Estruturas de Dados e Algoritmos Genéricos

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Ponteiro para função
  - ❒ Ponteiro genérico
  - ❒ Endereço de função
  - ❒ Lista generalizada
  - ❒ Átomo de lista
  - ❒ Sublista
  - ❒ Cabeça de lista
  - ❒ Cauda de lista
  - ❒ Nó de lista generalizada
  - ❒ Notação polonesa
  - ❒ Notações infixa, prefixa e sufixa
  - ❒ Tipo **void ***
- Descrever o funcionamento das seguintes funções da biblioteca padrão de C:
  - ❒ **memcpy()**
  - ❒ **memmove()**
  - ❒ **qsort()**
  - ❒ **bsearch()**
- Explicitar a relação entre nome de função e ponteiro para função em C
- Expor situações nas quais ponteiros para funções são usados na prática
- Explicar por que o uso de recursão é adequado em implementações de operações sobre listas generalizadas
- Descrever estrutura de dados genérica e quando ela se faz necessária
- Implementar uma pilha ou fila capaz de armazenar qualquer tipo de elemento
- Mostrar como deve ser definida uma função de comparação que possa ser usada com **qsort()** ou **bsearch()**
- Descrever as notações com as quais uma expressão aritmética pode ser escrita
- Efetuar transformações entre as formas infixa, prefixa e sufixa de uma expressão
- Descrever como se avalia uma expressão aritmética em forma sufixa usando pilha

objetivos

ESTE CAPÍTULO INICIA discutindo o conceito e a aplicação de ponteiros para funções. Esse importante tópico de programação em C permite a implementação de algoritmos e estruturas de dados genéricos, que constituem o tema central do capítulo.

11.1 Ponteiros para Funções

Ponteiros para funções constituem uma ferramenta bastante poderosa em programação em C. Antes de introduzir esse conceito, entretanto, é necessário que se examine mais profundamente o modo como o compilador de C interpreta uma chamada de função.

Não apenas variáveis são armazenadas na memória do computador durante a execução de um programa. O próprio código executável do programa também é armazenado em memória. Consequentemente, do mesmo modo que cada variável possui um endereço em memória, cada instrução de um programa em linguagem de máquina também possui um endereço. Quando uma função é definida, o compilador de C considera o nome da função como sendo o endereço em memória da primeira instrução da função em linguagem de máquina. Assim uma chamada de função num programa causa a transferência do fluxo de execução do programa para o endereço da função.

Assim como ocorre com variáveis, o endereço de uma função não pode ser modificado num programa. Entretanto, pode-se ter um ponteiro que aponte para várias funções em diferentes pontos de um programa, do mesmo modo que se pode ter um ponteiro capaz de apontar para várias variáveis em instantes diferentes.

11.1.1 Definição de Ponteiro para Função

Para definir um ponteiro para função deve-se preceder o nome dessa variável com asterisco, como na definição de qualquer ponteiro, mas, aqui, deve-se também colocar o asterisco e o ponteiro entre parênteses. A definição termina com um par de parênteses contendo os tipos dos parâmetros das funções para as quais o ponteiro pode apontar. Portanto uma definição de ponteiro para função assume a seguinte forma:

tipo (*nome-do-ponteiro)(tipos-dos-parâmetros);

em que *tipo* denota o tipo de retorno das funções para as quais o ponteiro pode apontar e *tipos-dos-parâmetros* representa a lista dos tipos dos parâmetros dessas funções.

O programador pode optar por não declarar os tipos dos parâmetros, obtendo, assim, um ponteiro que pode apontar para qualquer função que tenha o tipo de retorno especificado. Essa prática, no entanto, não é recomendável e não será usada neste livro.

Como exemplo de definição de um ponteiro para função, considere:

```
int (*pf)(double);
```

que define `pf` como um ponteiro para funções cujo tipo de retorno é **int** e que têm apenas um parâmetro do tipo **double**. Isto significa dizer, por exemplo, que `pf` pode apontar para uma função `F1()` com um parâmetro do tipo **double** cujo tipo de retorno é **int**, mas não pode apontar para uma função `F2()` sem parâmetro nem para uma função `F3()` com um parâmetro do tipo **double**, mas cujo tipo de retorno é **double**.

Note que, se os parênteses em torno de `*pf` na definição apresentada no exemplo anterior forem omitidos, o compilador tratará a definição como se fosse uma alusão a uma suposta função `pf()` cujo parâmetro é do tipo **double** e cujo tipo de retorno é um ponteiro para o tipo **int**.

11.1.2 Atribuição de Valor a um Ponteiro para Função

Como o compilador trata nomes de funções como endereços, atribuir um valor a um ponteiro para função requer atribuir ao ponteiro um nome de função compatível com o ponteiro. Por exemplo:

```
extern int F1(double); /* Alusão da função F1() */

int  (*pf)(double); /* Ponteiro para função que recebe um parâmetro */
                    /* do tipo double e retorna um valor int        */

pf = F1; /* pf passa a apontar para a função F1() */
```

Outras possibilidades de atribuição a `pf` consideradas incorretas são exemplificadas a seguir:

```
pf = F1(2.5);
```

Essa instrução não é ilegal, mas é uma conversão para ponteiro dependente de implementação, pois `F1(2.5)` é uma chamada de função que retorna **int** e, portanto, se está tentando atribuir um valor do tipo **int** a um ponteiro.

```
pf = &F1(2.5);
```

A expressão `&F1(2.5)` é ilegal, pois se está tentando obter o endereço do valor retornado por `F1()`.

```
pf = &F1;
```

Estritamente falando, a expressão `&F1` é incorreta, pois se está tentando obter o endereço de um endereço, mas compiladores ISO não irão considerar isso um erro. Eles apenas ignorarão o operador **&**.

Outros pontos importantes que o programador deve considerar quando atribui um valor a um ponteiro para função são:

[1] Os tipos de retorno na definição da função e na definição do ponteiro para função devem ser os mesmos.

[2] Os tipos dos parâmetros na definição da função e na definição do ponteiro para função devem ser os mesmos.

Por exemplo, se foi declarado um ponteiro para função com parâmetro **double** e que retorna um valor do tipo **int**, deve-se atribuir a esse ponteiro o endereço de uma função cujos tipos de parâmetros e de retorno sejam respectivamente iguais a esses tipos. Por exemplo, dadas as seguintes declarações:

```
extern double F2(double);
extern int    F3(void);
int           (*pf)(double);
```

os resultados das seguintes atribuições seriam indefinidos:

```
pf = F2;
```

`pf` e `F2` são incompatíveis, pois seus tipos de retorno são diferentes.

```
pf = F3;
```

`pf` e `F3` são incompatíveis, pois seus tipos de parâmetros são diferentes.

Apesar das incompatibilidades apontadas nas atribuições acima, elas são legais; i.e., elas compilam num compilador padrão de C. Em tais casos, o padrão ISO de C apenas requer que o compilador emita uma mensagem de advertência.

Conversões entre ponteiros para funções que têm tipos de parâmetros diferentes efetuadas com o uso de conversão explícita evitam que o compilador emita mensagens de advertência. Mas, o uso de tal conversão para chamar uma função tem comportamento indefinido. Considere os seguintes exemplos:

```
extern int F1(void);
int (*fptr1) (void) = F1; /* OK */
extern int F2(int);
int (*fptr2) (int) = F2; /* OK */
fptr1 = (int (*) (void))F2; /* OK */
(*fptr1)(); /* Resultado indefinido, pois F2() deve receber um parâmetro int */
```

Um ponteiro para função não deve ser convertido num ponteiro para um tipo de dado, ou vice-versa, pois o resultado é indefinido. Por exemplo:

```
extern int F1(void);
int *p = F1; /* Compila, mas o resultado é indefinido */
```

11.1.3 Chamada de Função Mediante Ponteiro

Chamar uma função por meio de um ponteiro ao qual se atribuiu o endereço dela é similar a chamá-la utilizando seu próprio nome. Por exemplo:

```
extern  int  F1(char c);
int          (*pf)(char);
int          resultado;
char         a;
pf = F1;
resultado = pf(a); /* Chama a função F1() por meio do ponteiro */
                   /* pf, passando a como  parâmetro           */
```

Outra notação que pode ser utilizada para chamar uma função utilizando um ponteiro para função consiste em preceder o ponteiro com * e envolver ambos, o ponteiro e o asterisco, com parênteses. Utilizando essa sintaxe, a chamada de função do último exemplo poderia ser escrita como:

```
resultado = (*pf)(a);
```

Observe que, se, no último exemplo, o par de parênteses em torno de `*pf` fosse omitido o lado direito da atribuição seria interpretado como:

```
*(pf(a))
```

pois uma chamada de função tem precedência maior do que a do operador de indireção. Essa última expressão é obviamente ilegal, tendo em vista a definição de `pf`.

11.1.4 Retorno de Ponteiro para Função

Uma função pode retornar um ponteiro para função. No exemplo a seguir, a função `EscolheFuncao()` é definida com o parâmetro `condicao` do tipo **int** e retorna um ponteiro para uma função que recebe um parâmetro do tipo **double** e cujo tipo de retorno é **int**.

```
extern int  F1(double);
extern int  F2(double);
```

```
int (*EscolheFuncao(int condicao))(double)
{
   if (condicao)
      return  F1;

   return  F2;
}
```

A função `EscolheFuncao()` do exemplo acima, retorna um ponteiro para a função `F1()` se o parâmetro `condicao` é diferente de zero ou um ponteiro para a função `F2()` em caso contrário (lembre-se que `F1` e `F2` são endereços de funções e, portanto, são compatíveis com ponteiros para funções). A função `EscolheFuncao()` poderia ser chamada conforme ilustrado no fragmento de programa a seguir:

```
int  (*pf)(double);
...
pf = EscolheFuncao(2);
```

O uso de definições de tipo facilita a definição de funções que retornam ponteiros para funções e favorece a legibilidade. Por exemplo, usando uma definição de tipo, a função `EscolheFuncao()` poderia ser redefinida como:

```
typedef int (*tFuncPtr) (double);

tFuncPtr EscolheFuncao2(int condicao)
{
   if (condicao)
      return  F1;

   return  F2;
}
```

Claramente, esse conjunto de declarações é mais inteligível do que a definição anterior da função `EscolheFuncao()`.

11.1.5 Ponteiro para Função como Parâmetro de Função

O uso mais comum de ponteiros para funções é como parâmetros de funções. Uma situação típica é o uso de ponteiros para funções como parâmetros de funções que implementam algoritmos de ordenação, como será visto na **Seção 11.5**.

Como parâmetro formal na definição de uma função, um ponteiro para função deve ser declarado exatamente do mesmo modo apresentado na **Seção 11.1.1**. Por exemplo:

```
void FuncaoComPonteiroParaFuncao(double f, int (*pf)(double))
{
   /* Corpo da função FuncaoComPonteiroParaFuncao() */
}
```

Na chamada de uma função que tem um ponteiro para função como um de seus parâmetros formais, deve-se utilizar como parâmetro real correspondente a esse parâmetro apenas o nome de uma função compatível com ele. Por exemplo, suponha que se tenham as seguintes definições de funções no mesmo programa da função `FuncaoComPonteiroParaFuncao()` do exemplo acima:

```
double   x;

int  F1(double umDouble)
{
       /* Corpo da função F1() */
}
```

```
int  F2(double umDouble)
{
       /* Corpo da função F2() */
}
void  F3(double umDouble)
{
       /* Corpo da função F3() */
}
```

Então, as seguintes chamadas seriam perfeitamente legais:

```
FuncaoComPonteiroParaFuncao(x, F1);
FuncaoComPonteiroParaFuncao(x, F2);
```

Entretanto, a chamada:

```
FuncaoComPonteiroParaFuncao(x, F3);
```

produziria um resultado indefinido, pois a função `F3()` não é compatível com o segundo parâmetro da função `FuncaoComPonteiroParaFuncao()`.

11.2 Listas Generalizadas

11.2.1 Conceitos

Uma **lista generalizada** *A* é uma sequência finita de $n \geq 0$ elementos *a1*, *a2*, ..., *an*, em que cada *ai* pode ser um **átomo** ou uma lista generalizada. Os elementos *ai* que não são atômicos são chamados de **sublistas** de *A*. Deve-se observar que essa definição é recursiva, pois define-se lista generalizada em termos de si própria.

Aqui, será adotada a convenção de uso de letras maiúsculas para denotar listas (inclusive sublistas) e letras minúsculas para denotar átomos. A **Tabela 11–1** apresenta exemplos de listas generalizadas e ilustra os conceitos apresentados.

Lista	Interpretação
A = ()	**Lista vazia (ou nula), seu comprimento é *0***
B = (a, (b, c))	**Lista de comprimento dois; o primeiro elemento da lista é o átomo *a* e o segundo elemento da lista é a lista *(b, c)***
C = (B, B, ())	**Lista de comprimento 3, cujos dois primeiros elementos são as listas *B* e o terceiro elemento é a lista vazia**
D = (a, D)	**Lista recursiva de comprimento 2. *D* consiste da lista *(a, (a, (a, ...)***

Tabela 11–1: Exemplos de Listas Generalizadas

Dada uma lista $A = (a_1, a_2, ..., a_n)$, com $n \geq 1$, o elemento a_1 é denominado a **cabeça** de *A*, enquanto a lista $(a_2, ..., a_n)$ é denominada a **cauda** de *A*. Se uma lista possui um único elemento, então sua cauda é a **lista vazia**. Note que a cauda de uma lista é sempre uma lista. Por exemplo, considerando-se a lista *A = (a, (b, c))* do exemplo acima, tem-se que:

cabeça(A) = a

cauda(A) = ((b,c))

cabeça(cauda(A)) = (b,c)

cabeça(cabeça(cauda(A))) = b
cauda(cabeça(cauda(A))) = (c)
cauda(cauda(A)) = ()

11.2.2 Implementação

Uma forma de representação de listas generalizadas é por meio da utilização de nós da forma mostrada na **Figura 11–1**.

FIGURA 11–1: NÓ DE LISTA GENERALIZADA

Na **Figura 11–1**, o campo `indicador` serve para informar se o campo `conteudo` contém um átomo ou um ponteiro para uma sublista. O campo `proximo` é utilizado como ponteiro para a cauda da lista, enquanto o campo `conteudo` pode conter um átomo, no caso de a cabeça de uma lista *A* ser um átomo, ou apontar para uma lista, no caso em que a cabeça de *A* é uma lista. Com essa representação, a lista *A = (a, (b, c))* seria apresentada esquematicamente como na **Figura 11–2**.

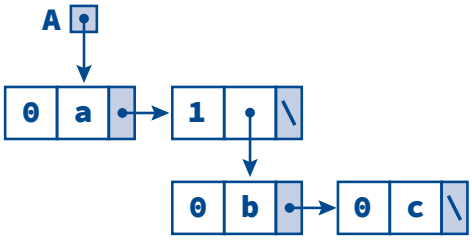

FIGURA 11–2: REPRESENTAÇÃO ESQUEMÁTICA DE UMA LISTA GENERALIZADA

Esta discussão sugere que listas generalizadas sejam implementadas em C por meio de registros variantes (v. **Seção 3.10**) e ponteiros. Uma possível implementação seria obtida por meio das definições de tipos apresentada abaixo, na qual os átomos são do tipo **char**.

```
typedef enum {ATOMO, LISTA} tAtomoOuLista;

typedef union {
           char                atomo;
           struct rotNoListaG *lista;
        } tConteudoListaG;

typedef struct rotNoListaG {
           tAtomoOuLista       indicador;
           tConteudoListaG     conteudo;
           struct rotNoListaG *proximo;
        } tNoListaG, *tListaG;
```

O trecho de programa a seguir demonstra como a lista *A = (a, (b, c))* pode ser criada:

```
   tListaG lista, ptr1, ptr2, lista2;

   /* Construção da lista: A = (a,(b,c)) */

   lista = malloc(sizeof(tNoListaG)); /* Cria o primeiro nó */
   lista->indicador = ATOMO;   /* Este nó contém um átomo */
   lista->conteudo.atomo = 'a';

   ptr1 = malloc(sizeof(tNoListaG)); /* Cria o segundo nó */
   lista->proximo = ptr1; /* Faz primeiro nó apontar para o segundo */
   ptr1->indicador = LISTA; /* Este nó aponta para uma sublista */
```

```
   ptr1->proximo = NULL; /* Este nó é o último da lista */

   /*                                    */
   /* Criação da sublista do segundo nó */
   /*                                    */

   ptr2 = malloc(sizeof(tNoListaG)); /* Cria o primeiro nó da sublista */
      /* Este nó contém um átomo */
   ptr2->indicador = ATOMO;
   ptr2->conteudo.atomo = 'b';

      /* Faz segundo nó da lista apontar para o primeiro nó da sublista */
   ptr1->conteudo.lista = ptr2;

   ptr1 = malloc(sizeof(tNoListaG)); /* Cria o segundo nó da sublista */
      /* Faz o primeiro nó da sublista apontar para o segundo */
   ptr2->proximo = ptr1;

   ptr1->indicador = ATOMO; /* Este nó contém um átomo */
   ptr1->conteudo.atomo = 'c';
   ptr1->proximo = NULL; /* Este nó é o último da sublista */
```

Exercício: Para cada instrução do último exemplo, desenhe uma parte do diagrama que representa esquematicamente a lista *A = (a, (b, c))*. Ao final do conjunto de instruções, você deverá obter uma representação semelhante àquela apresentada na **Figura 11–2**.

11.2.3 Funções Recursivas para Listas Generalizadas

Quando uma estrutura de dados é definida recursivamente, como no caso de listas generalizadas, é usualmente fácil escrever algoritmos recursivos para processá-la. As funções que serão apresentadas a seguir exemplificam a utilização de recursividade na manipulação de listas generalizadas de uma forma bem natural.

Clonagem

A função `CopiaListaG()` produz a cópia de uma lista ou, mais precisamente, ela retorna um ponteiro para uma lista que é uma duplicata daquela recebida como parâmetro.

```
tListaG CopiaListaG(tListaG L)
{
   tListaG ptr = NULL;

   if (L) {
      ptr = malloc(sizeof(tNoListaG));
      if (L->indicador == ATOMO)
         ptr->conteudo.atomo = L->conteudo.atomo;
      else
         ptr->conteudo.lista = CopiaListaG(L->conteudo.lista);

      ptr->proximo = CopiaListaG(L->proximo);
      ptr->indicador = L->indicador;
   }

   return ptr;
}
```

Igualdade

A função `SaoIguaisListasG()` determina se duas listas são iguais; i.e., ela retorna `1` se as listas recebidas como parâmetros são iguais ou `0`, em caso contrário.

```
int SaoIguaisListasG(tListaG  L1, tListaG  L2)
{
   int resultado = 0;

   if (!L1 && !L2)
      resultado = 1;
   else if (L1 && L2)
      if (L1->indicador == L2->indicador) {
         if (L1->indicador == ATOMO)
            resultado = (L1->conteudo.atomo == L2->conteudo.atomo);
         else
            resultado = SaoIguaisListasG(L1->conteudo.lista, L2->conteudo.lista);

         if (resultado)
            resultado = SaoIguaisListasG(L1->proximo, L2->proximo);
      }

   return resultado;
}
```

Profundidade

A profundidade de uma lista generalizada é definida como:

$$Prof(L) = \begin{cases} 0, \text{ se } L \text{ é um átomo} \\ \\ 1 + max\{Prof(a_1), Prof(a_2), \ldots, Prof(a_n)\}, \\ \quad \text{se } L \text{ é a lista } (a_1, a_2, \ldots, a_n) \end{cases}$$

Note que, neste contexto, a lista vazia é considerada um átomo e, portanto, tem profundidade igual a zero. A função `ProfListaG()`, apresentada a seguir, calcula a profundidade de uma lista, conforme a definição apresentada acima.

```
int ProfListaG(tListaG  L)
{
   int     max, resultado;
   tListaG ptr;

   if (!L)
      return 0; /* A lista vazia tem profundidade igual a zero */

   max = 0;

   ptr = L;

      /* Calcula a profundidade de cada nó */
   while (ptr) {
         /* Profundidade do nó corrente */
      if (ptr->indicador == ATOMO)
         resultado = 0;
      else                        /* Idem */
         resultado = ProfListaG(ptr->conteudo.lista);

      if (resultado > max)
         max = resultado; /* Atualiza o maior valor encontrado até aqui */

      ptr = ptr->proximo;
   }

   return max + 1;
}
```

Exibição

A função `ExibeListaG()`, apresentada a seguir, exibe na tela uma lista generalizada num formato semelhante àquele que vem sendo utilizado. Por exemplo, a lista *(a, (b, c))* é exibida como `(a(bc))`.

```
void  ExibeListaG(tListaG  p)
{
   if (!p) {
      printf("()");
      return;
   }

   putchar('(');

   while (p)  {
      if (p->indicador == ATOMO)
         putchar(p->conteudo.atomo);
      else
         ExibeListaG(p->conteudo.lista);

      p = p->proximo;
   }

   putchar(')');
}
```

11.3 Estruturas de Dados Genéricas

As descrições conceituais de estruturas de dados discutidas até aqui não especificam os tipos de dados que devem ser manipulados das maneiras prescritas pelas respectivas operações. Por exemplo, na descrição conceitual da estrutura de dados pilha, as operações de empilhamento e desempilhamento não especificam quais tipos de dados devem ser empilhados ou desempilhados. Ou seja, essas operações são as mesmas, quer o tipo de dado em questão seja inteiro, string ou qualquer outro. Entretanto, em cada exemplo de implementação de estrutura de dados apresentado até aqui, foi necessário especificar um tipo de dado a ser manipulado de certa maneira pela estrutura.

Considere como exemplo a implementação de pilha apresentada na **Seção 8.1**. Essa implementação utiliza **char** como tipo dos valores empilhados e desempilhados. Então, se um programa precisar utilizar uma pilha que manipule elementos de um tipo diferente de **char**, a única alteração que precisa ser efetuada na referida implementação é a definição do tipo `tItemPilha`, que representa o tipo dos elementos armazenados numa pilha.

Agora, suponha que um programa precise utilizar pilhas que manipulem dois ou mais tipos de conteúdos diferentes. Nesse caso, utilizando a abordagem de especificação prévia dos dados a serem processados considerada até aqui, seria necessário um módulo separado para cada tipo de conteúdo utilizado pelo programa. Além disso, as respectivas funções que representam as operações sobre pilhas apresentariam diferenças mínimas em suas implementações. Para tornar a argumentação mais clara, suponha que o programa em questão use pilhas de elementos dos tipos **int** e **double**. Então, as funções de empilhamento dos módulos em questão seriam implementadas como mostrado a seguir, em que os tipos `tPilhaInt` e `tPilhaDouble` são obtidos substituindo-se o tipo `tItemPilha` da **Seção 8.1** por **int** e **double**, respectivamente.

```
void EmpilhaInt(int item, tPilhaInt  *p)
{
   ASSEGURA(!PilhaCheiaInt(*p), "Erro: Pilha cheia.");
   p->itens[++p->topo] = item;
}
```

e

```
void EmpilhaDouble(double item, tPilhaDouble *p)
{
   ASSEGURA(!PilhaCheiaDouble(*p), "Erro: Pilha cheia.");
   p->itens[++p->topo] = item;
}
```

Observe que as diferenças entre as funções `EmpilhaInt()` e `EmpilhaDouble()` são seus cabeçalhos e as funções chamadas para verificar se uma pilha está cheia. Nas demais respectivas funções de cada um dos módulos do exemplo em questão, as diferenças são mínimas.

Num caso como esse do último exemplo, o ideal seria ter uma estrutura de dados que fosse capaz de manipular dados de qualquer tipo. Uma estrutura de dados dessa natureza é denominada **estrutura de dados genérica** e é importante notar que essa denominação diz respeito apenas à implementação de estruturas de dados e não a abstrações.

11.4 Implementação de Pilha Genérica

Em linguagens orientadas a objetos é relativamente fácil implementar estruturas de dados genéricas. Por exemplo, a linguagem C++ provê uma facilidade, denominada *template*, que pode ser usada para tal finalidade. Para implementar tais estruturas em C, faz-se necessário o uso de ponteiros genéricos, funções de alocação dinâmica de memória e da função **memcpy()** da biblioteca padrão (v. adiante).

A seguir, será apresentado um exemplo de implementação de pilha genérica em C. Outras estruturas de dados genéricas (p. ex., fila genérica) podem ser implementadas utilizando-se abordagens semelhantes àquela apresentada no exemplo de pilha genérica visto nesta seção.

11.4.1 Definição de Tipo

Nesta implementação, o contêiner que armazena os elementos da pilha é implementado como um array dinâmico, de modo que o tipo de pilha genérica é definido como:

```
typedef struct {
           void *elementos; /* Ponteiro para o contêiner de elementos da pilha */
           int   tamElemento; /* Tamanho de cada elemento */
           int   nElementos; /* Número de elementos na pilha */
           int   capacidade; /* Tamanho corrente do contêiner */
       } tPilhaGen;
```

11.4.2 Criação

A função `CriaPilhaG()`, apresentada a seguir, aloca um espaço inicial para o array que armazenará os elementos da pilha e inicia os demais campos da estrutura que representa uma pilha.

```
void CriaPilhaG(tPilhaGen *p, int tamElemento)
{
      /* O tamanho de um elemento deve ser positivo */
   ASSEGURA( tamElemento > 0, "Tamanho de elemento deve ser positivo" );

      /* Inicia o campo que indica o tamanho de cada elemento da pilha */
   p->tamElemento = tamElemento;

   p->nElementos = 0; /* Inicialmente, a pilha não tem elementos */

      /* Inicia o campo que indica a capacidade da pilha */
   p->capacidade = TAM_INCREMENTO_PILHA;

      /* Aloca o espaço inicial do contêiner da pilha */
   p->elementos = malloc(TAM_INCREMENTO_PILHA*tamElemento);
```

```
      /* Se ocorreu falha de alocação, aborta o programa */
   ASSEGURA(p->elementos, "Erro de alocacao");
}
```

A função `CriaPilhaG()` utiliza a constante simbólica `TAM_INCREMENTO_PILHA` que representa o tamanho inicial e o valor de acréscimo do contêiner que armazenará os elementos da pilha, quando esse acréscimo se fizer necessário (i.e., quando o contêiner estiver completo).

11.4.3 Destruição

Na implementação de pilha genérica apresentada como exemplo, ocorre alocação dinâmica de espaço sempre que ele é necessário, mas nunca ocorre liberação quando ele deixa de ser necessário. Portanto deve-se implementar uma função que libere o espaço alocado dinamicamente por uma pilha quando ela deixa de ser utilizada por um programa-cliente. É isso que faz a função `DestroiPilhaG()` apresentada a seguir.

```
void DestroiPilhaG(tPilhaGen *p)
{
      /* Libera o espaço ocupado pelo array que armazena os elementos da pilha */
   free(p->elementos);

      /* p->elementos deixou de ser um ponteiro válido */
   p->elementos = NULL;
}
```

11.4.4 Verificação de Pilha Vazia

Uma pilha está vazia quando seu número de elementos, representado pelo campo `nElementos`, for igual a zero. A função `PilhaVaziaG()` verifica se uma pilha genérica está vazia:

```
int PilhaVaziaG(const tPilhaGen *p)
{
      /* A pilha está vazia quando seu número de elementos é igual a zero */
   return p->nElementos == 0;
}
```

11.4.5 Empilhamento

A função `EmpilhaG()` implementa a operação de empilhamento de um novo item numa pilha genérica. Comentários adicionais serão apresentados logo em seguida à apresentação desta função.

```
void EmpilhaG(tPilhaGen *p, const void *pItem)
{
   char *pTopo; /* Apontará para o local do empilhamento */

      /* Se a pilha atingiu seu limite de capacidade, */
      /* tenta aumentar o tamanho de seu contêiner    */
   if (p->nElementos == p->capacidade) {
         /* Calcula a nova capacidade */
      p->capacidade = p->capacidade + TAM_INCREMENTO_PILHA;

         /* Tenta aumentar o tamanho do array que armazena os elementos da pilha */
      p->elementos = realloc(p->elementos, p->capacidade*p->tamElemento);

         /* Se o redimensionamento não foi possível, aborta o programa */
      ASSEGURA(p->elementos, "Erro de alocacao");
   }

      /* Faz 'pTopo' apontar para o local onde o novo elemento será empilhado */
   pTopo = (char *)p->elementos + p->nElementos*p->tamElemento;
```

```
      /* Copia o conteúdo do novo item para o local determinado */
   memcpy(pTopo, pItem, p->tamElemento);

   p->nElementos++; /* Mais um elemento foi empilhado */
}
```

A função `EmpilhaG()` funciona da seguinte maneira:

1. Se o array que armazena os elementos da pilha estiver repleto, tenta-se aumentar seu tamanho. Se essa tentativa for malsucedida, o programa é abortado.
2. Armazena-se no ponteiro `pTopo` o endereço do local onde o novo item será empilhado. Esse endereço é calculado somando-se o endereço do primeiro elemento do array (determinado por `p->elementos`) ao produto do número atual de elementos armazenados na pilha pelo tamanho de cada elemento (i.e., `p->nElementos*p->tamElemento`). É importante notar que o operador de conversão explícita **(char *)** aplicado ao ponteiro `p->elementos` é obrigatório, visto que esse ponteiro é do tipo **void** * [i.e., o tipo da variável para a qual ele aponta é indefinido].
3. Copia-se o conteúdo do elemento a ser empilhado usando-se a função **memcpy**() (`#include <string.h>`). Essa função copia um número especificado de bytes de um bloco de memória para outro e tem o seguinte protótipo:

   ```
   void *memcpy(void *destino, const void *origem, size_t n)
   ```

   em que:

   - `destino` é um ponteiro para o bloco de destino.
   - `origem` é um ponteiro para o bloco de origem.
   - `n` é o número de bytes que serão copiados.

   A função **memcpy**() retorna o endereço do bloco que recebe a cópia. No caso da função `EmpilhaG()` acima, esse valor retornado não precisa ser usado.
4. Finalmente, incrementa-se o número de elementos correntemente empilhados.

11.4.6 Desempilhamento

A função `DesempilhaG()`, apresentada a seguir, implementa a operação de desempilhamento numa pilha genérica. Se você entendeu bem o funcionamento da função `EmpilhaG()`, não terá dificuldade em entender a função a seguir acompanhando os comentários nela inseridos.

```
void *DesempilhaG(tPilhaGen *p, void *pItem)
{
   const char *pTopo; /* Apontará para o elemento no topo da pilha */

      /* Se a pilha estiver vazia, aborta o programa */
   ASSEGURA(!PilhaVaziaG(p), "A pilha esta' vazia");

   p->nElementos--; /* A pilha ficará com um elemento a menos */

      /* Faz 'pTopo' apontar para o endereço do elemento no topo da    */
      /* pilha. Se o número de elementos não tivesse sido decrementado */
      /* antes, 'pTopo' apontaria para o local onde o próximo elemento */
      /* seria empilhado (i.e., para um elemento adiante)              */
   pTopo = (const char *)p->elementos + p->nElementos*p->tamElemento;

      /* Copia o conteúdo do elemento desempilhado */
   memcpy(pItem, pTopo, p->tamElemento);

   return pItem;
}
```

11.4.7 Elemento do Topo

A função `ElementoTopoG()` apresentada a seguir obtém o elemento do topo de uma de uma pilha genérica. Se você entendeu bem o funcionamento da função `DesempilhaG()`, não terá dificuldade em entender a função `ElementoTopoG()`.

```
void *ElementoTopoG(const tPilhaGen *p, void *pItem)
{
   const char *pTopo; /* Apontará para o elemento no topo da pilha */

      /* Se a pilha estiver vazia, apresenta mensagem de erro e aborta o programa */
   ASSEGURA(!PilhaVaziaG(p), "\nErro: Pilha vazia\n");

      /* Faz 'pTopo' apontar para o elemento no topo da pilha  */
   pTopo = (const char *)p->elementos + (p->nElementos - 1)*p->tamElemento;

      /* Copia o conteúdo do elemento que se encontra no topo da pilha */
   memcpy(pItem, pTopo, p->tamElemento);

   return pItem; /* Retorna o endereço do conteúdo copiado */
}
```

11.5 Usando bsearch() e qsort()

As funções **bsearch()** e **qsort()** da biblioteca padrão de C são utilizadas, respectivamente, para busca binária (v. **Seção 7.2.3**) e ordenação de arrays (v. **Seção 3.12.2**). Essas funções são declaradas no cabeçalho `<stdlib.h>` e têm em comum o fato de ambas utilizarem ponteiros para funções como parâmetros. A função **bsearch()** retorna o endereço do valor procurado, se ele for encontrado; caso contrário, ela retorna **NULL**. O protótipo dessa função é:

```
void *bsearch( const void *chave, const void *array,
               size_t nElem, size_t tamanho,
               int (*FComp)(const void *e1, const void *e2)
```

Nesse protótipo, os parâmetros devem ser interpretados como:

- `chave` é um ponteiro para a chave a ser procurada
- `array` é um ponteiro para o array a ser examinado
- `nElem` é o número de elementos do array
- `tamanho` é o tamanho de cada elemento do array
- `FComp` é um ponteiro para uma função que compara os elementos do array (v. discussão adiante)

A função **qsort()** tem protótipo semelhante ao da função **bsearch()**:

```
void qsort( void *array, size_t nElem, size_t tamanho,
            int (*FComp) (const void *, const void *) )
```

Os parâmetros que possuem as mesmas denominações nos dois protótipos acima, possuem idêntica interpretação.

O problema que muitos programadores encontram ao tentar utilizar essas duas funções frequentemente é a passagem do parâmetro que representa o endereço de uma função com dois parâmetros do tipo **const void** *, cada um dos quais aponta para um elemento do array a ser pesquisado [no caso de **bsearch()**] ou ordenado [no caso de **qsort()**]. Essa função deve comparar dois elementos do array passado como parâmetro e retornar um valor do tipo **int** de acordo com os seguintes critérios de comparação:

- Um valor menor do que zero, se o primeiro elemento for menor do que o segundo
- Zero, se os elementos forem iguais
- Um valor maior do que zero, se o primeiro elemento for maior do que o segundo

A função de comparação descrita acima supõe que o array está [no caso de **bsearch**()] ou será [no caso de **qsort**()] ordenado em ordem crescente. Se, em qualquer dos casos, a ordem for decrescente, deve-se trocar menor por maior e vice-versa na especificação de retorno da função de comparação.

De acordo com os protótipos de **bsearch**() e **qsort**(), o protótipo da função de comparação, cujo endereço deve ser passado como último parâmetro dos protótipos apresentados acima, deve ser:

```
int Comparacao(const void *, const void *)
```

Como se vê acima, a função de comparação possui dois parâmetros, cada um dos quais é um ponteiro genérico. Esses ponteiros devem apontar para os elementos do array que serão comparados. Portanto, para comparar os dois elementos do array, é preciso acessar os conteúdos para os quais esses ponteiros apontam. Mas, para que esse acesso seja possível, é imperioso prover uma interpretação para esses ponteiros, pois, sem essa interpretação, um compilador não possui subsídios para interpretar os conteúdos para os quais eles apontam. De fato, é ilegal tentar acessar conteúdos apontados por ponteiros genéricos. Existem duas maneiras para prover a aludida interpretação, que serão melhor compreendidas por meio de exemplos.

Suponha, por exemplo, que um array a ser ordenado por **qsort**() seja composto de elementos do tipo **int**. Então, a função de comparação poderia ser implementada corretamente como:

```
int Compara1(const void *p1, const void *p2)
{
   const int *e1 = (const int *)p1; /* Provê interpretação */
   const int *e2 = (const int *)p2; /* para os parâmetros  */

      /* Se *e1 < *e2, o retorno será negativo; */
      /* se *e1 == *e2, o retorno será 0;       */
      /* se *e1 > *e2, o retorno será positivo  */
   return *e1 - *e2;
}
```

Alternativamente, essa função poderia ser implementada de modo mais sucinto como:

```
int Compara2(const void *p1, const void *p2)
{
   return *(int *)p1 - *(int *)p2;
}
```

Finalmente, essa função poderia ser ainda implementada como:

```
int Compara3(const int *p1, const int *p2)
{
   return *p1 - *p2;
}
```

Nesse último caso, a referida interpretação dos ponteiros genéricos é efetuada na declaração dos parâmetros da função. Essa última definição de função não irá agradar o compilador que emitirá uma mensagem de advertência informando que o último parâmetro de **bsearch**() não é compatível com o endereço dessa última função.

Agora que você já aprendeu a implementar uma função simples de comparação que pode ser usada com **bsearch**() ou **qsort**(), pode-se considerar um caso mais sofisticado de uso de **qsort**().

Suponha que se deseje ordenar usando **qsort**() um array de strings constantes definido, por exemplo, como:

```
char *timao[] = { "Neuer", "Lahm", "Boateng", "Hummels", "Howedes", "Schweinsteiger",
                  "Khedira", "Kroos", "Ozil", "Muller", "Klose" };
```

Para realizar essa tarefa, pode-se começar definindo a função de comparação cujo endereço deve ser passado como parâmetro para **qsort**(). Conforme foi visto acima, essa função deve prover uma interpretação para os

ponteiros genéricos que apontam para dois elementos do array a ser ordenado. Nesse passo, um erro muito frequente entre programadores desatentos é escrever essa função como:

```
int ComparaErrado(const void *p1, const void *p2)
{
   const char *e1 = (const char *)p1;
   const char *e2 = (const char *)p2;

   return *e1 - *e2;
}
```

Mas, se você cotejar essa última função com a função `Compara1()` apresentada antes, verá que ela compara caracteres e não strings. Então, talvez, uma outra opção para o programador desatento seja definir a função de comparação como:

```
int ComparaErrado2(const void *p1, const void *p2)
{
   const char *e1 = (const char *)p1;
   const char *e2 = (const char *)p2;

   return strcmp(e1, e2);
}
```

Mas, infelizmente, essa última opção também não funciona porque cada parâmetro dessa função é um ponteiro para um string que, por sua vez, também é representado por um ponteiro para o tipo **char**. Portanto, em vez de converter `p1` e `p2` usando `(const char *)`, deve-se efetuar essa conversão usando `(const char **)`. Então, aplica-se o operador de indireção sobre o resultado para obter os strings desejados. Enfim, a solução correta para a almejada função de comparação é:

```
int Compara1(const void *s1, const void *s2)
{
   const char *str1 = *(char **)s1;
   const char *str2 = *(char **)s2;
   return strcmp(str1, str2);
}
```

ou

```
int Compara2(const void *s1, const void *s2)
{
   return strcmp(*(char **)s1, *(char **)s2);
}
```

Assim a função **qsort()** pode ser chamada para ordenar o array `timao[]` apresentado acima como:

```
qsort(timao, n, sizeof(timao[0]), Compara1);
```

11.6 Conversões e Avaliações de Expressões Aritméticas

11.6.1 Formas de Representação de Expressões Aritméticas

Existem três formas de representação de expressões aritméticas: **infixa**, **prefixa** e **sufixa**. Essas denominações dizem respeito às posições de operadores com relação aos seus respectivos operandos. Por exemplo, a soma dos operandos `A` e `B` seria representada em cada uma dessas formas como:

- Infixa: `A+B`
- Prefixa: `+AB`
- Sufixa: `AB+`

Observe no exemplo acima que, na notação prefixa, o operador precede os dois operandos; na notação sufixa, o operador segue os dois operandos e, na notação infixa, o operador aparece entre os dois operandos. Quando o operador é unário, ele precede seu único operando na forma infixa. As notações prefixa e sufixa são denominadas **notações polonesas**.

A avaliação de uma expressão escrita na forma infixa, tal como `A+B*C`, requer conhecimento sobre a ordem de execução das operações (i.e., sobre qual dos operandos, + ou *, deve ser aplicado primeiro). Neste exemplo, sabe-se que a multiplicação deve ser efetuada antes da soma. Assim a expressão `A+B*C` deve ser interpretada como `A + (B*C)` [e não (`A + B)*C`]. Por isso, diz-se que a multiplicação tem maior precedência que a soma.

Para representar uma expressão em forma sufixa, devem-se escrever os operandos seguidos de seus respectivos operadores. No caso da expressão `A+B*C`, os operandos da soma são `A` e a (sub)expressão `B*C`, que é escrita como `BC*` em forma sufixa. Assim a expressão `A+B*C` é escrita em forma sufixa como `ABC*+`.

A conversão de uma expressão infixa em sufixa é simples desde que se conheçam as regras de precedência e associatividade dos operadores. Na discussão que segue, serão considerados apenas os cinco operadores aritméticos de C: adição, subtração, multiplicação, divisão e menos unário. As mesmas regras de precedência e associatividade de C também serão usadas aqui (v. **Apêndice A**). Além disso, os símbolos que representam esses operadores em C serão os mesmos aqui, com exceção do símbolo que representa menos unário, que, por simplicidade, aqui será representado por subtraço (_).

Deve-se observar que a ordem dos operandos é a mesma em qualquer forma de uma expressão. Além disso, durante a conversão de uma expressão da forma infixa para sufixa, as operações com maior precedência devem ser convertidas primeiro. Então, depois que uma porção da expressão tiver sido convertida, ela deve ser tratada como um único operando.

Um algoritmo manual que pode ser seguido até que se adquira prática na conversão da forma infixa para a forma sufixa consiste da seguinte sequência de passos:

1. Envolva entre parênteses toda a expressão escrita na forma infixa, bem como todas suas subexpressões, de acordo com as regras de precedência e associatividade dos operadores contidos nela, de modo que a cada operador corresponda exatamente um par de parênteses.
2. Desloque cada operador para a posição adjacente ao seu fecha parênteses correspondente.
3. Remova todos os parênteses e o resultado será a expressão escrita na forma sufixa.

Seguindo os passos desse algoritmo na transformação da expressão `A + B*C` para a forma sufixa, obtém-se:

- **Passo 1**: `(A + (B*C))`
- **Passo 2**: `(A (BC*)+)`
- **Passo 3**: `ABC*+`

As regras utilizadas para converter uma expressão da forma infixa para a forma prefixa são idênticas àquelas vistas para a conversão em forma sufixa. A única diferença é que, na transformação em forma prefixa, os operadores são colocados antes dos operandos, ao invés de depois como no caso da conversão para a forma sufixa. Assim, no caso de transformação da forma infixa para prefixa, pode-se utilizar um algoritmo semelhante ao apresentado para transformação de expressões para a forma sufixa, apenas reescrevendo-se o **Passo 2** daquele algoritmo como:

2. Desloque cada operador para a posição adjacente ao seu abre parênteses correspondente.

Embora as formas prefixa e sufixa não mostrem de forma muito legível a quais operandos está associado um dado operador (como ocorre na forma infixa), essas duas notações possuem duas grandes vantagens. Primeiro, elas tornam desnecessário o uso de quaisquer parênteses e, segundo, elas tornam desnecessário qualquer conhecimento

sobre precedência ou associatividade de operadores. Com efeito, a ordem das operações é exatamente a ordem em que os operadores aparecem na forma sufixa. Ou seja, na avaliação de uma expressão na forma sufixa, quando é encontrado um operador binário ele é logo aplicado aos operandos que imediatamente o precedem. Se o operador for unário, ele é aplicado ao único operando que imediatamente o precede.

Considere, por exemplo, a avaliação de `3 4 5 * +` (que corresponde, na forma infixa, a `3 + 4*5`). Examinando-se a expressão da esquerda para a direita, o primeiro operador encontrado é `*`, que é aplicado aos operandos `4` e `5`, resultando em `20`. Então, a expressão transforma-se em `3 20 +` e o operador + é aplicado aos operandos `3` e `20` produzindo `23`, que é o resultado esperado.

Serão apresentados a seguir algoritmos para avaliar uma expressão na forma sufixa e para converter uma expressão da forma infixa para a forma sufixa. Algoritmos semelhantes referentes à forma prefixa não serão abordados aqui, pois, afinal, o objetivo maior é demonstrar a utilização de pilhas genéricas na avaliação de expressões.

11.6.2 Avaliação de Expressão Sufixa

Conforme foi visto acima, um operador encontrado numa expressão sufixa aplica-se ao último operando (se o operador for unário) ou aos dois últimos operandos (se o operador for binário). Também foi visto que um operando pode ser o resultado de uma outra operação já efetuada. Agora, suponha que se esteja examinando uma expressão e que, cada vez que for encontrado um operando, ele seja colocado numa pilha. Então, para cada operador encontrado, ter-se-á que seus operandos estarão no topo da pilha. Mais precisamente, se o operador for unário, seu operando encontra-se no topo da pilha. Se o operador for binário, seu primeiro operando encontra-se no topo e seu segundo operando estará no topo após seu primeiro operando ser desempilhado. Pode-se, então, desempilhar os operandos do operador encontrado, executar a operação correspondente sobre eles e empilhar o resultado, de modo que o resultado esteja disponível para uso como operando da próxima operação (se ele existir). O algoritmo da **Figura 11–3** traduz o que foi exposto até aqui:

ALGORITMO AVALIAEXPRESSÃOSUFIXA

ENTRADA: Expressão *E* na forma sufixa
SAÍDA: Resultado da avaliação da expressão *E*

1. Crie uma pilha *p*
2. Enquanto E ainda não foi totalmente explorada, faça:
   - **2.1** Atribua a *token* o próximo token de *E*
   - **2.2** Se *token* for um operando, empilhe-o em *p*
   - **2.3** Caso contrário (*token* é um operador):
     - **2.3.1** Desempilhe o(s) operando(s) da pilha *p*
     - **2.3.2** Execute a operação correspondente
     - **2.3.3** Empilhe o resultado da operação na pilha *p*
3. Retorne o elemento que se encontra no topo da pilha *p*

FIGURA 11–3: ALGORITMO DE AVALIAÇÃO DE EXPRESSÃO SUFIXA

11.6.3 Conversão de Forma Infixa para Forma Sufixa

A ordem dos operandos é a mesma nas formas infixa e sufixa de uma expressão. Assim, ao se examinar uma expressão infixa com o intuito de transformá-la em sufixa, os operandos podem ser imediatamente anexados à expressão sufixa à medida que forem sendo encontrados. Portanto a questão reduz-se a como anexar os operadores. Uma boa ideia é armazená-los em uma pilha até o momento em que devam ser desempilhados e anexados à expressão sufixa. Agora, o problema fica reduzido à escolha certa de tal momento.

Suponha que a expressão a ser transformada para a forma sufixa seja `A+B*C`. Então, o algoritmo de conversão deve proceder conforme esquematizado na **Tabela 11–2**. Nessa tabela, *Token Corrente* refere-se ao token ora levado em consideração na expressão infixa.

Neste ponto, o algoritmo deve decidir se o operador `*` deve ser empilhado ou se o operador `+` deve ser desempilhado e anexado à forma sufixa. Como `*` tem prioridade maior do que `+`, o operador `*` deve ser empilhado e o estado de conversão torna-se aquele mostrado na **Tabela 11–3**.

Token Corrente	Pilha	Expressão Sufixa
A	*Vazia*	*A*
+	+	*A*
B	+	*AB*

Tabela 11–2: Conversão da Expressão *A + B*C* para a Forma Sufixa 1

Token Corrente	Pilha	Expressão Sufixa
*	+*	*AB*
C	+*	*ABC*

Tabela 11–3: Conversão da Expressão *A + B*C* para a Forma Sufixa 2

Explorada toda a expressão infixa, os operadores remanescentes na pilha devem ser desempilhados e anexados à expressão sufixa, resultando em `ABC*+`.

Considere, como outro exemplo, a expressão `A*(B+C)/D` que tem as precedências dos operadores modificadas por parênteses. O algoritmo de conversão deveria ser responsável pela sequência de eventos ilustrada na **Tabela 11–4** e na **Figura 11–4**.

Token Corrente	Pilha	Expressão Sufixa
A	*Vazia*	*A*
*	*	*A*
(	*(	*A*
B	*(	*AB*
+	*(+	*AB*
C	*(+	*ABC*

Tabela 11–4: Conversão da Expressão *A*(B + C)/D* para a Forma Sufixa 1

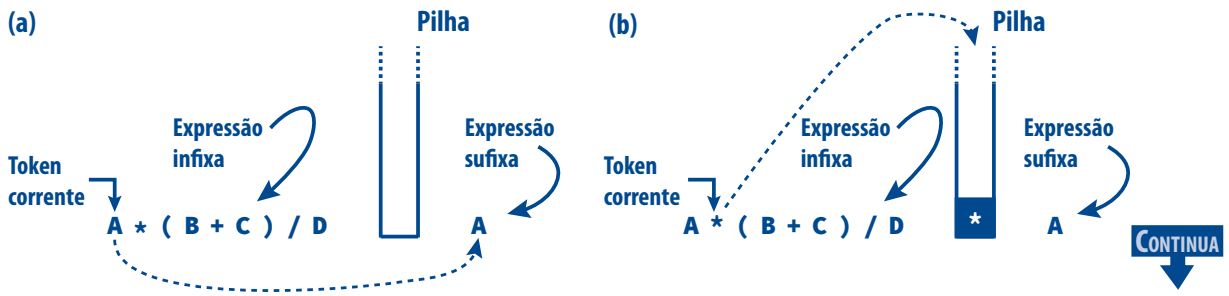

Figura 11–4: Conversão da Expressão A*(B + C)/D para a Forma Sufixa 1

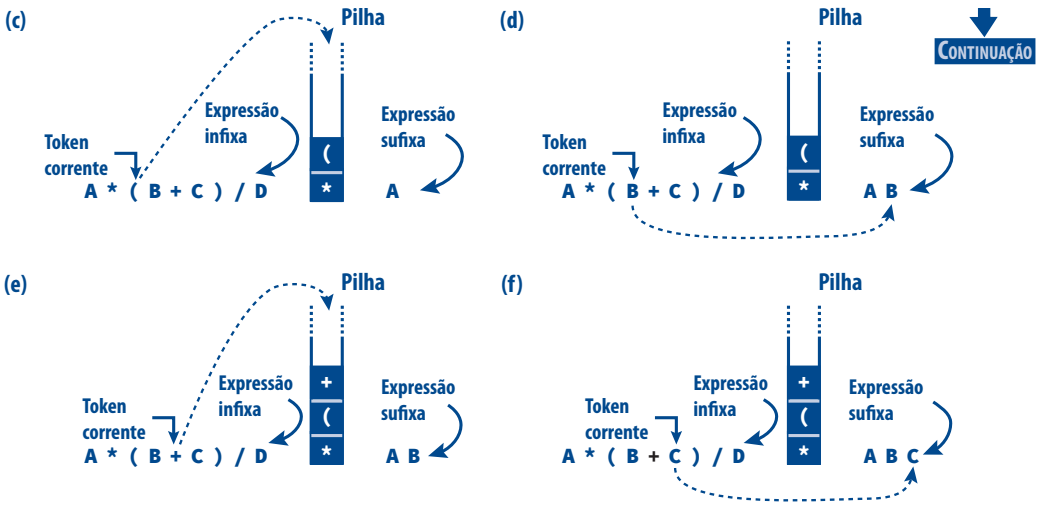

FIGURA 11–4 (CONT.): CONVERSÃO DA EXPRESSÃO A*(B + C)/D PARA A FORMA SUFIXA 1

Neste ponto, deve-se desempilhar até o abre parênteses e descartá-lo, o que resulta nas ações mostradas na **Tabela 11–5** e na **Figura 11–5**.

TOKEN CORRENTE	PILHA	EXPRESSÃO SUFIXA
)	*	*ABC+*
/	*/*	*ABC+**
D	*/*	*ABC+*D*
Nenhum	*Vazia*	*ABC+*D/*

TABELA 11–5: CONVERSÃO DA EXPRESSÃO *A*(B + C)/D* PARA A FORMA SUFIXA 2

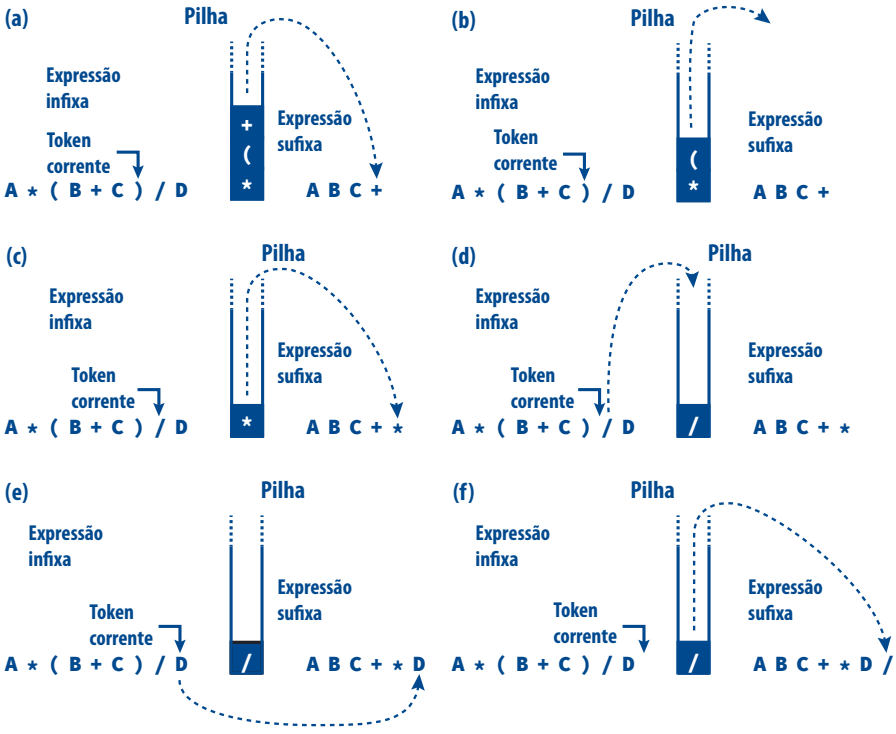

FIGURA 11–5: CONVERSÃO DA EXPRESSÃO A*(B + C)/D PARA A FORMA SUFIXA 2

Os dois últimos exemplos sugerem a existência de ordens de prioridades para operadores e parênteses. A regra a ser adotada é que os operadores são desempilhados sempre que sua *prioridade dentro da pilha* (**PDP**) for maior do que ou igual à *prioridade de chegada* (**PC**) do novo operador. Os valores absolutos dessas prioridades, mostrados na **Tabela 11–6**, são arbitrários, mas eles devem ser escolhidos de tal modo que seus valores resultem no efeito desejado.

Token	Prioridade Dentro da Pilha	Prioridade de Chegada
_	3	4
*, /	2	2
+, -	1	1
(	0	4

Tabela 11–6: Precedências em Conversão de Forma Infixa para Forma Sufixa

O algoritmo apresentado na **Figura 11–6** considera a existência das funções `PDP(token)` e `PC(token)` que representam as prioridades apresentadas na **Tabela 11–6**.

Algoritmo ConverteExpressãoInfixaEmSufixa

Entrada: Expressão *Ei* na forma infixa

Saída: Expressão *Es* na forma sufixa

1. Crie uma pilha *p*

2. Repita o seguinte:

- **2.1** Atribua a *token* o próximo token de *Ei*
- **2.2** Se a expressão *Ei* foi totalmente explorada então
  - **2.2.1** Enquanto a pilha *p* não estiver vazia, faça:
    - **2.2.1.1** Desempilhe um item da pilha *p*
    - **2.2.1.2** Anexe o item desempilhado a *Es*
  - **2.2.2** Retorne
- **2.3** Caso contrário, se *token* for um operando, anexe *token* a *Es*
- **2.4** Caso contrário, se *token* = *')'*
  - **2.4.1** Enquanto o elemento do topo de $p \neq$ *'('*, faça:
    - **2.4.1.1** Desempilhe um item da pilha *p*
    - **2.4.1.2** Anexe o item desempilhado a *Es*
  - **2.4.2** Desempilhe um item da pilha *p* (para descartar *'('*)
- **2.5** Caso contrário (*token* é um operador ou *'('*):
  - **2.5.1** Enquanto a pilha *p* não estiver vazia e *PDP(ElementoTopo(p)* $\geq$ *PC(token)*, faça
    - **2.5.1.1** Desempilhe um item da pilha *p*
    - **2.5.1.2** Anexe o item desempilhado a *Es*
  - **2.5.2** Empilhe *token* na pilha *p*

Figura 11–6: Algoritmo de Conversão de Expressão Infixa em Sufixa

11.7 Exemplos de Programação

11.7.1 Ordenação Generalizada de Listas Indexadas

Problema: (a) Utilize ponteiros para funções e ponteiros genéricos para generalizar a função `BubbleSort()` apresentada na **Seção 3.12.2**, de tal modo que ela seja capaz de ordenar arrays de qualquer tipo como faz a função **qsort()** apresentada na **Seção 11.5**. (b) Escreva uma função **main()** que chama a função solicitada no item (a) para ordenar um array de elementos do tipo **int** de modos crescente e decrescente

Solução de (a): A função `BubbleSortGen()` apresentada a seguir utiliza um ponteiro para uma função de comparação com as mesmas especificações daquela função de comparação utilizada por **qsort()**. Além disso, o primeiro parâmetro é um ponteiro genérico que deve apontar para o array a ser ordenado e o segundo parâmetro representa o tamanho de cada elemento do array. Esses três parâmetros permitem que a função `BubbleSortGen()` possa ser utilizada para ordenar arrays de elementos de quaisquer tipos.

```
void BubbleSortGen( void *lista, int tamElemento, int nElementos,
                    int (*compara) (const void *, const void *) )
{
   int   i, ordenada = 0;
   char *aux;

      /* Tenta alocar um bloco que auxilia a troca de elementos do array */
   ASSEGURA(aux = malloc(tamElemento), "Impossivel alocar bloco para ordenar array");

   while (!ordenada) {
      ordenada = 1;  /* Supõe que a lista está ordenada */

      for (i = 0; i < nElementos - 1; i++){
         if (compara( (char *) lista + i*tamElemento,
                      (char *) lista + (i+1)*tamElemento) > 0 ){
            ordenada = 0;

               /* Troca elementos adjacentes */
            memcpy( aux, (char *)lista + i*tamElemento, tamElemento );
            memcpy( (char *) lista + i*tamElemento,
                    (char *) lista + (i + 1)*tamElemento, tamElemento );
            memcpy( (char *) lista + (i + 1)*tamElemento, aux, tamElemento );
         }
      }
   }

   free(aux); /* Libera o espaço que foi alocado dinamicamente */
}
```

Observações sobre a função `BubbleSortGen()`:

- A lista de parâmetros da função é semelhante àquela da função **qsort()** da biblioteca padrão (v. **Seção 11.5**).
- A variável **aux**, definida como ponteiro para **char**, apontará para um bloco de memória do tamanho de cada elemento do array a ser ordenado. Esse bloco auxilia a troca de valores entre elementos (blocos) adjacentes desse array e é alocado dinamicamente. Quando não é possível alocar espaço para esse bloco a função causa o aborto do programa no qual é chamada.
- O endereço do i-ésimo elemento do array é obtido por meio da avaliação da expressão:

```
(char *) lista + i*tamElemento
```

- As chamadas da função **memcpy**() (`#include <string.h>`) efetuam trocas de valores entre elementos adjacentes. Essa função copia `n` bytes de um bloco de memória para outro e foi descrita em detalhes na **Seção 11.4.5**.

Solução de (b): A função **main**() a seguir chama a função `BubbleSortGen()` com um array de elementos do tipo **int** para classificá-lo nas ordens crescente e decrescente:

```
int main(void)
{
   int ar[] = {12, 55, 21, 1, 6, 8, 17, 220, 5, 83},
       i;

   printf("\t>>> Ordem original <<<\n");
   for (i = 0; i < sizeof(ar)/sizeof(ar[0]); i++)
      printf(" %d ", ar[i]);

   BubbleSortGen( ar, sizeof(ar[0]), sizeof(ar)/sizeof(ar[0]), ComparaCrescente );

   printf("\n\n\t>>> Ordem crescente <<<\n");
   for (i = 0; i < sizeof(ar)/sizeof(ar[0]); i++)
     printf(" %d ", ar[i]);

   BubbleSortGen(ar, sizeof(ar[0]), sizeof(ar)/sizeof(ar[0]), ComparaDecrescente);

   printf("\n\n\t>>> Ordem decrescente <<<\n");
   for (i = 0; i < sizeof(ar)/sizeof(ar[0]); i++)
     printf(" %d ", ar[i]);

   return 0;
}
```

As funções responsáveis por comparar elementos do array são definidas como:

```
int ComparaCrescente(const void *a, const void *b)
{
   return *(int *)a - *(int *)b;
}

int ComparaDecrescente(const void *a, const void *b)
{
   return *(int *)b - *(int *)a;
}
```

11.7.2 Implementando Conversão de Expressões Infixas em Sufixas

Problema: Escreva funções que implementam a conversão de expressões infixas em expressões sufixas discutida na **Seção 11.6**.

Solução: Para implementação dessas funções, as seguintes definições de tipos serão necessárias:

```
typedef struct {
           char operador; /* O operador  */
           int  aridade;  /* Sua aridade */
           int  pdp, /* Sua prioridade PDP */
                pc;  /* Sua prioridade PC  */
        } tOperador;

typedef enum {PDP, PC} tPrioridade;
```

O tipo `tOperador` é utilizado para definir variáveis e parâmetros que armazenam informações sobre operadores, enquanto o tipo `tPrioridade` representa os dois tipos de prioridade discutidos na **Seção 11.6**:

- `PDP`: *prioridade dentro da pilha*
- `PC`: *prioridade de chegada*

Por questão de eficiência, é conveniente definir a seguinte variável (array) com escopo de arquivo, já que ela é usada por várias funções do módulo `Expressoes`:

```
static const tOperador operadores[] = { {'_', 1, 3, 4},
                                        {'*', 2, 2, 2},
                                        {'/', 2, 2, 2},
                                        {'+', 2, 1, 1},
                                        {'-', 2, 1, 1},
                                        {'(', 0, 0, 4}
                                      };
```

O array `operadores[]` definido acima armazena informações sobre operadores e abre parênteses. Também é conveniente definir uma variável com escopo de arquivo que armazene o tamanho desse array:

```
static const int nOperadores = sizeof(operadores) / sizeof(operadores[0]);
```

A função `ExpressaoSufixa()` definida abaixo cria um string contendo um expressão sufixa correspondente à expressão infixa recebida como parâmetro. Um programa que chama essa função será abortado se o array que armazenará a expressão sufixa não puder ser alocado ou se houver um fecha parênteses sem o respectivo abre parênteses na expressão infixa.

```
char *ExpressaoSufixa(const char *infixa)
{
   tPilhaGen pilha; /* Pilha que armazenará operadores e abre-parênteses */
   char     *inicio, /* Apontará para o início de um token*/
            *fim; /* Apontará para o final de um token */
   char      e; /* Elemento da pilha */
   char     *sufixa; /* Ponteiro para a expressão sufixa */
   int       tamSufixa; /* Tamanho da expressão sufixa */

      /* O tamanho inicial da expressão sufixa é estimado como  */
      /* sendo igual ao da expressão infixa. Se for necessário, */
      /* esse tamanho será devidamente aumentado.               */
   tamSufixa = strlen(infixa) + 1;

      /* Tenta alocar espaço para conter a expressão sufixa */
   ASSEGURA( sufixa = calloc(tamSufixa, sizeof(char)),
             "Erro em ExpressaoSufixa(): Impossivel alocar espaco para a expressao" );

      /* Inicia a pilha que armazenará operadores */
   CriaPilhaG(&pilha, sizeof(char));

      /* Obtém o primeiro token */
   inicio = ProximoToken(infixa, &fim);

      /* O laço a seguir encerra quando não houver mais tokens na expressão infixa */
   while (inicio) {
      if (*inicio == ')') {
            /* O token corrente é um fecha-parênteses. Então, a pilha deve */
            /* ser desempilhada até o correspondente abre-parênteses.      */
         while (!PilhaVaziaG(&pilha) && *(char *)(ElementoTopoG(&pilha, &e)) != '('){
            DesempilhaG(&pilha, &e);
            AnexaEmExpressao(&sufixa, &tamSufixa, &e, 1);
         }
```

```
            /* Neste ponto, se a pilha estiver vazia ou se o próximo token    */
            /* desempilhado não for abre-parênteses, o programa será abortado */
         ASSEGURA(!PilhaVaziaG(&pilha) && *(char *) DesempilhaG(&pilha, &e) == '(',
                  "Erro: Nao foi encontrado '('");
      } else if (EhOperador(*inicio)) {
            /* O token corrente é um operador ou abre-parênteses. Então,  */
            /* enquanto a pilha não estiver vazia e a prioridade PDP do   */
            /* token no topo da pilha for maior ou igual que a prioridade */
            /* PC do token corrente, o token encontrado no topo da pilha  */
            /* é desempilhado e anexado à expressão sufixa.               */
         while (!PilhaVaziaG(&pilha) &&
               Prioridade(*(char *)ElementoTopoG(&pilha,&e), PDP) >=
                Prioridade(*inicio, PC) ) {
            DesempilhaG(&pilha, &e);
            AnexaEmExpressao(&sufixa, &tamSufixa, &e, 1);
         }
            /* Empilha o token corrente. NB: Apesar de se estar passando */
            /* um ponteiro para um string como segundo parâmetro, apenas */
            /* o primeiro caractere desse string será empilhado.         */
         EmpilhaG(&pilha, inicio);
      } else {
            /* O token corrente é um operando, que é    */
            /* imediatamente anexado à expressão sufixa */
         AnexaEmExpressao( &sufixa, &tamSufixa, inicio, fim - inicio + 1 );
      }
      inicio = ProximoToken(NULL, &fim); /* Obtém o próximo token */
   }
      /* Neste ponto, a pilha contém apenas operadores que */
      /* serão desempilhados e anexados à expressão sufixa */
   while (!PilhaVaziaG(&pilha)) {
      DesempilhaG(&pilha, &e);
      AnexaEmExpressao(&sufixa, &tamSufixa, &e, 1);
   }
   DestroiPilhaG(&pilha); /* Libera espaço ocupado pela pilha */
   return sufixa; /* Serviço completo */
}
```

A função `ExpressaoSufixa()`, apresentada acima, utiliza uma pilha genérica do tipo discutido na **Seção 11.4**. Essa pilha é utilizada para empilhar e desempilhar operadores e abre parênteses, conforme o algoritmo discutido na **Seção 11.6**. Além disso, a função `ExpressaoSufixa()` utiliza as seguintes funções auxiliares, que serão apresentadas adiante:

- `ProximoToken()` que obtém o próximo token da expressão infixa recebida como parâmetro. Aqui, *token* significa operando, operador, abre parênteses ou fecha parênteses.
- `AnexaEmExpressao()` que anexa um novo termo (i.e., operando ou operador) à expressão sufixa que resultará da conversão.
- `EhOperador()` que verifica se um caractere é um operador ou abre parênteses.
- `Prioridade()` que retorna a prioridade (PDP/PC) de um operador.

A função `ProximoToken()`, vista a seguir, retorna o endereço inicial do token, se ele existir ou **NULL**, em caso contrário. O parâmetro `expressao` (entrada) é um string que representa a expressão, ao passo que `fim` (saída) é o endereço de um ponteiro para o último caractere do token.

```
char *ProximoToken(const char *expressao, char **fim)
{
   static char *proximoToken; /* Aponta para o próximo token, se ele existir */
   char        *s, *inicio;

   if (expressao) { /* A procura iniciará em 'expressao' */
      s = (char *)expressao;
   } else if (proximoToken) { /* A procura iniciará */
      s = proximoToken;       /* em 'proximoToken'  */
   } else {/* 'expressao' e 'proximoToken' são ambos NULL */
      return NULL; /* Não há token */
   }

      /* O laço a seguir salta eventuais espaços em branco que porventura se */
      /* encontrem no início do string. Ele encerra quando for encontrado um */
      /* caractere que não seja espaço em branco ou o caractere terminal do  */
      /* string apontado por s.                                              */
   while (*s && isspace(*s))
      ++s; /* Salta mais um espaço */

      /* Se o laço anterior encerrou porque foi encontrado o */
      /* caractere terminal, não há  mais tokens no string   */
   if (!*s) {
      proximoToken = NULL; /* Não haverá próximo token */

      return NULL; /* Nem há agora */
   }

   inicio = s; /* Guarda o início do token corrente */

   /*********************************************************************/
   /* Como, por simplicidade, parênteses são considerados operadores, só */
   /* há dois tipos de tokens: operadores e operandos numéricos inteiros */
   /*********************************************************************/

      /* Verifica se o próximo caractere é um operador (incluindo parênteses) */
   if (EhOperador(*s)) { /* Trata-se de um operador */
      *fim = inicio; /* Operador só usa um caractere */

      proximoToken = inicio + 1; /* O proximo token iniciará no próximo caractere */

      return inicio; /* Serviço concluído */
   }

      /* Se o caractere corrente não era um operador, ele deve ser um */
      /* dígito. Se ele não for um dígito, o programa será abortado.  */
   ASSEGURA(isdigit(*s), "Erro em ProximoToken(): caractere espurio encontrado");

      /* Enquanto o caractere corrente for um dígito, */
      /* faz s apontar para o próximo caractere       */
   while (isdigit(*s))
      ++s; /* Passa para o próximo caractere */

      /* Neste ponto, s aponta para o primeiro caractere que não */
      /* é dígito. Esse será o início do próximo token.          */
   proximoToken = s;

   *fim = s - 1; /* Faz *fim apontar para o último dígito encontrado */

   return inicio; /* Retorna o endereço inicial do token corrente */
}
```

A função `AnexaEmExpressao()` acrescenta um novo termo a uma expressão. Seus parâmetros são:

- `expressao` (entrada/saída) é um ponteiro para um array contendo um string que representa a expressão
- `tam` (entrada/saída) é um ponteiro para o tamanho do array que contém a expressão
- `termo` (entrada) representa o array de caracteres contendo o novo termo (esse parâmetro não é string)
- `tamTermo` (entrada) é o número de caracteres do termo

```
void AnexaEmExpressao(char **expressao, int *tam, char *termo, int tamTermo)
{
   char *p, *str;
   int   tamExpressao, i;

   tamExpressao = strlen(*expressao) + /* Tamanho corrente da expressão */
                  tamTermo + /* Tamanho do novo termo */
                  2;   /* Caractere terminal mais espaço em branco */

      /* Se o número de caracteres necessários for maior do que o */
      /* tamanho do array, tenta-se aumentar o tamanho do array   */
   if (tamExpressao > *tam){ /* Se a realocação não for possível, aborta */
      ASSEGURA( (p = realloc(*expressao, tamExpressao)),
                "Erro em AnexaEmExpressao(): Realocacao nao foi possivel" );

      *tam = tamExpressao; /* Atualiza o tamanho do array */

      *expressao = p; /* Atualiza ponteiro que aponta para a expressão */
   }

      /* O uso do ponteiro 'str' não é essencial, mas facilita */
      /* a escrita e a legibilidade das instruções seguintes   */
   str = *expressao;

      /* Faz str apontar para o caractere terminal */
      /* do string que representa a expressão      */
   str = str + strlen(str);

      /* Copia os caracteres do novo termo para a expressão, de modo   */
      /* que o primeiro caractere do termo seja armazenado no lugar do */
      /* caractere terminal da expressão e os demais caracteres sejam  */
      /* armazenados nas posições subsequentes                         */
   for (i = 1; i <= tamTermo; ++i)
      *str++ = *termo++;

   *str++ = ' '; /* Armazena um espaço em branco */
   *str = '\0'; /* Termina a nova expressão */
}
```

A função `EhOperador()` verifica se um caractere é um operador e retorna `1`, se esse for caso, ou `0`, em caso contrário.

```
int EhOperador(int c)
{
   static const char *const strOperadores = "_*/+-()";

      /* Se strchr() retornar um ponteiro diferente de NULL, */
      /* o caractere faz parte do string 'strOperadores'     */
   return strchr(strOperadores, c) != NULL;
}
```

A função `Prioridade()` retorna a prioridade (PDP ou PC — v. **Seção 11.6**) de um operador. O primeiro parâmetro representa o operando, enquanto o segundo parâmetro é o tipo de prioridade (PDP ou PC). Se o operador não for encontrado no array que armazena essas prioridades, essa função retorna um valor negativo.

```
int Prioridade(int operador, tPrioridade tp)
{
   int i;

      /* O programa será abortado se o tipo de prioridade não for conhecido */
   ASSEGURA( tp == PDP || tp == PC, "Erro: Tipo de prioridade desconhecida");

      /* Procura o operador no array */
   for (i = 0; i < nOperadores; ++i)
      if (operadores[i].operador == operador)
            /* O operador foi encontrado. Retorna a prioridade do tipo especificado */
         return tp == PDP ? operadores[i].pdp : operadores[i].pc;

      /* O operador não foi encontrado */
   return -1;
}
```

11.7.3 Implementando Avaliação de Expressões Sufixas

Problema: (a) Escreva uma função que avalia expressões sufixas de acordo com o algoritmo discutido na **Seção 11.6**. (b) Escreva uma função que converte uma expressão infixa em sufixa e, então, obtém o resultado da avaliação da referida expressão sufixa.

Solução de (a): A função `AvalExpressaoSufixa()`, definida abaixo, avalia uma expressão sufixa representada pelo string recebido como parâmetro e retorna o resultado dessa expressão.

```
int AvalExpressaoSufixa(char *sufixa)
{
   tPilhaGen pilha; /* Pilha que armazenará operandos */
   int       op1, op2, /* Dois operandos de uma expressão */
             resultado, /* Resultado de uma expressão */
             ari; /* Aridade de um operador */
   char     *token, /* Um token da expressão sufixa */
            *final; /* Apontará para o caractere que termina */
                    /* uma conversão de string em inteiro    */

      /* Inicia a pilha que armazenará operandos */
   CriaPilhaG(&pilha, sizeof(int));

      /* Obtém o primeiro token. Espaço em branco é o único separador de tokens */
   token = strtok(sufixa, " ");

      /* O laço a seguir encerra quando não     */
      /* houver mais tokens na expressão sufixa */
   while (token) {
      if (EhOperador(*token)) { /* Token é operador */
         ari = Aridade(*token); /* Obtém a aridade do operador */

            /* A aridade deve ser 1 ou 2. Se não for */
            /* o caso, o programa será abortado.     */
         ASSEGURA( (ari == 1 || ari == 2),
                   "Erro em AvalExpressaoSufixa(): Aridade deveria ser 1 ou 2" );

         if (ari == 1) {
               /* O operador é unário. Então, desempilha-se um */
               /* operando e obtém-se o resultado da expressão */
            DesempilhaG(&pilha, &op1);

               /* O terceiro parâmetro da chamada de ResultadoOperacao()  */
               /* pode ter qualquer valor, uma vez que ele não será usado */
            resultado = ResultadoOperacao(*token, op1, 0);
         } else {
```

```
            /* O operador é binário. Então, desempilham-se dois */
            /* operandos e obtém-se o resultado da expressão    */
         DesempilhaG(&pilha, &op2);
         DesempilhaG(&pilha, &op1);
         resultado = ResultadoOperacao(*token, op1, op2);
      }

         /* O resultado da operação deve ser empilhado */
      EmpilhaG(&pilha, &resultado);
   } else {
         /* O token corrente é um string que deve */
         /* representar um operando inteiro       */

         /* Obtém o valor inteiro representado pelo string */
      op1 = strtol(token, &final, 10);

      ASSEGURA(!*final, "Conversao de operando falhou");

      EmpilhaG(&pilha, &op1); /* Empilha o operando inteiro */
   }

   token = strtok(NULL, " "); /* Obtém o próximo token da expressão sufixa */
}

   /* O resultado da avaliação está no topo da pilha */
DesempilhaG(&pilha, &resultado);

   /* Teste de consistência: neste ponto, a pilha deve estar vazia */
ASSEGURA( PilhaVaziaG(&pilha),
          "Erro em AvalExpressaoSufixa(): A pilha deveria estar vazia" );

DestroiPilhaG(&pilha); /* Libera espaço ocupado pela pilha */

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

A função `AvalExpressaoSufixa()` utiliza uma pilha genérica, como aquela apresentada na **Seção 11.4**, para armazenar operandos. Além disso, a função `AvalExpressaoSufixa()` utiliza as seguintes funções auxiliares:

- `Aridade()` que retorna a aridade de um operador.
- `ResultadoOperacao()` que calcula o resultado de uma operação aritmética simples envolvendo um operador unário e seu único operando ou um operador binário e seus dois operandos.
- **strtol()** que converte num valor inteiro um string que, supostamente, representa tal valor. Essa função foi discutida na **Seção 3.7**.

As definições dessas funções serão apresentadas a seguir.

A função `Aridade()` retorna a aridade do operador recebido como parâmetro se ele for encontrado. Caso contrário, ela retorna um valor negativo.

```
int Aridade(int operador)
{
   int i;

      /* Procura o operador no array */
   for (i = 0; i < nOperadores; ++i)
      if (operadores[i].operador == operador)
            /* O operador foi encontrado. Retorna sua aridade. */
         return operadores[i].aridade;

   return -1; /* O operador não foi encontrado */
}
```

A função `ResultadoOperacao()`, que será vista a seguir, calcula o resultado de uma operação aritmética simples envolvendo um operador unário e seu único operando ou um operador binário e seus dois operandos. Antes de apresentar essa função, algumas observações são necessárias:

- Se o operador for unário, seu operando é o parâmetro `op1`. Nesse caso, o parâmetro `op2` não é levado em consideração.
- Se o operador for de divisão, `op1` é o dividendo e `op2` é o divisor.
- Se o operador for de subtração, `op1` é o minuendo e `op2` é o subtraendo.
- O programa será abortado se o operador for indeterminado.

```
int ResultadoOperacao(int operador, int op1, int op2)
{
      /* Calcula e retorna o resultado da operação */
   switch (operador) {
      case '_': /* Inversão de sinal */
         return -op1;
         break;
      case '*': /* Multiplicação */
         return op1 * op2;
         break;
      case '/': /* Divisão */
         return op1 / op2;
         break;
      case '+': /* Soma */
         return op1 + op2;
         break;
      case '-': /* Subtração */
         return op1 - op2;
         break;
      default: /* Operador indeterminado */
         ASSEGURA(0, "Nao foi possivel encontrar operador\n");
   }

   return 0;
}
```

Solução de (b): A função `AvalExpressaoInfixa()` converte a expressão infixa recebida como parâmetro em expressão sufixa utilizando a função `ExpressaoSufixa()`, apresentada na **Seção 11.7.2**, e, em seguida, obtém, por meio de uma chamada da função `AvalExpressaoSufixa()` definida na solução do item (a), o resultado da avaliação da referida expressão sufixa.

```
int AvalExpressaoInfixa(const char *infixa)
{
   char *sufixa; /* Apontará para a expressão sufixa correspondente */
   int   resultado; /* Armazenará o resultado da avaliação da expressão */

      /* Obtém a expressão sufixa correspondente */
   sufixa = ExpressaoSufixa(infixa);

      /* Obtém resultado da avaliação da expressão sufixa */
   resultado = AvalExpressaoSufixa(sufixa);

      /* O espaço ocupado pela expressão sufixa foi  */
      /* alocado dinamicamente e é preciso liberá-lo */
   free(sufixa);

   return resultado;
}
```

11.8 Exercícios de Revisão

Ponteiros para Funções (Seção 11.1)

1. Qual é a relação existente entre o nome de uma função e um ponteiro para função?
2. (a) Como a seguinte definição é interpretada em C? (b) O que ocorrerá se o primeiro par de parênteses dessa definição for removido?

```
int (*pf)(int);
```

3. Em que situações práticas ponteiros para funções são usados com mais frequência?
4. Escreva uma declaração apropriada para cada uma das seguintes situações:
   (a) Um ponteiro para uma função que recebe dois parâmetros do tipo **int** e retorna um valor do tipo **int**.
   (b) Um ponteiro para uma função que recebe dois ponteiros para o tipo **int** como parâmetros e retorna um ponteiro para um valor do tipo **int**.
   (c) Uma alusão de função que recebe um ponteiro para função como parâmetro e retorna um ponteiro para um valor do tipo **int**. O parâmetro deve ser compatível com funções que possuem um parâmetro do tipo **int** e retornam um valor do tipo **int**.
5. Suponha que se tenham os seguintes protótipos:

```
void UmaFuncao(double f, double (*pf)(double))

double F1(double d)
double F2(int f)
int    F3(double f)
int    F4(int d)
```

   Quais das chamadas da função `UmaFuncao()` a seguir são ilegais e por que?
   (a) `UmaFuncao(2.4, F1);`
   (b) `UmaFuncao(3.5f, F2);`
   (c) `UmaFuncao(2.2, F3(1.5));`
   (d) `UmaFuncao(1.8, &F4);`
   (e) `UmaFuncao(2.8, &F4(2.1));`
   (f) `UmaFuncao(2.4, F4);`
6. Suponha que se tenham os seguintes protótipos:

```
void   OutraFuncao(double (*pf)())

double F1(double d)
double F2(int f)
double F3(float f)
double F4(const char *d)
```

   Quais das chamadas da função `OutraFuncao()` a seguir são ilegais e por que?
   (a) `OutraFuncao(F1);`
   (b) `OutraFuncao(F2);`
   (c) `OutraFuncao(&F3);`
   (d) `OutraFuncao(F4);`

Listas Generalizadas (Seção 11.2)

7. O que é uma lista generalizada?
8. Descreva os seguintes conceitos associados a listas generalizadas:

(a) Átomo
(b) Sublista
(c) Cabeça
(d) Cauda

9. Dada a lista generalizada *A = ((a), (b, (c)))*, determine os resultados das seguintes operações:
(a) *cabeça(A)*
(b) *cauda(A)*
(c) *cabeça(cauda(A))*
(d) *cauda(cauda(A))*
(e) *cabeça(cauda(cauda(A)))*
(f) *cabeça(cabeça(A))*

10. Seja *L* a lista *(a, b, (c, (d), e), f)*. Quais são os resultados das operações seguintes?
(a) *cabeça(cauda(cabeça(cauda(cauda(L)))))*
(b) *cauda(cabeça(cabeça(cauda(cabeça(cauda(cauda(L)))))))*

11. Esta é uma questão conceitual (i.e., matemática), de modo que o símbolo = representa igualdade (e não atribuição). Suponha a existência das funções (matemáticas) seguintes:

(1) *Cab(L)* resulta na lista constituída pela cabeça da lista *L*, se *L* não for vazia

(2) *Cauda(L)* resulta na cauda da lista *L*, se *L* não for vazia

(3) *Une(L1, L2)* resulta na lista que é a concatenação das listas *L1* e *L2*

Se *L* não for uma lista vazia, qual das seguintes expressões será necessariamente verdadeira?
(i) *Cab(Cauda(L)) = Cauda(Cab(L))*
(ii) *Une(Cab(L),Cauda(L)) = L*
(iii) *Cab(Cauda(L)) = L*
(iv) *Cauda(Cab(L)) = L*
(v) *Une(Cab(L),L) = Une(L,Cauda(L))*

12. Para que serve o campo indicador de um nó de uma lista generalizada?

13. Como a profundidade de uma lista generalizada é definida?

14. Por que o uso de recursão é adequado na implementação de operações sobre listas generalizadas?

Estruturas de Dados Genéricas (Seção 11.3)

15. O que é uma estrutura de dados genérica?

16. Quando uma estrutura de dados genérica se faz necessária?

Implementação de Pilha Genérica (Seção 11.4)

17. O que é uma pilha genérica?

18. (a) Para que serve a função **memcpy()**? (b) Como ela funciona?

19. Descreva o uso da função **memcpy()** na implementação de pilha generalizada.

20. (a) Explique por que o uso do operador de conversão explícita **(char *)** é obrigatório na seguinte expressão da função `EmpilhaG()`. (b) O que ocorrerá se esse operador for removido?

```
(char *)p->elementos + p->nElementos*p->tamElemento
```

Usando bsearch() e qsort() (Seção 11.5)

21. Para que servem as funções (a) **qsort()** e (b) **bsearch()** da biblioteca padrão de C?

22. Suponha que você tenha uma função de comparação cujo endereço seja passado como parâmetro para a função **qsort()** quando a ordenação deve ser em ordem crescente. Como você reescreveria essa função para ordenar o array em questão em ordem decrescente?

23. Explique por que os parâmetros da função de comparação utilizada como quarto parâmetro da função **qsort()** são do tipo **const void** *.

24. Como deve ser definida a função de comparação cujo endereço é passado como parâmetro para a função **bsearch()**?

25. O que há de errado com cada uma das seguintes funções de comparação que seriam usadas numa chamada de **qsort()** para ordenar um array de strings constantes?

(a)
```
int ComparaErrado1(const void *p1, const void *p2)
{
   const char *e1 = (const char *)p1;
   const char *e2 = (const char *)p2;

   return *e1 - *e2;
}
```

(b)
```
int ComparaErrado2(const void *p1, const void *p2)
{
   const char *e1 = (const char *)p1;
   const char *e2 = (const char *)p2;
   return strcmp(e1, e2);
}
```

26. Explique o funcionamento da seguinte função de comparação que utilizada para comparar strings:
```
int Compara1(const void *s1, const void *s2)
{
   const char *str1 = *(char **)s1;
   const char *str2 = *(char **)s2;
   return strcmp(str1, str2);
}
```

Conversões e Avaliações de Expressões Aritméticas (Seção 11.6)

27. (a) O que é notação sufixa de uma expressão aritmética? (b) O que é notação prefixa de uma expressão aritmética? (c) O que é notação infixa de uma expressão aritmética?

28. (a) O que é notação polonesa de uma expressão aritmética? (b) Qual é a vantagem obtida com o uso dessa notação com relação à notação infixa?

29. Escreva cada uma das expressões abaixo nas formas prefixa e sufixa:
(a) `A + B - C`
(b) `(A + B)*(C - D)`
(c) `(A + B)*(C - D)%E*F`
(d) `(A + B)*(C%(D - E) + F) - G`
(e) `(A + B)*C%(D - E)/F`
(f) `(A + B)*D + E/(F + A*D) + C`

30. Transforme cada uma das expressões da forma sufixa para a forma infixa:
(a) `ABC+-`
(b) `AB-C+DEF-+%`
(c) `ABCDE-+%*EF*-`
(d) `ABCDE%*/-`

31. Escreva na forma infixa a expressão prefixa: `-+/A%BC*DE*AC`.

32. Mostre como se avalia uma expressão em forma sufixa usando pilha.

Nos exercícios **33** e **34**, suponha que o símbolo `^` represente a operação de exponenciação e que essa operação tenha precedência maior do que as precedências dos demais operadores.

33. Verifique que a forma sufixa de cada uma das expressões infixas na tabela a seguir está correta.

Infixa	Sufixa
`A^B*C - D + E/F/(G + H)`	`AB^C*D-EF/GH+/+`
`A - B/(C*D^E)`	`ABCDE^*/-`

34. Verifique que a forma prefixa de cada uma das expressões infixas na tabela a seguir está correta.

Infixa	Prefixa
`A^B*C - D + E/F/(G + H)`	`+-*^ABCD//EF+GH`
`A - B/(C*D^E)`	`-A/B*C^DE`

Exemplos de Programação (Seção 11.7)

35. (a) Explique o uso da função **memcpy()** na definição da função `BubbleSortGen()` apresentada na **Seção 11.7.1**. (b) Por que ele se faz necessário nessa definição?

11.9 Exercícios de Programação

EP11.1 (a) Modifique a função `BubbleSortG()` apresentada na **Seção 11.7.1** de modo que, ao invés de rearranjar os elementos de um array, ela armazene a ordem correta num novo array denominado `arrayOrdenado[]`. (b) Escreva um programa para testar essa nova versão de `BubbleSortGen()`.

EP11.2 As técnicas ilustradas na construção de pilha genérica podem ser utilizadas para a implementação de tipos de dados genéricos de outras espécies. Construa o tipo `tFilaGen` que represente uma fila genérica, utilizando a metodologia empregada na construção de pilha genérica.

EP11.3 Escreva um programa em C, denominado `penta`, que exibe na tela a formação da seleção brasileira na final da copa do mundo de 2002. Esse programa deve oferecer duas opções ao usuário introduzidas via linha de comando:

(i) `E` ou nenhuma opção: exibe na tela os nomes dos jogadores em ordem de formação (i.e., do goleiro ao ponta-esquerda);

(ii) `A`: exibe na tela os nomes dos jogadores em ordem alfabética crescente

O programa deverá definir um array de strings constantes que representam os nomes dos jogadores e são arranjados seguindo a escalação da seleção em ordem de formação. O programa deverá também utilizar a função **qsort()** da biblioteca padrão de C para ordenar esse array quando necessário. Exemplos de uso do programa:

```
$ penta
Escalacao original: Marcos, Cafu, Edmilson, Lucio, Roque Junior,
Roberto Carlos, Gilberto Silva, Kleberson, Rivaldo, Ronaldinho, Ronaldo

$ penta A
Escalacao ordenada: Cafu, Edmilson, Gilberto Silva, Kleberson, Lucio,
Marcos, Rivaldo, Roberto Carlos, Ronaldinho, Ronaldo, Roque Junior
```

[**Sugestão:** Esse problema não é tão trivial quanto parece. Consulte a **Seção 11.5**, que mostra como uma função de comparação deve ser definida.]

EP11.4 **Preâmbulo:** Uma lista **auto-ajustável** é uma lista na qual sempre que um elemento é acessado por meio de uma operação de busca, ele é movido para a frente da lista. Além disso, o acréscimo de elementos na lista ocorre sempre na frente da lista.

Problema: Suponha que os tipos de nó e de ponteiro para nó de uma lista simplesmente encadeada sejam definidos como:

```
typedef struct no {
           tDados     dado;
           struct no *proximo;
        } tNo, *tLista;
```

Nessa definição de tipo sabe-se apenas que `tDados` é um tipo previamente definido. Escreva uma função que efetua uma operação de busca numa lista encadeada considerando-a auto-ajustável. Isto é, caso um elemento da lista seja encontrado durante essa operação, ele deve ser movido para a frente da lista. O protótipo dessa função deve ser:

```
tLista BuscaAutoAjuste(tLista *lista, const tDados *procurase,
                       int (*Compara(const tDados *d1, const tDados *d1)))
```

Nesse protótipo, `Compara` é um ponteiro para uma função que compara as chaves de dois valores do tipo `tDados` e retorna `0`, quando as chaves são iguais, um valor negativo quando a chave do primeiro valor é menor do que a chave do segundo valor e um valor positivo em caso contrário. A função solicitada deve retornar um ponteiro para o nó contendo a chave procurada, que deve ser o primeiro nó da lista devido ao auto-ajuste. Se a chave não for encontrada, essa função deve retornar **NULL**.

EP11.5 Suponha que um processador hipotético possua um único registrador e seis instruções definidas como segue:

- `LD A` — carrega o conteúdo da variável `A` no registrador
- `ST A` — armazena o conteúdo do registrador na variável `A`
- `AD A` — adiciona o conteúdo do registrador com a variável `A`
- `SB A` — subtrai o conteúdo da variável `A` do registrador
- `ML A` — multiplica o conteúdo do registrador pela variável `A`
- `DV A` — divide o conteúdo do registrador pela variável `A`

Escreva uma função em C que receba como entrada uma expressão escrita na forma sufixa contendo operandos representados por letras simples e os operadores `+`, `-`, `*` e `/` e exiba na tela uma sequência de instruções na linguagem descrita acima para avaliar a expressão, deixando o resultado no registrador. Utilize variáveis da forma `TEMPn` como variáveis temporárias. Por exemplo, a expressão sufixa `ABC*+DE-/` deve resultar em:

```
LD B
ML C
ST TEMP1
LD A
AD TEMP1
ST TEMP2
LD D
SB E
ST TEMP3
LD TEMP2
DV TEMP3
ST TEMP4
```

EP11.6 Utilizando como modelo a implementação de pilha genérica apresentada na **Seção 11.4**, implemente uma lista simplesmente encadeada genérica.

EP11.7 Utilizando como modelo a implementação de pilha genérica apresentada na **Seção 11.4**, implemente uma fila genérica.

EP11.8 Modifique as funções apresentadas na **Seção 11.7.2** de modo que os operandos de uma expressão possam ser números reais.

EP11.9 Desenvolva um algoritmo para conversão de expressões infixas em prefixas e implemente esse algoritmo.

EP11.10 Desenvolva um algoritmo para avaliação de expressões prefixas e implemente esse algoritmo.