Capítulo 4

ESTRUTURAS E UNIÕES

4.1 Estruturas

Estruturas são tipos de dados **agregados** (ou **estruturados**) similares a arranjos. Entretanto, estruturas diferem de arranjos por permitirem que seus elementos, aqui denominados de **campos** (ou **membros**), sejam de tipos diferentes. Por causa desta característica, estruturas são consideradas tipos de dados **heterogêneos**. Em algumas linguagens de programação, estruturas são conhecidas como **registros**.

4.1.1 Definição 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 para construção de identificadores de C. Existem várias formas sintáticas permitidas para a declaração de uma estrutura em C, mas estas formas de declaração podem ser resumidas dois esquemas gerais: (1) ***sem*** definição de tipo e (2) ***com*** definição de tipo. A seguir estas formas serão apresentadas e exemplificadas:

Esquema 1: Sem Definição de Tipo

```
struct  <rótulo da estrutura> {
      <tipo 1>  <campo 1>;
      <tipo 2>  <campo 2>;
          .
          :
      <tipo N>  <campo N>;
} <lista de variáveis 1>;

struct   <rótulo da estrutura>  <lista de variáveis 2>;
...
struct   <rótulo da estrutura>  <lista de variáveis N>;
```

Neste formato, tanto *<rótulo da estrutura>* quanto *<lista de variáveis 1>* são opcionais, mas, na prática, se ambos forem simultaneamente abandonados, não se poderá ter nenhuma variável e a declaração de estrutura não terá nenhuma utilidade. Na realidade, utilizando este esquema, raramente, o rótulo é abandonado na prática, pois isto implicaria em declarar todas as variáveis daquele tipo de estrutura no local de declaração da própria estrutura.

Utilizando este 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 acima, `registro` é o rótulo da estrutura, que é utilizado tanto na declaração da estrutura (primeira declaração) quanto na definição da variável (segunda declaração). A declaração de uma estrutura é similar a uma definição de tipo e, portanto, não causa alocação de memória. Observe que campos de um mesmo tipo podem ser declarados juntos e separados por vírgulas como em declarações de variáveis de um mesmo tipo.

Do mesmo modo que uma declaração de tipo, o rótulo de uma estrutura pode ser utilizado para declaração de quaisquer outras variáveis. Por exemplo, suponha que, além da variável `registroDaPessoa`, também se deseje um ponteiro para uma estrutura do tipo `registro`, então poder-se-ía declarar um tal ponteiro como:

```
struct   registro  *ptrParaRegistro;
```

Esse formato de declaração difere de uma declaração de tipo porque ela ainda requer o uso da palavra **struct** na declaração da variável.

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 declaradas no mesmo local aonde `registroDaPessoa` é declarada. Conseqüentemente, este formato de declaração é recomendados em situações aonde a estrutura precisa ser utilizada em apenas uma parte do programa (usualmente, num único arquivo). Por exemplo, se quiséssemos ainda acrescentar a variável `ptrParaRegistro` para ser utilizada no mesmo arquivo em que esta declaração aparece, poderíamos escrever:

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

Neste último exemplo, se o ponteiro `ptrParaRegistro` precisasse ser declarado em outro arquivo (e não no mesmo arquivo aonde a variável `registroDaPessoa` é declarada), teríamos que repetir a declaração da estrutura no arquivo aonde `ptrParaRegistro` é declarado, como:

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

Esse esquema de declaração de estruturas é considerado em desuso[1], mas ainda é possível encontrar muitos programas que o utilizam.

Esquema 2: Com Definição de Tipo

O uso de uma definição de tipo é a melhor forma de declaração de uma estrutura. Esta forma de declaração é apresentada a seguir:
Aqui, define-se um novo tipo que é sinônimo de tudo que precede sua definição (v. **Seção 2.6**), 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 mais adiante.

```
typedef  struct  <rótulo da estrutura> {
     <tipo 1>  <campo 1>;
     <tipo 2>  <campo 2>;
         .
         :
     <tipo N> : <campo N>;
} <nome do tipo>;

<nome do tipo>  <lista de nomes de variáveis>;
```

[1] É oportuno ressaltar que o termo desuso se refere à programação em C, pois em C++ o rótulo de uma estrutura é equivalente a uma definição de tipo, o que torna o uso de **typedef** redundante nesse caso.

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;  /* Note que não se deve       */
                                    /* utilizar asterisco aqui!!! */
```

Exercício: Quantos bytes seriam alocados para este conjunto de declarações? (Suponha uma implementação aonde um **short** ocupa 2 bytes e um endereço ocupa 4 bytes.)

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-ía declarando um ponteiro para ponteiro.

Usualmente, quando uma estrutura é utilizada em várias partes de um programa com vários arquivos, 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. Aliás, esta prática deve ser seguida para quaisquer declarações de tipos (e não apenas para estruturas) que são utilizados em vários arquivos.

É interessante observar que 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 referenciados (v. mais adiante), não existe perigo de colisão de identificadores de campos no exemplo acima. Também, é permitido o uso de um mesmo nome para um rótulo, nome de variável e nome de campo de uma estrutura, como por exemplo:

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

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

4.1.2 Inicialização de Estruturas

Uma estrutura pode ser inicializada de modo similar a um arranjo. Isto é, a estrutura a ser inicializada deve ser seguida pelo sinal de igualdade e uma lista de valores entre chaves. O número de valores de inicialização 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` vista anteriormente, a variável `registroDaPessoa` poderia ser inicializada como:

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

Obviamente, não se pode inicializar uma definição de tipo ou de rótulo de estrutura, pois estas declarações não alocam espaço em memória. Por exemplo, a seguinte tentativa de inicialização seria inválida:

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

4.1.3 Acesso a Campos de Estruturas

Existem duas formas de se acessarem os campos de uma estrutura, dependendo do fato de se ter uma variável do tipo estrutura ou um ponteiro para uma estrutura. No primeiro caso, utiliza-se o operador ponto (`.`) e, no segundo caso, 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 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;
```

Note que o ponteiro `ptrParaRegistro` foi inicializado com o endereço da variável `registroDaPessoa`. Portanto, 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`

Note que, na realidade, o operador **`->`** é uma abreviação para as operações conjuntas de referenciação do ponteiro e acesso a um campo utilizando o operador ponto. Por exemplo:

```
ptrParaRegistro->ano
```

é o mesmo que:

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

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

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

ou, eqüivalentemente:

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

Os operadores `.` e `->`, juntamente com os operadores `[]` e `()`, fazem parte de um mesmo grupo de precedência. 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`

4.1.4 Arranjos de Estruturas

Um arranjo de estruturas pode ser criado em C precedendo-se o nome do arranjo por um tipo estrutura previamente definido. Por exemplo, considerando a declaração do tipo `tRegistro` vista anteriormente, pode-se declarar o arranjo `arranjoDePessoas` como:

```
tRegistro  arranjoDePessoas[20];
```

Utilizando arranjos de estruturas, pode-se agora apresentar um exemplo mais completo e prático do uso de estruturas. Suponha que se tenha um arranjo contendo 20 estruturas do tipo `tRegistro` e que se deseje contar o número de pessoa nascidas entre dois determinados anos (por exemplo, entre 1960 e 1980). Esta contagem poderia ser realizada pela função a seguir:

```
unsigned   int  NumeroDePessoasEntreAnos( tRegistro pessoas[],
                                          int  numeroDePessoas,
                                          int  anoInicial,  int  anoFinal )
{
     unsigned  int  i, contador = 0;

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

     return  contador;
}
```

Um programa que utilizasse a função definida acima poderia ter o seguinte formato:

```
main()
{
     tRegistro  arranjoDePessoas[20];

       .      /* Atribuição de valores aos registros do arranjo */
       :

     printf("O numero de pessoa nascidas entre 1960 e 1980 é %d\n",
             NumeroDePessoasEntreAnos(arranjoDePessoas, 20, 1960,1980));
}
```

A função `NumeroDePessoasEntreAnos()` definida acima funciona perfeitamente conforme especificado. No entanto, esta função possui um inconveniente em termos de eficiência: cada vez que o laço **for** desta função é executado, são feitas duas referências ao arranjo `pessoas` por meio do índice `i` nas expressões:

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

Recorde-se do capítulo anterior que cada referência a um arranjo por meio de um índice é traduzida pelo compilador como uma referência da soma (com *scaling*) do ponteiro para o elemento inicial do arranjo com este índice. No caso corrente, cada referência `pessoas[i]` é traduzida pelo compilador como:

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

O problema de eficiência ocorre quando a operação de *scaling* é efetuada, pois então, o valor `i` deve ser multiplicado pelo tamanho de cada elemento do arranjo em bytes (neste caso, o tamanho do tipo `tRegistro`) antes de ser somado ao ponteiro `pessoas`. Será que esta complexidade pode ser reduzida? A resposta é: *sim*, a complexidade pode ser reduzida através do uso de ponteiros e incrementos destes, ao invés do uso de índices, para acessar os elementos do arranjo.

Para utilizar este método, recorde-se novamente do capítulo anterior que `pessoas` é interpretado como um ponteiro para o primeiro elemento do arranjo. Portanto, este elemento pode ser acessado por `*pessoas`, e os elementos seguintes podem ser acessados incrementando-se sucessivamente o ponteiro `pessoas`. Uma nova versão da função `NumeroDePessoasEntreAnos` poderia então ser escrita como:

```
unsigned int   NumeroDePessoasEntreAnos( tRegistro  pessoas[],
                                         int  numeroDePessoas,
                                         int  anoInicial,  int  anoFinal )
{
     unsigned  int  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 `NumeroDePessoasEntreAnos()`, as operações de multiplicação necessárias para *scaling* são evitadas, e o tempo economizado com esta estratégia pode ser substancial dependendo do tamanho do arranjo (por exemplo, com um arranjo 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. Estas duas versões da função `NumeroDePessoasEntreAnos()` são capazes de diferenciar um programador novato de um programador experiente em C.

Um ponto importante a ser observado quando se incrementa um argumento de uma função que é um ponteiro para um arranjo (como o argumento `pessoas` na função `NumeroDePessoasEntreAnos()` do último exemplo), com a finalidade de acessar os elementos do arranjo num laço de repetição, é que, ao final do laço, este argumento não estará mais apontando para o início do arranjo[2]. Conseqüentemente, se houver necessidade de a função processar novamente o arranjo a partir de seu início, é necessário que antes se estabeleça um ponteiro para o início do arranjo. Uma forma de se contornar este problema sem possibilidade de introduzir erro é utilizar um ponteiro auxiliar, ao invés do próprio argumento, para percorrer o arranjo. Por exemplo, se a função `NumeroDePessoasEntreAnos()` do último exemplo precisasse processar novamente o arranjo após o laço **for**, poder-se-ía utilizar a seguinte declaração no início da função:

```
tRegistro  *ptrAux = pessoas; /* Lembre-se de não utilizar "&" */
                              /* antes de pessoas             */
```

Então, com `ptrAux` apontando para o início do arranjo, podem-se substituir todas as ocorrências de `pessoas` no laço **for** por `ptrAux`. Portanto, ao final deste laço, apesar de o

[2] No último exemplo, ao final do laço **for** o ponteiro `pessoas` não estará apontando nem mesmo para algum elemento do arranjo; i.e., ele estará apontando para o próximo endereço ***além*** do último elemento do arranjo.

ponteiro `ptrAux` não mais apontar para o arranjo, o ponteiro `pessoas` continuará apontando para o início do arranjo.

Uma outra estratégia para retornar ao início do arranjo consiste em ***desfazer*** as modificações sofridas pelo ponteiro dentro do laço **for**[3]. Esta estratégia, no entanto, é sensível a erros e deve ser evitada (ou, pelo menos, utilizada com muito cuidado).

O uso de um ponteiro auxiliar para percorrer um arranjo é sempre recomendado, mesmo quando o arranjo não é processado após o laço **for**, pois, numa versão futura da função, tal processamento posterior poderá ser requerido.

4.1.5 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 uma **estrutura aninhada**. Por exemplo, a definição de estrutura `tRegistro` vista anteriormente poderia ser rescrita como:

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

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

Agora, a estrutura `tRegistro` é uma estrutura aninhada pois o segundo campo desta estrutura (denominado `data`) também é uma estrutura. Existem outras formas de se definirem estruturas aninhadas, mas o uso de **typedef**s ainda é o mais recomendado.

É importante notar que ambas as definições de `tRegistro` descrevem objetos que ocupam o mesmo espaço em memória. Também, apesar de aparentemente mais complexa, a última definição de `tRegistro` é mais elegante pois denota uma melhor organização hierárquica de dados.

Quando uma estrutura aninhada é inicializada, utilizam-se chaves do mesmo modo que em inicializações de arranjos multidimensionais. Por exemplo:

```
tRegistro   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
```

Agora, 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 `.`.

---

[3] 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.

O acesso por meio de ponteiros é similar. Suponha que `ptrParaRegistro` é 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 aninhamento de estruturas. No entanto, um número de níveis de níveis de aninhamento muito grande pode prejudicar a legibilidade não apenas da definição da estrutura resultante como também das expressões utilizadas para acessar os campos mais internos.

4.1.6 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;
```

4.1.7 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 de **estruturas com auto-referência**, e são bastante úteis para a criação de listas e outras estruturas de dados encadeadas (v. mais adiante).

É interessante notar ainda que ponteiros podem fazer referências a estruturas que ainda não foram declaradas, 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;
};
```

Note que cada estrutura do último exemplo possui um ponteiro para a outra estrutura, e portanto estas estruturas são consideradas como estruturas com referência mútua.

Finalmente, note que auto-referências ou referências mútuas não são permitidas com o uso de **typedef**s ***sem rótulos***. Por exemplo, a seguinte tentativa de auto-referência não é permitida:

```
typedef  struct  {
     int  a, b;
     tE  *ptrParaE;     /* ILEGAL: tE ainda não foi declarado */
} tE;
```

4.1.8 Estruturas como Parâmetros de Funções

Existem duas maneiras de se passar uma estrutura como argumento para uma função: (1) passagem da própria estrutura (denominada **passagem por valor**), ou (2) passagem de um ponteiro para a estrutura (denominada **passagem por referência**). Dependendo do método utilizado, o argumento na declaração da função deve ser declarado como uma estrutura (no caso de **passagem por valor**) ou como um ponteiro para uma estrutura (no caso de **passagem por referência**). Considere, por exemplo, o tipo `tRegistro` definido em seções anteriores. As funções `F1()` e `F2()`, apresentadas a seguir, ilustram declarações de argumentos para passagem por valor (`F1`) e por referência (`F2`) de uma estrutura do tipo `tRegistro`:

```
void F1(tRegistro  reg)  /* Passagem por valor; o argumento é uma estrutura */
{
     ...
}

void  F2(tRegistro  *ptrReg)     /* Passagem por referência;  */
                                 /* 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 declaração de variável-ponteiro:

```
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. (Convença-se de que realmente entendeu isso antes de prosseguir.)

Na passagem por referência de uma estrutura (como na função `F2` acima), qualquer alteração de valor em algum campo da estrutura dentro da função é comunicada para a função que fez a chamada. Por outro lado, na passagem por valor (como na função `F1` acima), qualquer alteração feita dentro da função incide, na realidade, sobre uma cópia da

estrutura; consequentemente, a estrutura passada para a função permanece inalterada após a chamada.

Existem duas situações aonde uma estrutura deve ser passada por valor:

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

2. Quando se deseja garantir que a função não modifica a estrutura sendo passada como argumento.

Em outras situações, estruturas devem ser passadas por referência.

Note que 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. Isto é, se a passagem é por valor, uma estrutura será utilizada, e, portanto, o operador `.` deve ser utilizado em referências aos campos da estrutura. Por outro lado, se a passagem é por referência, um ponteiro para estrutura será utilizado; portanto, o operador `->` deve ser utilizado em referências aos campos da estrutura.

O aprendiz de C deve atentar para o fato de passagens de estruturas para funções ser bem diferente de passagens de arranjos. Para passar um arranjo como parâmetro real numa chamada de função, deve-se simplesmente utilizar o nome do arranjo (sem subscritos). O compilador interpreta o nome do arranjo como um ponteiro para o elemento inicial do arranjo. 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 arranjos e estruturas também deve ser notada na definição de parâmetros formais de funções. Por exemplo, as duas definições de parâmetros do tipo arranjo a seguir são idênticas:

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

Entretanto, as seguintes definições de parâmetros do tipo estrutura são bem diferentes:

```
void  F3(struct  registro estr) /* estr representa uma estrutura inteira  */
void  F4(struct  registro *estr) /* estr é um ponteiro para uma estrutura */
```

4.1.9 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 deve ser compatível com o valor realmente retornado. 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` definido anteriormente:

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

Existem dois detalhes a serem 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 ao retorno da função e o endereço retornado apontaria para uma posição de memória que não está mais alocada. No caso da função `F3()`, aonde a própria estrutura é retornada, esta exigência não se faz necessária.

Normalmente, 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: retornar um ponteiro é muito mais eficiente do que retornar uma estrutura[4].

O retorno de uma estrutura ou ponteiro para uma estrutura pode ser recomendado em situações aonde 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 mais de um valor embutidos nos campos da estrutura.

4.2 Alocação Dinâmica de Memória

4.2.1 Introdução à Alocação Dinâmica de Memória

Uma variável de duração fixa tem memória reservada durante todo o tempo de execução do programa, enquanto que uma variável de duração automática é alocada cada vez que seu escopo é executado. Ambas as abordagens de alocação de memória, no entanto, assumem que o programador sabe, no instante de escrita do programa, que quantidade de memória será necessária para a execução do mesmo. Existem muitas situações, entretanto, em que a quantidade de memória necessária não pode ser determinada precisamente em tempo de programação. Isto é, freqüentemente, deseja-se ter a capacidade de alocar memória de acordo com a demanda apresentada durante a execução do programa. Um exemplo clássico de uma situação dessas é o de um programa para gerenciamento de lista de espera de uma companhia aérea. Neste caso, a quantidade de dados (i.e., registros de passageiros) pode variar bastante entre uma execução do programa e outra (ou, pelo menos, de uma temporada para outra). Por exemplo, num período de baixa temporada, pode não haver nenhum passageiro em lista de espera; enquanto que num período de alta temporada, o número de passageiro poderá atingir um número máximo indeterminado.

Existem duas formas de abordagem para um programa como esse de reserva de passagens. A mais fácil, mas menos eficiente, consiste em especificar um valor máximo de dados de entrada (por exemplo, 200 passageiros), e utilizar um arranjo capaz de conter estes dados (por exemplo, um arranjo de tamanho 200, onde cada elemento contém um registro de passageiro). No caso do programa de reserva de passagens, poder-se-íam ter as seguintes declarações:

```
#define   MAXIMO_DE_PASSAGEIROS  200

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

typedef  struct  {
            char        nome[30];
            tData       dataDeNascimento;
            char        identidade[12];
            char        numeroDoBilhete[10];
          } tRegistro;
```

[4] Existe uma outra forma de se retornar um ponteiro para uma estrutura diferente daquela utilizada pela função `F4()`. Este método utiliza alocação dinâmica de memória que será visto na próxima seção.

```
tRegistro  listaDeEspera[MAXIMO_DE_PASSAGEIROS];
```

Existem dois problemas com esta abordagem. Primeiro, o programador deve estabelecer um máximo arbitrário, e isto não é bom porque talvez, mais adiante, este número precise ser acrescido (implicando, pelo menos, na devida recompilação do programa). O segundo e maior problema desta abordagem é que quanto maior for o máximo estipulado, maior será o desperdício de memória em execuções do programa que não utilizem todo este máximo. Por exemplo, num período de baixa estação, onde houvesse, no máximo, 5 passageiros em lista de espera o programa do último exemplo estaria desperdiçando, em média, 195 vezes o espaço em memória necessário para conter um elemento do tipo `tRegistro`. Além disso, durante esse período de subutilização de memória, haveria também desperdício de tempo para a alocação de memória que não seria efetivamente utilizada durante aquele período.

A melhor solução para problemas como este, onde a quantidade de dados pode variar drasticamente entre várias execuções de um programa, é a alocação de memória durante a execução do programa e de acordo com a demanda de dados. Este tipo de alocação é denominado **alocação dinâmica de memória**, e contrasta com qualquer outro tipo de alocação de memória visto anteriormente, denominado **alocação estática de memória**, cujo espaço a ser alocado é conhecido em tempo de programação (i.e., antes mesmo de o programa ser executado). Tipos de dados alocados estaticamente são denominados **tipos de dados estáticos**, enquanto que tipos de dados alocados dinamicamente são denominados **tipos de dados dinâmicos**. Todos os tipos de dados vistos até aqui são tipos de dados estáticos. Um exemplo de tipo de dado dinâmico (i.e., cujo tamanho pode aumentar ou diminuir durante a execução do programa) são as listas encadeadas, que serão vistas mais adiante.

4.2.2 Funções de Alocação Dinâmica de Memória

Alocação dinâmica de memória em C é feita por meio de ponteiros e das quatro funções de biblioteca resumidas na **Tabela 1**. Para utilizar estas funções inclua o arquivo `stdlib.h` ou o arquivo `alloc.h`.

FUNÇÃO	DESCRIÇÃO RESUMIDA
malloc()	Aloca um número especificado de bytes em memória, e retorna um ponteiro para o início do bloco de memória alocado.
calloc()	Similar a **malloc()**, mas esta função inicializa todos os bytes alocados com zeros. Esta função também permite a alocação de memória para mais de um bloco numa mesma chamada.
realloc()	Modifica o tamanho de um bloco previamente alocado.
free()	Libera o espaço de um bloco de memória previamente alocado com **malloc()**, **calloc()** ou **realloc()**.

Tabela 1: Funções de Alocação Dinâmica de Memória

A função **malloc**() recebe como argumento o tamanho, em bytes, do bloco a ser dinamicamente alocado. Usualmente, este argumento envolve o operador **sizeof**[5]. Por exemplo, supondo que `prtParaRegistro` seja um ponteiro para o tipo `tRegistro`, então a chamada seguinte:

```
prtParaRegistro = malloc(sizeof(tRegistro));
```

alocaria (se fosse possível) um bloco capaz de conter uma estrutura do tipo `tRegistro` e retornaria um ponteiro contendo o endereço inicial deste bloco.

A função **calloc**() recebe dois argumentos: o primeiro é o número de blocos a serem alocados, e o segundo é o tamanho de cada bloco. Quando possível, esta função aloca o

[5] O uso deste operador é recomendado, principalmente, por questões de portabilidade.

espaço necessário para conter os blocos requisitados, e retorna um ponteiro para o início do primeiro bloco alocado. Todos os bits do espaço alocado são inicializados com zeros.

A função **realloc**() recebe dois argumentos. O primeiro argumento deve ser um ponteiro para um bloco de memória alocado utilizando **malloc**(), **calloc**() ou a própria função **realloc**(), e o segundo argumento especifica um novo tamanho desejado para o bloco. Se o novo tamanho for menor do que o tamanho atual do bloco, a diferença será retirada do final do bloco (i.e., a porção final do bloco, cujo tamanho é a diferença entre o tamanho original e o novo tamanho, será liberada). Ainda neste caso, o ponteiro retornado pela função apontará para o mesmo endereço que apontava anteriormente, e a porção de memória que não foi liberada manterá seu conteúdo original. Se o novo tamanho for maior do que o tamanho atual do bloco, um novo bloco do tamanho especificado será alocado (se for possível), o conteúdo do bloco original será copiado para a porção inicial deste novo bloco, o bloco original será liberado, e finalmente a função retornará um ponteiro para o novo bloco alocado. A porção final do novo bloco não será inicializada. Nesta última situação, se não for possível alocar um novo bloco, o ponteiro e o bloco antigos manterão seus valores originais, e a função retornará `NULL`. Se um ponteiro nulo for passado (como primeiro argumento) para a função **realloc**(), ela se comportará como **malloc**() para o tamanho especificado (segundo argumento). Se o novo tamanho for igual a zero e o ponteiro passado não for nulo, a chamada de **realloc**() comporta-se como uma chamada da função **free**() (i.e., libera o espaço em memória apontado pelo ponteiro).

A função **free()** recebe como único argumento um ponteiro que aponta para uma posição de memória alocada utilizando **malloc()**, **calloc()** ou **realloc()**, e libera este espaço em memória. Após uma chamada da função **free()**, você não deve mais utilizar o ponteiro utilizado nesta chamada para acessar o espaço de memória liberado; caso contrário, seu programa poderá *quebrar*. Se o ponteiro passado para **free()** for nulo, a função retorna sem executar nada.

Cuidado: Não utilize como argumento das funções **realloc()** ou **free()** um ponteiro que não esteja correntemente apontando para o ***início*** de um bloco de memória alocado com alguma das funções de alocação descritas aqui, pois o resultado será imprevisível (provavelmente, o programa irá *quebrar*).

O tipo de dados do tamanho do bloco de memória, requerido pelas funções de alocação de memória, é **size_t**, definido no arquivo `stddef.h`. Este tipo, que usualmente é definido como sendo **unsigned int** ou **unsigned long**, também é o tipo do valor resultante da aplicação do operador **sizeof**. Normalmente, você não precisa preocupar-se com isto quando estiver programando, mas é importante conhecer o significado deste tipo porque todas as funções de alocação o utilizam em suas definições.

É importante observar também que todas as funções de alocação de memória [**malloc**(), **calloc**() e **realloc**()] retornam um ponteiro nulo quando não é possível alocar a quantidade de memória requerida (devido, por exemplo, à fragmentação de *heap*[6]). Portanto, é considerado boa prática de programação testar o valor do ponteiro retornado com `NULL` antes de tentar utilizar este ponteiro para acessar uma porção de memória que não se tem certeza se foi realmente alocada. Caso o ponteiro retornado tenha valor nulo, o programador deve tomar as devidas providências antes de prosseguir (talvez apresentando uma mensagem de erro e abortando *graciosamente* o programa, se esta porção de memória é crucial para o prosseguimento do processamento). Por exemplo:

```
prtParaRegistro = malloc(sizeof(tRegistro));

if  (prtParaRegistro != NULL){
 /* Aqui a manipulação de memória pode ser feita normalmente com segurança */
}
else {
  /* Aqui o programador deve tomar as providências cabíveis quando     */
```

[6] *Heap* é a partição da memória alocada para execução de um programa reservada para alocação dinâmica de memória. *Fragmentação de heap* ocorre em conseqüência de várias alocações e liberações de blocos de tamanhos variados durante a execução do programa. Quando isso ocorre, pode ser impossível alocar um bloco de memória contíguo, mesmo que a quantidade total de memória disponível no *heap* seja maior do que o tamanho deste bloco. Alguns sistemas provêem esquemas de compactação que visam combater fragmentação de heap.

```
  /* não é possível alocar o espaço de memória necessário. Talvez seja */
  /* preciso abortar o programa, mas pode ser que haja alternativa     */
  /* menos drástica, dependendo da situação.                       */
}
```

Note que o teste:

```
if  (prtParaRegistro != NULL)
```

pode ser escrito, de forma eqüivalente, como:

```
if  (prtParaRegistro)
```

Esta última forma é a preferida por muitos programadores de C. Outros programadores preferem ainda utilizar:

```
if  ((prtParaRegistro = malloc(sizeof(tRegistro))) != NULL) {
   ...
}
```

ou ainda:

```
if  (prtParaRegistro = malloc(sizeof(tRegistro))) {
   ...
}
```

ao invés de:

```
prtParaRegistro = malloc(sizeof(tRegistro));

if  (prtParaRegistro != NULL){
   ...
}
```

Entretanto, em termos de estilo, não é recomendável misturar chamada de função com teste do valor retornado pela mesma.

A macro `NULL` é definida no arquivo `stddef.h`[7]. Portanto, para utilizá-la, inclua este arquivo no seu programa ou defina você mesmo esta macro (v. **Seção 3.3**).

Em algumas bibliotecas antigas, **malloc**(), **calloc**() e **realloc**() retornam um ponteiro para o tipo **char**. Portanto, nestes compiladores, deve-se fazer um *casting* do tipo de retorno de uma chamada de uma destas funções para o tipo do ponteiro desejado. No exemplo anterior, a chamada deveria ser substituída por:

```
prtParaRegistro = (tRegistro *) malloc(sizeof(tRegistro));
```

Compiladores ANSI/ISO mais recentes retornam um ponteiro de tipo genérico, a ser visto mais adiante, e este ponteiro não precisa de *casting*.

4.3 Ponteiros Genéricos

Recorde-se que, em C, dois ponteiros são compatíveis apenas quando eles apontam para tipos de dados da mesma espécie. Existe, entretanto, na linguagem C, o conceito de **ponteiro genérico**, que é um ponteiro compatível com ponteiros para quaisquer tipos de dados. Por definição, um ponteiro genérico é um ponteiro para o tipo **void**. Por exemplo, o ponteiro `ponteiroGenerico` a seguir é um ponteiro genérico:

```
void   *ponteiroGenerico;
```

Ponteiros genéricos são normalmente utilizados em duas situações:

---

[7] Em compiladores antigos, `NULL` pode ser definida no arquivo `stdio.h`.

(1) como tipos de **retorno de funções**, e
(2) como tipos de **argumentos de funções**.

No primeiro caso, ponteiros genéricos são utilizados para representar ponteiros que podem ser implicitamente convertidos para ponteiros de quaisquer tipos. Por exemplo, as funções de alocação de memória **malloc()**, **calloc()** e **realloc()** retornam ponteiros genéricos, e isto significa que o valor retornado por estas funções pode ser atribuído a um ponteiro de qualquer tipo sem a necessidade de *casting* explícito[8]. Por exemplo, conforme já foi visto anteriormente se `prtParaRegistro` é um ponteiro para o tipo `tRegistro`, um objeto deste tipo pode ser alocado com o uso de **malloc()** através da chamada:

```
prtParaRegistro = malloc(sizeof(tRegistro));
```

Em alguns compiladores antigos, **malloc()**, **calloc()** e **realloc()** retornam um ponteiro para o tipo **char**. Portanto, nestes compiladores, deve-se fazer um *casting* do tipo de retorno de uma chamada de uma destas funções para o tipo do ponteiro. No exemplo anterior, a chamada deveria ser substituída por:

```
prtParaRegistro = (tRegisto *) malloc(sizeof(tRegistro));
```

No segundo caso de uso de ponteiros genéricos, estes ponteiros são utilizados para representar argumentos compatíveis com qualquer tipo de ponteiro. Um exemplo é a função **free()**, que possui o seguinte protótipo:

```
void  free(void  *ptr);
```

A definição da função **free()** permite a passagem de um ponteiro de qualquer tipo para a função sem necessidade de *casting* explícito. Por exemplo, a chamada seguinte poderia ser utilizada para liberar o espaço alocado com a chamada de **malloc()** do último exemplo:

```
free(prtParaRegistro);
```

4.4 Estruturas Recursivas e Listas Encadeadas

4.4.1 Introdução às Estruturas Dinâmicas

Voltando ao nosso exemplo de reserva de passagens, um programa com tal finalidade poderia solicitar ao usuário o número máximo de elementos a serem alocados para o arranjo `listaDeEspera[]`, conforme mostrado no trecho de programa a seguir:

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

typedef  struct  {
             char         nome[30];
             tData        dataDeNascimento;
             char         identidade[12];
             char         numeroDoBilhete[10];
         } tRegistro;

tRegistro        *listaDeEspera;
unsigned  int    numeroDeRegistros;
       .
       :
printf("Introduza o número máximo de passageiros na lista de espera: ");
scanf("%d", &numeroDeRegistros);

listaDeEspera = malloc(numeroDeRegistros*sizeof(tRegistro));
if (listaDeEspera) {
       .            /* Processamento da lista  */
```

[8] Essa é a única situação em que a linguagem C faz conversão implícita de ponteiros.

```
        :
}
```

No programa acima, `numeroDeRegistros` é uma variável que irá representar o tamanho do arranjo alocado dinamicamente (i.e., durante a execução do programa). Esta variável é lida com uma chamada de **scanf()**, e de posse do valor introduzido, o arranjo é alocado com a chamada da função **malloc()**. Nesta chamada, o espaço necessário para conter o arranjo é especificado por:

```
numeroDeRegistros*sizeof(tRegistro)
```

o que é correto, pois cada elemento ocupa um espaço, em bytes, dado por `sizeof(tRegistro)`. Portanto, o espaço total a ser ocupado pelo arranjo é dado pelo produto do número de elementos do arranjo pelo espaço ocupado por cada elemento.

Observe que, como todos os elementos do arranjo são alocados como um único bloco, é garantido que estes elementos estarão em posições contíguas de memória. Consequentemente, estes elementos podem ser acessados por meio de índices do mesmo modo que um arranjo convencional (i.e., alocado estaticamente quando o programa é carregado em memória) visto anteriormente. É ainda mais importante notar que, se estes elementos fossem alocados individualmente por meio de chamadas sucessivas de **malloc()** (ou alguma outra função de alocação), essa contigüidade não estaria garantida. Por exemplo, suponha que `p1`, `p2` e `p3` são ponteiros para o tipo `tRegistro`, então as chamadas:

```
p1 = malloc(sizeof(tRegistro));
p2 = malloc(sizeof(tRegistro));
p3 = malloc(sizeof(tRegistro));
```

não garantem que as posições em memória apontadas por `p1`, `p2` e `p3` sejam adjacentes (i.e., a posição apontada por `p2` pode não começar logo após o final do espaço em memória apontado por `p1`, e a posição apontada por `p3` pode não começar logo após o final do espaço apontado por `p2`). Pode ser que, neste caso, esta contigüidade ***eventualmente*** aconteça, mas o programador não deve jamais confiar nisto.

A solução apresentada até aqui para o problema da lista de espera ainda não é a solução ideal para o mesmo, pois ainda deve ser especificado um número máximo de passageiros a cada execução do programa e antes do processamento da lista propriamente dito, mas já apresenta uma melhora com relação à primeira solução que aloca apenas um valor máximo em qualquer execução. A melhor solução seria a alocação de espaço à medida em que se precisasse dele (i.e., alocação de um registro à medida em que surge um novo candidato à lista de espera). Esta solução será esboçada a seguir.

4.4.2 Listas Encadeadas

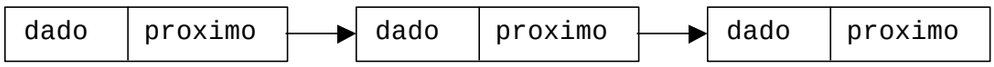
Foi visto na seção anterior que uma solução para um problema no qual a quantidade de memória alocada não pode ser determinada a priori é a alocação dinâmica de memória. Ainda naquela seção, foi apresentado um exemplo no qual um arranjo tinha seu tamanho especificado em tempo de execução do programa. Mas, mesmo neste caso, a solução não era ideal pois aquela especificação de tamanho era apenas uma previsão de demanda para uma dada execução, de modo que a alocação prevista poderia não corresponder exatamente à demanda real. Portanto, a melhor solução para o problema é alocar memória à medida em que esta se faz realmente necessária. No caso do problema específico da lista de espera, esta solução corresponde a alocar um registro no instante em que for necessário introduzir mais um passageiro na lista de espera. Acontece que, conforme foi visto na seção anterior, quando registros são alocados individualmente não se tem a garantia de que estes registros serão alocados em posições adjacentes e, consequentemente, estes registros não podem ser acessados seqüencialmente como um arranjo. A solução para este problema de acesso a elementos alocados individualmente está no uso de listas encadeadas.

Uma lista encadeada é uma estrutura de dados composta de elementos usualmente denominados de **nós**. Cada nó numa lista encadeada pode ser conceitualmente dividido em duas partes:

(1) uma parte contendo a informação (dados) propriamente dita (esta parte será doravante identificada por `dado`); e

(2) um ponteiro para o próximo elemento da lista (esta parte será doravante identificada por `proximo`).

O diagrama a seguir ilustra uma lista simplesmente encadeada.



Este tipo de lista é denominado ***lista simplesmente encadeada*** porque os ponteiros apontam numa única direção. Existem listas, denominadas ***duplamente encadeadas*** que possuem ponteiros que apontam em ambas as direções. Apenas listas simplesmente encadeadas serão abordadas aqui.

Os nós de uma lista (simplesmente) encadeada podem ser implementados em C como estruturas contendo dois campos que correspondem às partes componentes dos nós. Por exemplo, as declarações a seguir poderiam ser utilizadas para definir o tipo dos nós de uma lista encadeada cujos nós contêm informações de passageiros em lista de espera[9]:

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

typedef  struct {
            char        nome[30];
            tData       dataDeNascimento;
            char        identidade[12];
            char        numeroDoBilhete[10];
} tRegistro;

typedef  struct  no {
            tRegistro   dado;
            struct  no  *proximo;
} tNo;

tNo   *listaDeEspera;
```

Observe que, na última declaração de tipo acima, foi utilizado um rótulo para a estrutura, de modo a tornar possível a auto-referência no segundo campo desta estrutura. Como o exemplo acima sugere, o campo `dado` dos nós de uma lista encadeada pode ser de qualquer tipo. Note ainda que a forma de estruturação hierárquica de dados apresentada acima facilita a modificação do tipo de informação contida em cada nó da lista (ou o aproveitamento para a criação de uma nova lista de outro tipo): a única coisa que precisa ser feita, neste caso, é a substituição do tipo `tRegistro` pelo tipo desejado. Também, o identificador `dado` é genérico demais para ser útil em termos de legibilidade. No caso da lista de espera, provavelmente, o nome `passageiro` tornaria o programa mais legível.

4.4.3 Aplicações de Listas Encadeadas

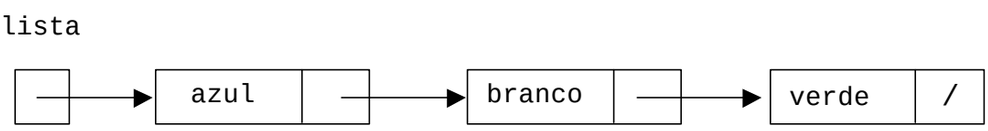
O uso de listas encadeadas tipicamente requer, pelo menos, as seguintes operações:

- **Criação** da lista
- **Inserção** de um elemento na lista
- **Remoção** de um elemento da lista
- **Localização** de (acesso a) um elemento da lista

[9] As duas primeiras declarações foram acrescentas aqui apenas para facilidade de referência. Concentre-se na última declaração que é a que, efetivamente, é utilizada na criação da lista encadeada.

Estas operações serão ilustradas a seguir com a utilização de uma lista encadeada cujos campos `dado` de seus nós contêm *strings* com nomes de cores[10].

Um exemplo de uma lista encadeada com três nós, onde cada nó contém o nome de uma cor, é apresentado a seguir:



Na lista ilustrada no diagrama acima, cada nó possui dois campos: o primeiro campo é um *string* e representa o verdadeiro conteúdo (i.e., a informação) contido no nó; o segundo campo, representado por setas é um ponteiro para o próximo elemento na lista. O símbolo `/` no segundo campo do último nó indica que aquele nó não aponta para nenhum outro nó. A variável `lista` é um ponteiro para o primeiro elemento (nó) da lista. Cada elemento da lista pode ser acessado começando-se pelo início da lista e seguindo-se os ponteiros para os nós seguintes.

Como ilustração do uso de ponteiros e alocação dinâmica de memória, será examinada a implementação de listas encadeadas como estruturas dinâmicas[11]. O primeiro passo para a criação de uma lista encadeada é a definição do tipo de cada nó da lista. Lembre-se que para assegurar o acesso a todos os elementos da estrutura, é necessário que cada elemento inclua uma indicação de onde o próximo elemento localiza-se em memória. Assim, cada elemento deve consistir de duas partes: (1) a informação que deve ser armazenada no elemento, e (2) o endereço de memória do próximo elemento. Portanto, para implementação da lista encadeada ilustrada no diagrama anterior, pode-se utilizar a seguinte declaração de tipos:

```
typedef  struct  no {
            char        dado[10];
            struct  no  *proximo;
} tNo, *tApontador;
```

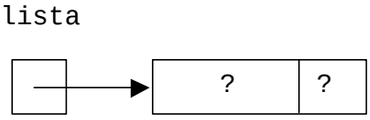
Para criar a lista, são necessárias três variáveis do tipo ponteiro `tApontador`: o primeiro ponteiro representará exatamente o início da lista, enquanto que o segundo e o terceiro ponteiros facilitarão a inserção de novos nós na lista como será visto em seguida. Estes ponteiros são declarados como:

```
tApontador      lista, ponteiroAux1, ponteiroAux2;
```

Procede-se agora à criação da lista ilustrada no último diagrama. O primeiro nó (v. ilustração acima) pode ser criado por meio das seguintes instruções (as instruções foram numeradas para facilitar a discussão que se segue):

```
lista = malloc(sizeof(tNo));  /* 1 */
strcpy(lista->dado, "azul");  /* 2 */
```

Após a execução da instrução 1, tem-se a seguinte situação (o símbolo `?` significa que o respectivo campo do nó é indeterminado naquele instante):

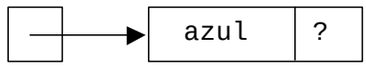


Recorde-se da **Seção 3.6** que a função de biblioteca **strcpy**() copia o conteúdo de seu segundo argumento (*string*) no primeiro argumento (outro *string*). Assim, após a execução da instrução 2, a situação passa a ser a seguinte:

`lista`

[10] Provavelmente, esta lista é de pouca utilidade prática, mas serve bem para o propósito didático aqui proposto.

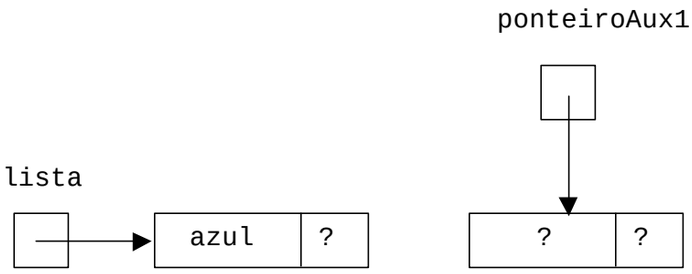
[11] Listas encadeadas podem também ser implementadas por meio de arranjos, mas isso não é de interesse aqui.



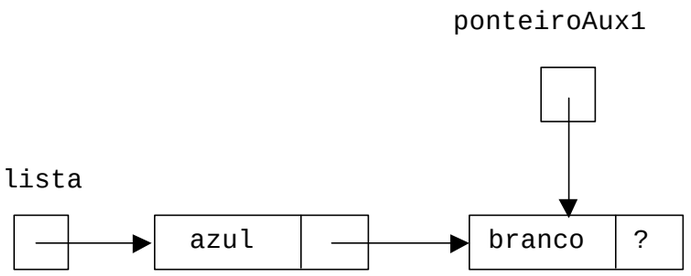
O segundo nó da lista é criado e acrescentado à lista através das seguintes instruções:

```
ponteiroAux1 = malloc(sizeof(tNo));       /* 3 */
strcpy(ponteiroAux1->dado, "branco");     /* 4 */
lista->proximo = ponteiroAux1;            /* 5 */
```

Após a execução da instrução 3 tem-se a seguinte situação (note que, nesse instante, o novo nó ainda não está encadeado na lista):



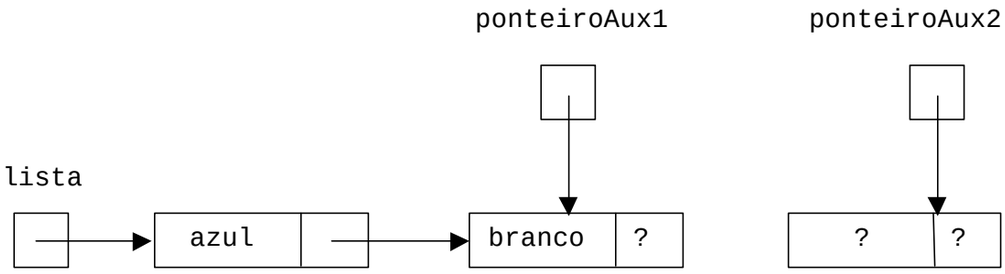
A instrução 4 preenche o valor do primeiro campo do novo nó, enquanto que a instrução 5 faz a devida anexação deste nó à lista. Assim, após a execução da instrução 5 a situação é seguinte:



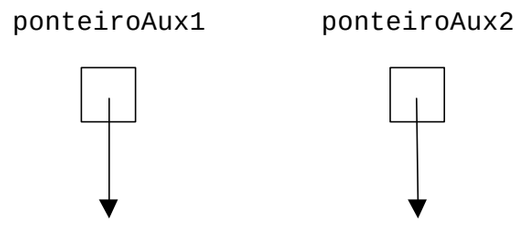
O último conjunto de instruções responsável pela inserção do último elemento da lista é:

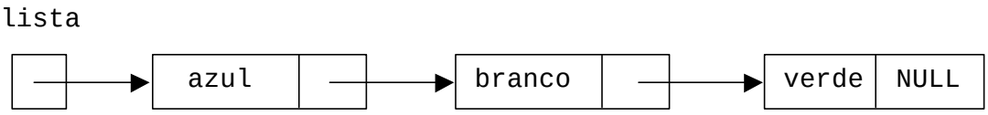
```
ponteiroAux2 = malloc(sizeof(tNo));             /* 6 */
strcpy(ponteiroAux2->dado, "verde");            /* 7 */
ponteiroAux1->proximo = ponteiroAux2;           /* 8 */
ponteiroAux2->proximo = NULL;                   /* 9 */
```

Após a execução da instrução 6 tem-se a seguinte situação (note, novamente, que, nesse ponto, o novo nó ainda não está encadeado na lista):



A instrução 7 preenche o campo de dado do nó apontado por `ponteiroAux2`, enquanto que a instrução 8 faz a ligação do nó apontado por `ponteiroAux1` para o nó apontado por `ponteiroAux2`. A última instrução (9) simplesmente indica que o último elemento não aponta para nenhum outro nó. A situação final da lista é ilustrada a seguir:





Pode parecer que foram utilizados três ponteiros (`lista`, `ponteiroAux1` e `ponteiroAux2`) devido ao fato de a lista possuir três elementos, mas isto não corresponde à realidade. Isto é, se quiséssemos acrescentar um quarto elemento à lista, não necessitaríamos de um quarto ponteiro auxiliar. Neste caso, o que teríamos que fazer seria o seguinte: (1) alocar o novo nó utilizando o ponteiro `ponteiroAux1`; (2) atribuir os devidos valores aos campos do novo nó (ainda através do ponteiro `ponteiroAux1`); (3) fazer a ligação do nó apontado pelo ponteiro `ponteiroAux2` para o nó apontado por `ponteiroAux1`.

Exercício: Suponha que o dado contido no novo nó seja o *string* `vermelho`. Escreva um conjunto de instruções para acrescentar um quarto nó na lista acima.

Existe algo importante ausente no conjunto de instruções acima. Como foi visto na **Seção 4.2**, deve-se ***sempre*** testar se uma dada alocação de memória foi bem sucedida antes de utilizar o espaço que se espera ter sido alocado. Isto não foi incluído no último exemplo apenas para não desviar a atenção dos pontos mais importantes de criação da lista.

Exercício: Inclua um teste após cada chamada da função **malloc()** que verifique se cada pedido de alocação de memória foi atendido.

Suponha, agora, que você deseja imprimir os valores contidos nos campos `dado` de cada nó da lista encadeada acima. Para fazer isso, você precisaria percorrer (ou **atravessar**) toda a lista do início ao fim. A função a seguir realiza isto:

```
/****
 * ImprimeListaEncadeada()
 *
 * Imprime os nos de uma lista simplesmente encadeada apontada por p.
 ****/
void  ImprimeListaEncadeada(tApontador  p)
{
     while  (p)  {
        printf("%s\n", p->dado);    /* Imprime o valor do nó */
        p = p->proximo;             /* Avança para o próximo nó */
     }  /* while */
}   /* ImprimeListaEncadeada */
```

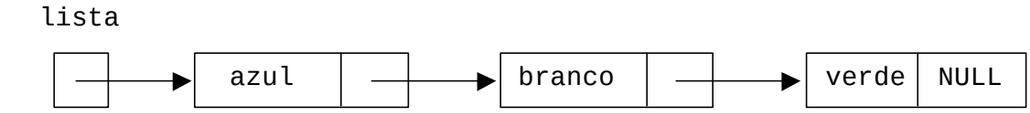
Uma chamada `ImprimeListaEncadeada(lista)` imprimiria todos os valores armazenados nos campos `dado` dos nós da lista apontada por `lista`. O funcionamento da função `ImprimeListaEncadeada()` é simples. Ela recebe como entrada um ponteiro para (o início de) uma lista encadeada. Esta lista pode estar vazia, ou conter um ou mais elementos. No primeiro caso, o ponteiro, representado pelo parâmetro formal `p`, é **`NULL`** e o laço **while** não é executado. No segundo caso, entra-se no laço **while**, imprime-se o valor do campo `dado` do primeiro elemento, e faz-se `p` apontar para o próximo nó. Se este último tiver o valor **`NULL`**, a iteração pára; caso contrário, repete-se o processo até que `p` assuma eventualmente o valor **`NULL`**[12].

Elementos (nós) podem ser inseridos numa lista encadeada de várias maneiras. Pode-se, por exemplo, requerer que um nó seja inserido no início ou no final da lista; ou pode-se desejar

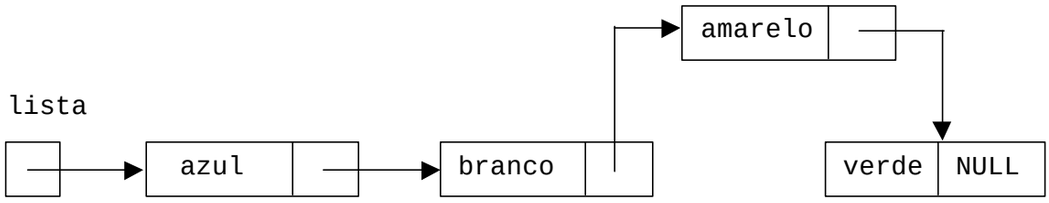
[12] **Cuidado**: Não caia na tentação de substituir a instrução `p = p->proximo` por `p++` ou `++p`. Lembre-se que os elementos de uma lista encadeada não são necessariamente contíguos.

que a inserção seja feita imediatamente antes ou imediatamente depois de um nó contendo um determinado valor. Como exemplo, a função apresentada em seguida pode ser utilizada para inserir um nó imediatamente depois de um nó contendo um determinado valor numa lista cujos nós são do tipo do exemplo anterior.

Como ilustração do processo, suponha, por exemplo, que se deseje inserir um nó cujo valor do campo `dado` seja `amarelo` imediatamente depois do nó contendo `branco` na lista abaixo:



A situação final deveria então ser a seguinte:



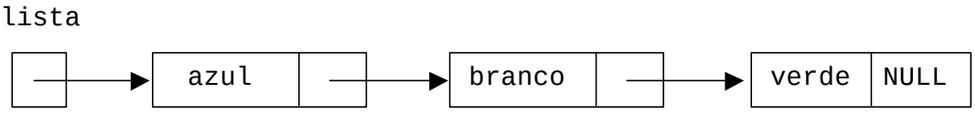
As mudanças feitas na lista para a inserção do nó `amarelo` são simples: o nó `branco` passa a apontar para o novo nó (`amarelo`), que por sua vez passa a apontar para o nó `verde`. Note, entretanto, que não se podem fazer as novas atribuições de endereços nesta ordem. Isto é, se o endereço do nó `amarelo` for atribuído ao campo `proximo` do nó `branco` antes da atribuição do endereço do nó `verde` ao campo `proximo` do nó `amarelo`, o nó `verde` estará perdido para sempre[13]. A função `InsereDepois()`, apresentada a seguir, implementa essas idéias.

```
/****
 *
 * InsereDepois()
 *
 * Insere um novo nó cujo conteúdo é stringNovo após o nó cujo conteúdo é
 * stringNaLista na lista apontada por p.
 *
 ****/
void   InsereDepois(tApontador  p, char  *stringNovo, char  *stringNaLista)
{
    tApontador   posicaoDoNo, noNovo;

    posicaoDoNo = NULL;
    while  (p  &&  !posicaoDoNo){       /* Procura o nó */
       if   (!strcmp(p->dado, stringNaLista))
         posicaoDoNo = p;        /* Nó procurado foi encontrado */
       else
         p = p->proximo;         /* Passa para o próximo nó */
    }

    if  (posicaoDoNo) { /* posicaoDoNo foi encontrado e aponta para o nó */
                        /* após o qual a inserção será feita.           */
        noNovo = malloc(sizeof(tNo)); /* Aloca o novo nó */
        if (noNovo) {
          strcpy(noNovo->dado, stringNovo); /* Preenche conteúdo do novo nó */
          noNovo->proximo = posicaoDoNo->proximo; /* Faz o novo nó apontar  */
                                                  /* para o nó apontado    */
                                                  /* antes pelo nó          */
                                                  /* encontrado             */
          posicaoDoNo->proximo = noNovo;  /* Faz o nó encontrado apontar  */
        }  /* if */                /* para o nó  novo              */
    } /* if */
}  /* InsereDepois */
```

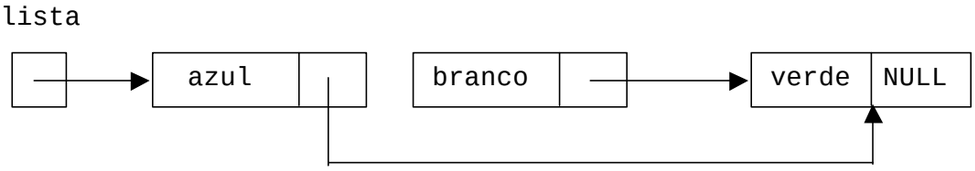
[13] No diagrama, isto significa que se você desenhar primeiro a seta do nó `branco` para o nó `amarelo`, você não será nunca capaz de desenhar a seta do nó `amarelo` para o nó `verde`, simplesmente porque não sabe aonde o nó `verde` se encontra. Convença-se de que realmente entendeu isso antes de prosseguir.

Exercício: Transforme a função acima numa função que retorna `1` se a inserção foi bem sucedida (i.e., se o nó foi realmente inserido na lista) e `0` em caso contrário.

Como último exemplo, a função apresentada a seguir remove um nó, cujo conteúdo é especificado, de uma lista do tipo apresentado acima. Como ilustração, suponha que se deseje remover o nó cujo valor do campo `dado` seja `branco` na lista a seguir:



A situação final deveria então ser a seguinte:



Conforme pode ser observado nesta última ilustração, a remoção do nó desejado envolve apenas o desvio do endereço do campo `proximo` do nó anterior àquele sendo removido para o nó que era anteriormente apontado pelo nó a ser removido. Na prática, deve-se também liberar o espaço reservado para o nó removido, mas em princípio, o desvio representado pelo esquema acima é suficiente para a remoção do nó da lista encadeada. A função a seguir produz o efeito de remoção desejado:

```
/****
 * RemoveNo()
 *
 * Remove o primeiro nó encontrado cujo conteúdo é stringNaLista na lista
 * apontada por p.
 ****/

void   RemoveNo(tApontador   *p, char   *stringNaLista)
{
    tApontador noAnterior = NULL, /* No início da lista não há nó anterior  */
               posicaoDoNo= *p; /* Começa busca a partir do início da lista */
    unsigned char noFoiEncontrado = FALSE; /* Nó não foi ainda encontrado  */

    while  (posicaoDoNo  &&  !noFoiEncontrado)  {  /* Procura o nó */
        if   (!strcmp(posicaoDoNo->dado, stringNaLista))
          noFoiEncontrado = TRUE; /* Nó procurado foi encontrado */
        else {                          /* O nó anterior passa a ser     */
          noAnterior = posicaoDoNo;     /* o atual e o atual passa a ser */
          posicaoDoNo = posicaoDoNo->proximo; /*  o próximo        */
        }    /* if */
    }

    if  (noFoiEncontrado) {
        if  (!noAnterior)   /* O nó a ser removido é o primeiro da lista */
          *p = (*p)->proximo;   /* O primeiro nó será o antigo segundo nó */
        else         /* O nó a ser removido NÃO é o primeiro da lista */
          noAnterior->proximo = posicaoDoNo->proximo;/* Faz desvio desejado */

    /* Neste ponto, o nó apontado por posicaoDoNo não mais pertence à */
    /* lista e a remoção está logicamente concluída. Resta apenas     */
    /* liberar o espaço ocupado pelo nó removido                       */

       free(posicaoDoNo);    /* Libera a alocação de memória do nó removido */
    }
}
```

Note que a remoção do primeiro nó da lista é um caso que deve ser tratado à parte pela função `RemoveNo()`. Note ainda que se houver na lista mais de um nó com o mesmo conteúdo especificado para remoção, apenas o primeiro nó com este conteúdo será removido com uma única chamada da função acima. Finalmente, observe que o primeiro argumento desta última função não é do tipo `tApontador` como na função `InsereDepois()`, mas um ponteiro para este tipo (por que?).

Exercício: Modifique a função `RemoveNo()` acima de modo que ela seja capaz de remover (com uma única chamada) todas as ocorrências de nós com o conteúdo especificado, e não apenas o primeiro nó encontrado.

4.5 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***. Portanto, uniões são utilizadas primariamente com o objetivo de economizar memória.

Uniões obedecem regras sintáticas semelhantes àquelas das estruturas. Portanto, para declarar-se uma união, pode-se utilizar qualquer um dos formatos apresentados anteriormente 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:

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

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

Uniões podem ser inicializadas através da atribuição de um valor inicial ao primeiro componente da variável do tipo união. 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**. Este tipo de estrutura, é composta de, pelo menos, uma parte fixa e, pelo menos, uma parte variante. Considere, por exemplo, as seguintes definições:

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

typedef  struct {
            char   rua[30];
            char   numero[5];
            char   cidade[20];
            char   uf[3];
```

```
            char   cep[10];
         } tEndereco;

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

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

A variável `empregado` é um registro variante, cuja parte fixa consiste dos campos `nome`, `endereco` e `estadoCivil`, enquanto que a parte variante consiste dos campos da união (`nomeDoConjuge`, `moraSozinho` e `dataDoDivorcio`). Normalmente, em registros variantes, utiliza-se um dos campos fixos como indicador (***flag***) para informar qual é o campo variante da estrutura válido num dado instante. No último exemplo, o campo `estadoCivil` pode servir como indicador. Utilizando um campo indicador, o programador pode acessar corretamente o campo variante, como mostra o seguinte exemplo:

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

Se um registro variante não possui campo indicador, não existe nenhuma forma de se determinar qual campo variante está correntemente em uso. Portanto, o programador deve dispor de um outro meio para certificar-se de não referenciar 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 pelo uso de conversões explícitas de dados (v. **Seção 1.3**).

4.6 Exercícios de Revisão

1. Qual é a principal diferença entre estruturas e arranjos?

2. (a) Descreva os formatos permitidos para a declaração de uma variável de um tipo estrutura. (b) Qual desses formatos é mais apropriado e por que?

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

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

5. Como um arranjo de estruturas é inicializado?

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) Quais são as propriedades de associatividade e de precedência destes operadores?

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

10. Como uma estrutura pode ser retornada por uma função?

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

12. (a) O que são listas encadeadas? (b) Por que listas encadeadas são consideradas estruturas de dados dinâmicas? (c) Qual é a vantagem do uso de listas encadeadas com relação a arranjos?

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

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

15. Suponha que o tipo **int** ocupe 4 bytes e o tipo **float** ocupe 8 byte 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;
```

4.7 Exercícios de Programação

EP4.1) Defina o tipo `tComplexo`, a ser utilizado na representação de números complexos, da seguinte forma:

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

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

(i) Leitura de um número complexo do teclado.
Protótipo: `tComplexo  *LeComplexo( void );`

(ii) Impressão de um número complexo na tela (numa forma legível).
Protótipo: `void  ImprimeComplexo( tComplexo  *umComplexo );`

(iii) Soma de dois números complexos.
Protótipo:
```
tComplexo *SomaComplexos(tComplexo  *umComplexo,
                         tComplexo  *outroComplexo);
```

(iv) Subtração de dois números complexos.
Protótipo:
```
tComplexo  *SubtraiComplexos( tComplexo  *umComplexo,
                              tComplexo  *outroComplexo);
```

(v) Multiplicação de dois números complexos.
Protótipo:
```
tComplexo  *MultiplicaComplexos( tComplexo  *umComplexo,
                                                    tComplexo
*outroComplexo);
```

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

EP4.2) Escreva uma função em C que recebe dois ponteiros para listas encadeadas como entrada e retorne um ponteiro para uma terceira lista que é a concatenação destas duas listas de entrada. As duas listas de entrada devem permanecer imutáveis.

EP4.3) Uma ***pilha*** é um tipo especial de lista na qual acréscimo de elementos, denominado empilhamento, ocorre apenas no final da lista, e retirada de elementos, denominado desempilhamento, também ocorre sempre no final da lista. Em outras palavras o último elemento acrescentado na pilha é sempre o primeiro a ser retirado (por causa disto, a pilha é denominada de estrutura LIFO, do Inglês: *Last In*, *First Out*). Considerando uma implementação de pilhas por meio de listas encadeadas, escreva duas funções em C, denominadas `Empilha()` e `Desempilha()`, que implementem as operações de acréscimo e retirada de elementos de uma pilha cujos elementos são do tipo `tElemento`.

EP4.4) Uma ***fila*** é um outro tipo especial de lista na qual acréscimo de elementos ocorre apenas no final da lista, enquanto que retirada de elementos ocorre sempre no início da lista. Em outras palavras o primeiro elemento acrescentado na fila é sempre o primeiro a ser retirado (por causa disto, a pilha é denominada de estrutura FIFO, do Inglês: *First In*, *First Out*). Considerando uma implementação de filas por meio de listas encadeadas, escreva duas funções em C, denominadas `Acrescenta()` e `Retira()`, que implementem as operações de acréscimo e retirada de elementos de uma fila cujos elementos são do tipo `tElemento`.