador designado?

18. (a) O que é um array flexível?

(b) Para que servem arrays flexíveis?

(c) Que regras devem ser seguidas na declaração de um array flexível?

19. (a) O que é e para que serve um literal composto?

(b) O que é uma variável anônima?

(c) Qual é a relação entre literais compostos e variáveis anônimas?

20. (a) O que é uma enumeração?

(b) Para que servem enumerações?

21. (a) Quais são as semelhanças entre estruturas, uniões e enumerações?

(b) Qual é a principal diferença entre enumerações e estruturas (ou uniões)?

22. (a) O que é um tipo incompleto?

(b) Em que situações práticas utilizam-se tipos incompletos?

23. (a) Quando dois rótulos de estruturas são compatíveis?

(b) Por que é necessário que sejam definidas regras de compatibilidade entre tipos que representam estruturas, uniões e enumerações?

9.15 Exercícios de Programação

EP9.1) Defina o tipo `tComplexo`, a ser utilizado na representação de números complexos, da seguinte forma:

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

(a) Considerando esta 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 *umComplexo)
```

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

```
void ImprimeComplexo(const tComplexo *umComplexo)
```

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

```
tComplexo *SomaComplexos(tComplexo *resultado,
                const tComplexo *umComplexo,
                const tComplexo *outroComplexo)
```

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

```
tComplexo *SubtraiComplexos(tComplexo *resultado,
                  const tComplexo *umComplexo,
                  const tComplexo *outroComplexo)
```

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

```
tComplexo *MultiplicaComplexos(tComplexo *resultado,
                     const tComplexo *umCompl,
                     const tComplexo *outroCompl)
```

(b) Escreva um programa em C que leia dois números complexos introduzidos pelo teclado e ofereça ao usuário as opções de soma, subtração e multiplicação de complexos. Após a execução da operação escolhida pelo usuário, o programa deve imprimir o resultado na tela.

EP9.2) Escreva uma função que receba como argumento uma estrutura do tipo `tData`, definido na **Seção 9.4**, e retorne 1 se este argumento representar uma data válida ou 0 em caso contrário.

Sugestão

Este problema não é tão trivial quanto possa parecer, pois existem anos bissextos, de modo que uma data, como 29/02, é válida num ano bissexto mas inválida num ano que não é bissexto. As demais sugestões para este exercício são semelhantes àquelas apresentadas no exercício **EP4.3**.

EP9.3) Escreva uma função que receba como parâmetros duas estruturas do tipo `tData`, definido na **Seção 9.4**, representando datas válidas e retorne o número de dias decorridos entre as duas datas. O primeiro parâmetro deve representar uma data anterior à data representada pelo segundo parâmetro. Quando os parâmetros não satisfazem as condições especificadas, esta função deve retornar um número negativo, indicando que não foi possível determinar o número de dias decorridos entre as datas.

Sugestão

Consulte as sugestões apresentadas para os exercício **EP9.2** e **EP4.3**.