CAPÍTULO 3 FUNÇÕES

3.1 Introdução

Talvez a abordagem geral mais utilizada na construção de programas de pequeno porte a partir da descrição de um problema seja o método de **refinamentos sucessivos** (também conhecido como método *top-down*). Utilizando este método, a descrição geral do problema é dividida em passos (i.e., subproblemas menos complexos do que o problema original). Então, cada passo é subdividido sucessivamente em subpassos até que cada um deles seja resolvido por operações elementares da linguagem a ser utilizada na construção do programa ou existam rotinas que agrupem instruções que realizem a tarefa. Em C, funções têm esta última finalidade.

Este capítulo basicamente explora a definição e uso de funções. O capítulo começa introduzindo os conceitos de endereços e ponteiros. Esses conceitos são essenciais para entender como se pode simular passagem de parâmetros por referência em C, que, a rigor, possui apenas passagem de parâmetros por valor. Para um bom acompanhamento do objeto deste capítulo, é importante que a seção sobre endereços e ponteiros seja bem compreendida. Portanto, leia e releia esta seção e convença-se de que realmente entendeu todos os conceitos e exemplos contidos aqui antes de explorar os demais assuntos.

Do ponto de vista prático, são apresentados dois tópicos essenciais para a construção de programas interativos amigáveis: leitura e validação

de dados e interação dirigida por menus. Funções recursivas, ***inline*** e com listas de argumentos variáveis também são apresentadas neste capítulo.

3.2 Endereços e Ponteiros

3.2.1 Endereços

Qualquer variável definida num programa em C possui um **endereço** em memória que indica o local onde a mesma é armazenada em memória. Freqüentemente, é necessário utilizar o endereço de uma variável num programa, em vez de seu próprio valor. Por exemplo, na função **scanf()**, apresentada no **Capítulo 2**, utilizam-se endereços de variáveis como argumentos. O endereço de uma variável pode ser determinado por meio do uso do operador de endereço **&**. Suponha, por exemplo, a existência da seguinte declaração de variável:

```
int  x;
```

Então, a expressão:

```
&x
```

resulta no endereço atribuído, em memória, à variável `x` quando o programa é carregado.

Deve-se observar que não é permitido modificar o endereço de uma variável por meio de uma atribuição. Isto é, a instrução a seguir, por exemplo, é ilegal em C:

```
&x = 5000;     /* ILEGAL */
```

3.2.2 Ponteiros

Ponteiro (ou **apontador**) é um tipo especial de variável capaz de conter um endereço em memória. Um ponteiro que contém um endereço em memória válido é dito ***apontar*** para tal endereço. Um ponteiro pode apontar para um valor de qualquer tipo armazenado em memória (por exemplo, **int**, **float** ou outros tipos mais complexos). Assim, pode-se ter ponteiros para **int**, ponteiros para **float** etc.

Uma definição de ponteiro em C tem o seguinte formato:

tipo-apontado ***variável-do-tipo-ponteiro;***

Por exemplo:

```
int  *ponteiroParaInteiro;
```

declara a variável `ponteiroParaInteiro` como sendo um ponteiro capaz de apontar para uma posição de memória que contenha uma variável do tipo **int**.

Variáveis do tipo ponteiro podem ser iniciadas da mesma forma que outros tipos de variáveis. Por exemplo, a segunda definição a seguir:

```
int    meuInteiro;
int   *ponteiroParaInteiro = &meuInteiro;
```

define a variável `ponteiroParaInteiro` como sendo um ponteiro para o tipo **int** e inicia o valor deste ponteiro com o endereço da variável `meuInteiro`. No caso de iniciação de um ponteiro com o endereço de uma variável, como no último exemplo, a variável deve já ter sido declarada. Por exemplo, inverter a ordem das declarações do exemplo anterior acarreta em erro de compilação.

Endereços e ponteiros podem ser impressos no meio da saída padrão utilizando-se a função **printf()** com o especificador de formato **%p**. Algumas vezes, a impressão de endereços é útil durante a depuração de um programa.

3.2.3 Indireção de Ponteiros

Pode-se acessar o conteúdo da porção de memória apontada por uma variável do tipo ponteiro por meio da **indireção** do ponteiro. Isto é, a indireção de um ponteiro resulta no valor contido na posição de memória para onde o ponteiro aponta. Para obter este valor, precede-se o mesmo com o **operador de indireção ***, que é o mesmo símbolo utilizado na declaração do ponteiro. Por exemplo, dadas as definições a seguir:

```
float  meuFloat = 3.14;
float  *ponteiroParaFloat = &meuFloat;
```

o valor de `*ponteiroParaFloat` é `3.14`, enquanto o valor de `ponteiroParaFloat` é o endereço atribuído à variável `meuFloat`. Ainda considerando o último exemplo, note que o valor de `*ponteiroParaFloat` neste ponto do programa será sempre `3.14`, mas o valor de `ponteiroParaFloat` (sem indireção) poderá variar entre uma execução e outra do programa, de computador para computador, etc.

Pode-se atribuir um valor ao conteúdo da posição de memória apontada por um ponteiro utilizando o operador de indireção. Por exemplo, considerando as definições do exemplo anterior, a instrução:

```
*ponteiroParaFloat = 1.6;
```

atribuiria o valor `1.6` ao conteúdo da posição de memória apontada por `ponteiroParaFloat`. É interessante notar que isso é equivalente a modificar o valor da variável `meuFloat` sem fazer referência direta à mesma. Isto é, a última instrução é equivalente à instrução:

```
meuFloat = 1.6;
```

O fato de C utilizar o mesmo símbolo * para declaração e indireção de ponteiros pode causar alguma confusão para o programador iniciante. Por exemplo, você pode ficar intrigado com o fato de, na iniciação:

```
float  *ponteiroParaFloat = &meuFloat;
```

`*ponteiroParaFloat` receber o valor de um endereço, enquanto na instrução:

```
*ponteiroParaFloat = 1.6;
```

`*ponteiroParaFloat` recebe o valor de um **float**. Entretanto, esta interpretação é errônea, pois o asterisco na declaração de `ponteiroParaFloat` **não** representa o operador de indireção. Portanto, na iniciação, o endereço é atribuído ao ponteiro `ponteiroParaFloat` e não à expressão `*ponteiroParaFloat`. Apenas na instrução de atribuição acima, o operador de indireção é utilizado.

Os operadores de indireção ***** e de endereço **&** fazem parte do mesmo grupo de precedência em que estão todos os outros operadores unários de C. A precedência destes operadores é a mais alta dentre todos os operadores vistos até aqui e a associatividade deles é da direita para a esquerda.

3.2.4 O Ponteiro Nulo

Um **ponteiro nulo** é um ponteiro que não aponta para nenhum endereço válido e, mais importante, esse fato é reconhecido pelo sistema. Um ponteiro torna-se nulo quando recebe o valor inteiro 0. Por exemplo,

```
char  *p;

p = 0;  /* Torna p um ponteiro nulo */
```

A seguinte macro pode tornar expressões de atribuição e comparação envolvendo ponteiros nulos mais legíveis:

```
#define  NULL  0
```

De acordo com o padrão ISO corrente, 0 é um valor inteiro compatível com qualquer ponteiro. Entretanto, alguns compiladores mais antigos requerem que a constante **NULL** seja definida como[1]:

```
#define NULL ((void *)0)
```

para que este valor seja compatível com qualquer tipo de ponteiro. De qualquer modo, **NULL** é definida no arquivo de cabeçalho `<stdlib.h>` e o programador não precisa se preocupar em defini-la.

É importante salientar que, quando um ponteiro recebe o valor 0, ele não está recebendo um endereço com este valor, mas sim um valor que depende de implementação. Mais importante ainda é o fato de o sistema operacional ***sempre*** reconhecer este endereço como inválido e causar o

[1] O tipo **void *** utilizado na operação de conversão representa ponteiros genéricos; i.e., ponteiros compatíveis com qualquer tipo de ponteiro (v. **Seção 11.4**).

aborto do programa quando se tenta acessar o conteúdo da posição apontada por este ponteiro.

Parece estranho à primeira vista que se atribua a um ponteiro um valor que causará o aborto do programa. Acontece que, quando um ponteiro assume um valor inválido que não seja o ponteiro nulo e se tenta acidentalmente acessar o conteúdo apontado, as conseqüências podem ser desastrosas e o erro muito mais difícil de detectar, pois o sistema operacional nem sempre é capaz de apontar a invalidade do ponteiro. Assim, na prática, o ponteiro nulo é utilizado para indicar que, no ponto do programa onde o ponteiro é nulo, não existe nenhum valor válido para lhe atribuir.

3.2.5 Compatibilidade e Conversões entre Ponteiros

Apesar de qualquer endereço num dado computador *poder* ser representado da mesma maneira (por exemplo, como um inteiro de 32 bits), não existe, em princípio, compatibilidade entre ponteiros para tipos diferentes. Isto é, mesmo que o padrão de C permita conversão entre estes ponteiros, ela pode causar problemas durante a execução do programa, mesmo quando se utiliza conversão explícita. O padrão de C requer apenas que o compilador emita uma mensagem de advertência quando for feita uma conversão entre ponteiros de tipos diferentes sem o uso de conversão explícita. Por exemplo, dadas as definições de variáveis:

```
float  x;
int    *p;
```

a seguinte conversão implícita de **float *** para **int *** é permitida, apesar de o compilador apresentar uma mensagem de advertência:

```
p = &x; /* Permitido, mas o compilador emite uma advertência */
```

Com o uso de conversão explícita, o compilador não é obrigado a emitir mensagem de advertência:

```
p = (int*) &x; /* Permitido sem advertência, mas perigoso */
```

O fato de o compilador não emitir nenhuma mensagem de advertência no último caso, não garante que não ocorrerá nenhum problema quando o programa for executado.

Não é portável considerar que ponteiros e inteiros têm o mesmo tamanho. Mesmo assim, um inteiro pode ser convertido em qualquer tipo de ponteiro e vice-versa. Em ambos os casos, o resultado é dependente de implementação e, portanto, deve ser evitado. O único inteiro que pode ser convertido em ponteiro é 0, conforme já foi visto na **Seção 3.2.4**. Por exemplo, as instruções a seguir são perfeitamente legais, apesar de não serem recomendáveis:

```
float *p;
int    x;
...
p = 12; /*OK, mas o resultado é dependente de implementação /*
x = p;  /* Idem */
```

Nos dois casos de conversão acima, o compilador emite mensagens de advertência, mas isto não impede o programa de ser compilador. O uso de *casting* pode fazer com que o compilador deixe de apresentar advertências, mas não resolve o problema que poderá ocorrer quando o programa for executado:

```
short  s;
long  *ptr;

ptr = (long  *) s;  /* OK, mas pode causar problemas */
```

Nas conversões entre ponteiros de tipos diferentes e entre ponteiros e inteiros, o padrão de C requer apenas diagnósticos (i.e., mensagens de advertência) por parte do compilador, mas elas não são proibidas. Por outro lado, conversões entre ponteiros e outros tipos primitivos não-inteiros não são permitidas. Por exemplo:

```
float x;
int  *p = x; /* ILEGAL: não compila */
```

3.3 Funções

Função em C é o nome genérico dado a subprograma, rotina ou procedimento em outras linguagens de programação. Pode-se definir uma função como sendo um conjunto de operações que executam ***uma tarefa específica*** que é, usualmente, mais complexa do que qualquer operação elementar da linguagem C. Uma função que executa tarefas múltiplas e distintas não é normalmente uma função bem projetada. Também, mesmo que realize um único objetivo, se uma função é tão complexa que seu entendimento torna-se difícil, ela deve ser subdividida em funções menores e mais fáceis de serem entendidas.

Uma função pode ainda ser vista como uma abreviação para um conjunto de instruções. Se este conjunto de instruções aparece mais de uma vez num programa, ele precisa ser definido apenas uma vez dentro de um programa, mas pode ser invocado nos vários pontos do programa em que essas instruções sejam necessárias. Outros benefícios obtidos com o uso de funções num programa são:

- **Facilidade de manutenção**. Quando várias instruções que aparecem repetidamente num programa estão confinadas dentro de uma função, a modificação deste conjunto de instruções, quando necessária, precisa ser efetuada apenas uma vez.

- **Melhora de legibilidade**. Mesmo que um conjunto de instruções apareça apenas uma vez num programa, muitas vezes é preferível manter estas instruções confinadas numa função e substituir sua ocorrência pela chamada da função. Assim, além de melhorar a legibilidade do programa, pode-se ter uma visão geral do programa no nível de detalhes desejado[2].

[2] Por exemplo, alguém que esteja lendo o programa pode estar interessado nos detalhes de funcionamento de uma dada função, mas outros podem estar interessados em ler um programa num nível de abstração mais elevado e saber que a função existe é suficiente

Funções podem aparecer de três maneiras diferentes num programa:

- Em forma de **definição**, que especifica aquilo que a função realiza, bem como o número e os tipos dos dados (**argumentos**) utilizados pela função.
- Em forma de **chamadas**, que causam a execução da função.
- Em forma de **alusões**, que contêm parte da definição da função e servem para informar o compilador que a função aludida é definida em outro local (muitas vezes, num outro arquivo).

Estas formas de referências a funções serão examinadas a seguir.

3.3.1 Definições de Funções

Uma **definição** de função é dividida em duas partes: (1) **cabeçalho** e (2) **corpo da função.**

▶ Cabeçalho de Função

O cabeçalho de uma função informa o **tipo do valor retornado** pela função, o **nome** da função e os **tipos dos argumentos** (ou **parâmetros**) utilizados pela função.

Existem dois formatos permitidos para a escrita do cabeçalho de uma função:

Formato 1:

tipo-de-retorno nome-da-função (argumentos) declaração-de-argumentos

Formato 2:

tipo-de-retorno nome-da-função (declaração-de-argumentos)

Note que a diferença entre os dois formatos está apenas na forma como os argumentos são declarados. O primeiro formato é obsoleto, mas deve ser aceito por qualquer compilador que utilize o padrão ISO.

O segundo formato é bem mais elegante, mas não é aceito por compiladores de C muito antigos. Aqui, apenas o formato 2 será utilizado, mas se o compilador que você estiver utilizando não o aceitar, você não terá dificuldades em adotar o formato 1.

Em padrões de C anteriores a C99, não era obrigatório incluir o tipo de retorno da função e, quando o mesmo não era incluído, o tipo assumido pelo compilador era **int**. No padrão C99, é obrigatória a indicação do tipo de retorno de uma função. Por exemplo, a definição de função:

```
F(int x)
{
   ...
}
```

que antes um compilador padrão assumia ter tipo de retorno **int**, agora é considerada ilegal em C99.

Quando não se deseja que a função retorne nenhum valor, utiliza-se o tipo primitivo **void** como tipo de retorno. Quando o tipo de retorno de uma função é **void**, ela não pode ser chamada dentro de uma expressão ou atribuição. O tipo de retorno de uma função pode ser qualquer tipo não-estruturado[3].

A declaração de argumentos no cabeçalho da função é similar a um conjunto de declarações de variáveis. No segundo formato de cabeçalho apresentado acima, cada argumento é declarado individualmente com seu respectivo tipo e as declarações são separadas por vírgulas, enquanto no primeiro formato argumentos de um mesmo tipo podem ser colocados juntos, precedidos pelo tipo comum e as declarações são separadas por ponto-e-vírgulas. Iniciações não são permitidas em nenhum dos dois formatos. Quando a função não possui argumentos, pode-se deixar vazio o espaço entre parênteses ou escrever a palavra-chave **void** entre os parênteses; esta segunda opção é mais recomendada, pois torna a declaração mais legível.

[3] Alguns tipos estruturados, a serem vistos posteriormente, podem ser utilizados como tipos de retorno.

Exemplos de cabeçalhos de função:

Formato 1:

```
void  f(a, b, c, d)
int   a, b;
char  c;
float *d;
```

Formato 2:

```
void  f(int  a, int  b, char  c, float  *d)
```

Note que, no formato 1, as declarações de argumentos não precisam estar em linhas diferentes, mas isso melhora a legibilidade.

Em termos de estilo e legibilidade, o nome de uma função deve ser representativo da tarefa ou operação que a função executa. É recomendável que cada palavra constituinte do nome da função comece por letra maiúscula, pois assim não é necessário usar sublinha (_) para separar as palavras. Não se guie pelos nomes de funções biblioteca de C, pois a maioria deles representa péssimo estilo de nomenclatura. Por exemplo, você faz alguma idéia daquilo que a função de biblioteca **strpbrk()** realiza? Também, por questões de portabilidade, evite o uso de identificadores que utilizem caracteres especiais de português (por exemplo, *ç*, *é*, *ã*), pois a maioria dos compiladores de C não os aceitam[4].

▶ Corpo de Função

O corpo de uma função contém declarações e instruções necessárias para implementar aquilo que a função deve realizar quando a mesma for executada. O corpo de uma função deve ser envolvido por chaves. É importante chamar a atenção para o fato de que variáveis declaradas dentro do corpo de uma função não são reconhecidas fora do mesmo**.** Isto

[4] O uso de caracteres universais (v. **Seção 8.7**) na construção de identificadores é previsto pelo padrão C99. Entretanto, o modo como caracteres UCN são implementados não permite que se construam com eles identificadores legíveis.

é, qualquer variável declarada dentro de uma função tem validade apenas dentro da função[5].

O corpo de uma função pode ser vazio e este fato é bastante utilizado durante a fase de desenvolvimento de um programa quando se deseja adiar a implementação da função. Nesta situação, o corpo vazio representa um guardador de lugar que será preenchido oportunamente com a implementação completa da função. Para evitar confusão ou esquecimento, é sempre bom indicar, por meio de comentários, quando o corpo de uma função é intencionalmente vazio. Por exemplo:

```
int  MinhaFuncao(short  s, long  *i)
{
   /* Corpo vazio a ser preenchido posteriormente. */
}
```

3.3.2 Chamadas de Funções

Chamar (ou **invocar**) uma função é transferir o controle do programa para a função a fim de executá-la. Uma chamada de função pode aparecer sozinha numa linha de instrução quando não existe valor de retorno (i.e., quando o tipo de retorno é **void**) ou quando este existe mas não há interesse em utilizá-lo. Se o tipo de retorno é diferente de **void** e deseja-se ignorá-lo, aconselha-se utilizar uma conversão explícita (*casting*) para **void** para tornar este fato explícito. Por exemplo,

```
(void) printf("Retorna o número de caracteres impressos");
```

Chamadas de funções, cujos tipos de retorno são diferentes de **void**, podem também ser utilizadas como parte de expressões. Numa tal situação, o valor resultante da chamada da função irá substituir a própria chamada da função na expressão. Por exemplo, suponha que uma função

[5] Mais geralmente, o corpo de uma função é um **bloco** e objetos declarados dentro de um bloco têm **escopo local** ao mesmo. Estes conceitos serão explorados em profundidade no **Capítulo 4**.

`f()` retorne `1` como resultado da execução de uma chamada; então, após a execução da chamada `f()` na expressão:

```
x = f()/3 + 5;
```

o valor `1` substituiria a chamada `f()` e a expressão tornar-se-ia:

```
x = 1/3 + 5;
```

Os parênteses utilizados numa chamada de função representam um operador (v. **Seção 10.2**) e o grupo de operadores do qual este operador faz parte tem a maior precedência dentre todos os operadores da linguagem C. Isto significa que, quando uma chamada de função aparece numa expressão, ela sempre é avaliada (i.e., chamada) antes da aplicação de qualquer outro operador que não faça parte desse grupo.

3.3.3 Passagem de Parâmetros

Argumentos ou **parâmetros** provêem o meio normal de comunicação entre funções em C (e em outras linguagens de programação). Normalmente, não se deve utilizar variáveis e constantes globais para fazer esta comunicação, a não ser que haja realmente um bom motivo para assim proceder. Além disso, se uma variável é necessária apenas dentro de uma função, ela deve ser uma variável local à função e, portanto, declarada dentro do corpo da mesma.

O **modo** de um parâmetro passado para uma função refere-se à forma como o mesmo é utilizado dentro da função. Existem três modos de parâmetros:

- **Parâmetro de entrada**. A função apenas usa (i.e., consulta) o valor do parâmetro, mas não o altera.
- **Parâmetro de saída**. A função apenas altera o valor do parâmetro, mas não utiliza este valor.
- **Parâmetro de entrada e saída**. Um parâmetro deste modo é uma combinação dos outros modos anteriores. Isto é, a função tanto consulta o valor do parâmetro quanto o altera.

Os argumentos que aparecem numa declaração de função são denominados **parâmetros formais**, enquanto argumentos utilizados numa chamada de função são denominados **parâmetros reais**. Numa chamada de função ocorre uma **ligação** (ou **casamento**) entre parâmetros reais e formais. Cada linguagem de programação estabelece **regras de casamento** (ou de ligação) entre parâmetros reais e formais.

Em C, duas regras de casamento devem ser seguidas para que uma ligação de parâmetros seja bem-sucedida:

(1) O número de parâmetros formais deve ser igual ao número de parâmetros reais. Isto é, se não há parâmetros formais na declaração da função, também não deve haver parâmetros reais na chamada; se houver três parâmetros formais na definição da função, deve haver três parâmetros reais na chamada e assim por diante.

(2) Os respectivos parâmetros devem ser compatíveis. Isto é, o primeiro parâmetro real deve ser compatível com o primeiro parâmetro formal, o segundo parâmetro real deve ser compatível com o segundo parâmetro formal e assim por diante.

Em algumas linguagens de programação, existem dois tipos de passagem de parâmetros:

(1) Passagem por valor. Quando um parâmetro é passado por valor, o parâmetro formal recebe uma cópia do parâmetro real correspondente. Qualquer modificação sofrida pelo parâmetro formal no interior da função fica restrita a esta cópia e, portanto, não é comunicada ao parâmetro real.

(2) Passagem por referência. Na passagem por referência, o parâmetro formal e o parâmetro real correspondente, que deve ser uma variável, compartilham a mesma posição de memória, de modo que qualquer alteração feita no parâmetro formal pela função corresponde à mesma alteração no parâmetro real correspondente.

A passagem de parâmetro por referência é obrigatória em algumas linguagens para parâmetros de saída ou de entrada/saída, porque qualquer

modificação deve ser comunicada ao exterior da função. Por outro lado, parâmetros de entrada podem ser passados por valor ou por referência; neste último caso, por uma questão de eficiência.

Em C, estritamente falando, existe apenas passagem de parâmetros *por valor*, de modo que nenhum parâmetro real tem seu valor modificado como conseqüência da execução de uma função. Este fato pode parecer estranho à primeira vista e uma questão que surge naturalmente é: ***Como é, então, que uma variável, utilizada como parâmetro real numa chamada de função, pode ter seu valor modificado em conseqüência da execução da função?***

A resposta à questão: mediante a utilização de ponteiros e endereços de variáveis é possível simular passagem por referência em C e modificar o valor de variáveis. Considere, como exemplo, a função a seguir, que tem por finalidade trocar os valores de duas variáveis do tipo **int**:

```
void  Troca(int  *ptrParaInt1, int *ptrParaInt2)
{
   int  aux = *ptrParaInt1; /* A variável aux é local à função */

   *ptrParaInt1 = *ptrParaInt2;
   *ptrParaInt2 = aux;
}
```

Então, dadas as declarações de variáveis a seguir:

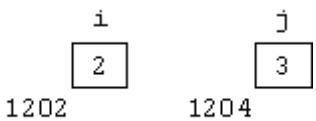
```
int  i = 2, j = 3;
```

a função `Troca()` poderia ser chamada como:

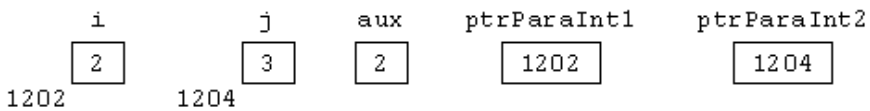
```
Troca(&i, &j);
```

e o efeito após a chamada seria a troca de valores entre `i` e `j` (i.e., `i` passaria a conter 3 e `j` passaria a conter 2). Note que, apesar de `i` e `j` terem seus valores modificados, os parâmetros reais (`&i` e `&j`) mantêm seus valores inalterados, como seria esperado numa passagem de parâmetros por valor. Para entender o funcionamento da chamada de função `Troca()` acima, acompanhe a seqüência de eventos descrita a seguir, que ocorre por trás da chamada `Troca(&i, &j)`.

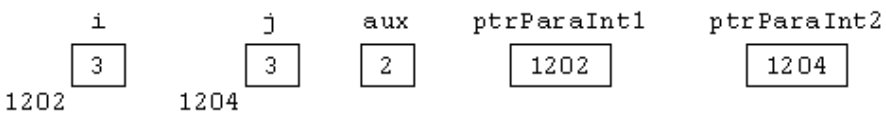
Como ocorre com qualquer variável declarada num programa em C, as variáveis `i` e `j` têm alocadas posições em memória com endereços únicos. Para facilitar o entendimento, suponha que às variáveis `i` e `j` sejam atribuídas posições de memória com os endereços `1202` e `1204`. Então, após a iniciação no trecho de programa acima, poder-se-ia representar a situação por meio do seguinte esquema:



Quando ocorre a chamada da função, os parâmetros passados são os endereços de `i` e `j`; ou seja `1202` e `1204`, respectivamente. Portanto, os parâmetros formais `ptrParaInt1` e `ptrParaInt2` recebem uma cópia dos endereços `1202` e `1204`, respectivamente. Na primeira instrução da função `Troca()`, que é a iniciação da variável local `aux`, esta variável recebe o valor do conteúdo do endereço apontado por `ptrParaInt1`, que é `2`, de modo que, neste ponto, tem-se a seguinte situação esquemática[6]:

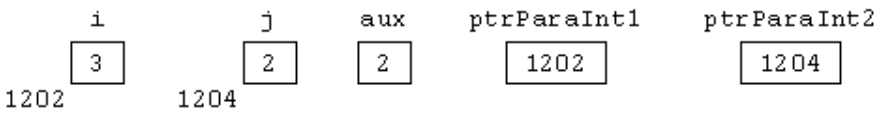


Na segunda instrução da função, `*ptrParaInt1 = *ptrParaInt2`, o conteúdo do endereço apontado por `ptrParaInt1` é substituído pelo conteúdo do endereço apontado por `ptrParaInt2`, de modo que, agora, obtém-se a seguinte configuração:



[6] Os endereços da variável local `aux` e dos parâmetros `ptrParaInt1` e `ptrParaInt2` não são de interesse aqui.

Finalmente, na terceira e última instrução, o conteúdo do endereço apontado por `ptrParaInt2` é substituído pelo valor da variável `aux`, resultando no seguinte esquema:



A variável `aux` deixa de existir ao final da execução da função, o que é realmente desejável, visto que sua finalidade é apenas auxiliar o processamento local da função `Troca()`.

A função `Troca()` do último exemplo espera como parâmetros dois endereços para valores do tipo **int**. Pode-se, portanto, passar tanto endereços de variáveis do tipo **int**, como foi feito acima, quanto ponteiros para o tipo **int**. Por exemplo, a chamada da função `Troca()` a seguir:

```
int main()
{
   int     i = 2, j = 3;
   int    *ponteiroParaInt = &i;
   ...
   Troca(ponteiroParaInt, &j);
   ...
}
```

teria exatamente o mesmo efeito da chamada do exemplo anterior. O problema na utilização de um ponteiro em chamadas de funções ocorre quando ele não foi iniciado com um endereço válido. Por exemplo, suponha que na chamada `Troca(ponteiroParaInt, &j)` do último exemplo o ponteiro `ponteiroParaInt` não tivesse sido iniciado com o valor do endereço da variável `i`. Como qualquer variável definida no interior de uma função que não tem um valor explicitamente atribuído, `ponteiroParaInt` contém um valor indeterminado que, neste caso, é um suposto endereço qualquer em memória, que pode, inclusive, nem existir num dado computador. Os resultados de uma operação deste tipo são imprevisíveis e desastrosos, pois se estará modificando aleatoriamente o conteúdo da memória do computador.

O problema descrito no parágrafo anterior aflige não apenas programadores inexperientes como também aqueles mais experientes (mas descuidados). Para evitar este tipo de problema, discipline-se: sempre que definir um ponteiro, inicie-o com um valor conhecido. Se não houver nenhum valor válido para ser atribuído a um ponteiro no instante de sua definição, inicie-o com o valor constante **NULL**, que é reconhecido como um endereço inválido. Desta maneira, se você esquecer de atribuir um endereço válido a um ponteiro e tentar acessar o conteúdo deste endereço, o computador indicará uma operação inválida e impedirá que a execução do programa prossiga. Pode até parecer que isto não seja uma boa idéia, mas pior seria permitir que o programa prosseguisse e causasse o mau funcionamento de seu programa, de outros programas, ou até mesmo do próprio computador. Para utilizar a constante **NULL**, você deve incluir em seu programa o arquivo de cabeçalho `<stddef.h>` por meio da diretiva **#include**.

Conforme foi afirmado na **Seção 3.3.2**, os parênteses utilizados numa chamada de função representam um operador. Como a maioria dos operadores de C, este operador não possui ordem de avaliação de operandos definida. Isto significa que a ordem de avaliação de argumentos reais numa chamada de função é dependente de implementação. Por exemplo, suponha que uma função possua o seguinte cabeçalho:

```
void F(int a, int b)
```

Então, os valores que receberão os parâmetros `a` e `b` da função no seguinte trecho de programa:

```
int x = 0;
...
F(x, ++x);
```

são dependentes de implementação. Isto é, numa dada implementação, `a` e `b` podem receber, respectivamente, 0 e 1, enquanto em outra implementação eles podem receber, respectivamente, 1 e 1.

3.3.4 Instrução return

Cada função que não tem **void** como tipo de retorno pode retornar no máximo um valor de um tipo compatível com o tipo de retorno declarado em seu cabeçalho. Para esta finalidade utiliza-se a instrução **return** seguida do valor a ser retornado, que pode ser uma constante, variável ou expressão. A sintaxe mais geral de uma instrução **return** é:

return *expressão*;

O efeito da instrução **return** dentro de uma função é causar o final da execução da função com o conseqüente retorno, para o ponto de chamada da função, do valor da expressão que a acompanha. Quando esta expressão é complexa, recomenda-se o uso de parênteses em torno da mesma para uma melhor legibilidade, mas o uso de parênteses não é obrigatório e deve ser seguido apenas nestes casos.

De acordo com o padrão C99, funções com tipo de retorno **void** não podem retornar nenhum valor. Isto é, quando o tipo de retorno da função é **void**, uma instrução **return** não pode conter nenhuma expressão. Neste caso, a função será encerrada quando tal instrução for executada ou quando o final da função for atingido sem que nenhum valor válido seja retornado. O padrão C99 especifica ainda que funções com tipo de retorno diferente de **void** devem retornar algum valor compatível com o tipo declarado. Em versões anteriores do padrão de C, ambos os casos resultavam em comportamento indefinido do programa.

Pode haver mais de uma instrução **return** no corpo de uma função, mas isto não quer dizer que mais de um valor pode ser retornado a cada execução da função. Normalmente, quando uma função possui mais de uma instrução **return**, cada uma delas acompanhada de uma expressão diferente, valores diferentes poderão ser retornados em chamadas diferentes da função, dependendo dos valores dos argumentos recebidos pela função. A primeira instrução **return** executada causará a parada de execução da função e o retorno do respectivo valor associado à instrução. Por exemplo,

```
unsigned  char  MinhaFuncao2( long   x )
{
    if ( x < 0)
        return  0;
    else
        return  1;
}
```

No exemplo acima, a função retornará `0` quando o valor de seu argumento no instante da chamada for negativo; caso contrário, ela retornará `1`.

3.3.5 Conversões Automáticas em Chamada e Retorno de Funções

Como foi afirmado na **Seção 3.3.3**, uma das regras de ligação de parâmetros dita que parâmetros reais e formais correspondentes devem ser compatíveis. Em casos em que o tipo de um parâmetro formal e o tipo do parâmetro real correspondente diferem mas uma conversão de tipos é possível, haverá uma conversão implícita do tipo do parâmetro real para o tipo do parâmetro formal. Por exemplo, se uma função é declarada como:

```
void F(int  a, long  b, double  c)
{
    /* Corpo da função */
}
```

a chamada a seguir será legal:

```
long   x;
float  y;
short  z;

F(x, y, z);
```

Nesta chamada, o valor da variável `x` seria convertido para **int**, o valor de `y` seria convertido para **long** e o valor de `z` seria convertido para **double**. Lembre-se de que as regras para compatibilidade entre ponteiros são mais rígidas do que aquelas relativas aos tipos aritméticos. Por exemplo, dada a seguinte definição de função:

```
void G(int *ptrParaInt)
{
    /* Corpo da função */
}
```

a chamada a seguir seria *ilegal*:

```
float  f;

G(f);
```

Para prevenir-se contra surpresas desagradáveis como resultado de uma chamada de uma função, é importante passar parâmetros reais dos mesmos tipos dos parâmetros formais correspondentes ou então certificar-se de que uma dada conversão não produzirá um efeito indesejável.

O tipo resultante da avaliação da expressão de retorno deve ser compatível com o tipo de retorno declarado no cabeçalho da função. Quando o tipo de retorno declarado não coincide com o tipo resultante da expressão e é possível uma conversão entre estes tipos, o valor retornado é implicitamente convertido para o tipo de retorno declarado. Por exemplo, na função a seguir:

```
long  MinhaFuncao3( void )
{
   return  2*3.14
}
```

o resultado da expressão `2*3.14`, que é do tipo **double**, será implicitamente convertido para **long**, que é o tipo de retorno declarado.

3.3.6 Protótipos e Alusões de Funções

Uma **alusão** a uma função contém informações sobre a função que permitem ao compilador reconhecer uma chamada da função como sendo legal. Freqüentemente, a função aludida é definida num arquivo outro que não aquele onde é feita a alusão. A forma de alusão de uma função é muito parecida com o cabeçalho da própria função, exceto que numa alusão não é necessário especificar nomes de argumentos (embora isto seja recomendável) e pode-se, ainda, iniciar a alusão com a palavra-chave

extern (também recomendável). Portanto, uma alusão deve ter o seguinte formato:

extern *tipo-de-retorno nome-da-função*(*tipos-dos-argumentos*);

Por exemplo, a função `Troca()` apresentada na **Seção 3.3.3** poderia ter a seguinte alusão:

```
extern void Troca(int *, int *);
```

A sentença seguindo a palavra-chave **extern** numa alusão de função é conhecida como **protótipo** da função e podem-se incluir nomes de argumentos para torná-lo mais legível, como, por exemplo:

```
void Troca(int *prtInt1, int  *ptrInt2);
```

Os nomes de argumentos incluídos na última declaração têm como único objetivo tornar a alusão mais clara; eles não precisam coincidir com os nomes dos argumentos formais na definição da função. A palavra-chave **extern** é opcional, mas seu uso também é recomendável, pois torna mais claro que se está fazendo uma alusão.

Para compiladores que não aceitam o formato 2 de cabeçalho apresentado na **Seção 3.3.1**, uma alusão deve conter apenas o tipo de retorno da função, seguido do nome da função, e seguido por abre e fecha parênteses. Este formato de alusão também pode ser utilizado com compiladores que aceitam o uso de protótipos. Entretanto, neste caso, ele não é recomendável, pois não permite que o compilador cheque se uma dada chamada da respectiva função satisfaz as regras de casamento de parâmetros. Por exemplo, pode-se aludir à função `Troca()` da **Seção 3.3.3** como:

```
extern void Troca();
```

Neste caso, o compilador não teria como constatar que a chamada:

```
Troca(5);
```

seria ilegal e a tarefa de verificação ficaria a cargo do *linker*. Acontece, porém, que encontrar a origem de erros detectados pelo *linker* é mais difícil do que ocorre com erros apontados pelo compilador.

3.4 Funções com Listas de Argumentos Variáveis

Algumas funções, como, por exemplo, **printf()** e **scanf()**, possuem argumentos cujos número e tipos não são especificados a priori. Isto é, o número e os tipos destes argumentos podem variar entre uma chamada e outra da função. Tais argumentos são denominados **argumentos variáveis**[7]. Na definição e em alusões de funções com argumentos variáveis, os argumentos indeterminados são representados por três pontos seguidos. Por exemplo, o protótipo da função **printf()** pode ser dado por:

```
int  printf(const char *formato, ...);
```

Além dos exemplos deste tipo de função encontrados na biblioteca padrão de C, a linguagem permite que o programador defina suas próprias funções com argumentos variáveis.

Uma função com argumentos variáveis deve preencher alguns requisitos. O primeiro é que ela deve possuir pelo menos um argumento fixo (i.e., com identificador e tipo definidos, conforme foi visto anteriormente). O segundo é que os argumentos fixos da função [por exemplo, o argumento `formato` na função **printf()**] devem preceder os argumentos variáveis, representados por `...` no cabeçalho da função. O último requisito é utilizar as macros[8] **va_arg**, **va_end** e **va_start**, definidas no arquivo de cabeçalho `<stdarg.h>`, para acessar os argumentos não especificados. A lista de argumentos variáveis é considerada uma única variável do tipo **va_list**, também definido em `<stdarg.h>`.

---

[7] Neste contexto, o termo *variável* é empregado para denotar o fato de o número e os tipos dos argumentos não serem especificados na definição da função. Não existe relação direta entre o sentido deste termo aqui e o sentido usual de variável em programação.

[8] Apesar de macros desse tipo serem definidas apenas no **Capítulo 5**, aqui é suficiente saber apenas que elas são usadas (i.e., chamadas) de modo semelhante às funções.

No início do corpo de uma função de argumentos variáveis, deve-se declarar uma variável do tipo **va_list**. Esta variável (que é um ponteiro) conterá (ou melhor, apontará para) ***todos*** os argumentos passados para a função quando esta for chamada e é utilizada para acessar cada um dos argumentos variáveis da função. Por exemplo, a variável `argumentos`, declarada como:

```
va_list  argumentos;
```

apontaria para o primeiro argumento passado para a função (lembre-se de que este primeiro argumento é sempre fixo).

Antes de acessar qualquer argumento variável, a macro **va_start** deve ser utilizada para fazer com que a variável do tipo **va_list** declarada no corpo da função aponte para o ***primeiro argumento variável*** da lista de argumentos. A macro **va_start** deve receber dois argumentos: o primeiro é a variável do tipo **va_list** declarada no início da função e o segundo é o nome do ***último*** parâmetro fixo da função. O seguinte esboço de função clarifica o que foi exposto:

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

/****
*
* Função SomaComArgumentosVariaveis()
*
* Descrição:Calcular a soma de um número indeterminado de valores
            inteiros. Pelo menos um valor inteiro deve ser
            fornecido. O último argumento deve ser zero.
*
* Parâmetros:
*      argl  (entrada): primeira parcela da soma
*      ...   (entrada): as outras parcelas da soma
*
+ Retorno: A soma dos argumentos recebidos.
*
****/
int SomaComArgumentosVariaveis(int arg1,...)
{
   int       soma = arg1;
```

```
   va_list       argumentos;
   int           umArgumento;

   va_start(argumentos, arg1); //Inicia lista de argumentos

      /* Lê cada argumento e acrescente à soma.     */
      /* O valor 0 encerra o processamento da lista */
   while  ((umArgumento = va_arg(argumentos, int))  !=0)  {
      soma += um Argumento;
   }
    va_end(argumentos); /* Encerra processamento dos argumentos */

    return soma;
}

int main(void)
{
   printf("A soma de 1+2+3+4 e': %d/n",
          SomaComArgumentosVariveis(1,2,3,4,0));
    return 0;
}
```

A macro **va_arg** permite o acesso seqüencial a cada um dos argumentos variáveis da função. Esta macro possui dois argumentos: o primeiro é a variável do tipo **va_list** que aponta para a lista de argumentos (a variável `argumentos` no esquema acima) e o segundo é o ***nome do tipo*** do próximo argumento a ser acessado. Como os tipos **char** (**signed** ou **unsigned**), **short** (**signed** ou **unsigned**) e **_Bool** são implicitamente promovidos (v. **Seção 1.5.2**) para **int** (**signed** ou **unsigned**), os tipos **char**, **short** e **_Bool** não devem ser utilizados aqui.

A macro **va_arg** resulta num valor do mesmo tipo do seu segundo argumento e passa a apontar para o próximo argumento variável (se houver mais algum). Isto é, na primeira invocação desta macro, ela retorna o valor do primeiro argumento variável e passa a apontar para o segundo argumento variável; na segunda invocação, ela retorna o valor do segundo argumento variável e passa a apontar para o terceiro argumento variável e assim por diante.

Para que os valores dos argumentos variáveis sejam retornados corretamente, é importante que se informe (no segundo argumento da

macro) o tipo do argumento da função a ser processado com exatidão. Quando todos os argumentos são de um mesmo tipo conhecido em tempo de programação, esta tarefa é simples, mas, quando estes tipos são desconhecidos quando a função é criada, o programador deve elaborar meios para descobrir os tipos corretos dos argumentos variáveis. Por exemplo, o tipo de cada argumento variável da função **printf()** é descoberto através de uma busca pelo especificador de formato no primeiro argumento (string de formatação) correspondente ao dado argumento variável.

Finalmente, após processar todos os argumentos variáveis, para permitir um ***retorno normal*** da função[9], deve-se utilizar a macro **va_end**. Esta macro recebe como único argumento a variável do tipo **va_list**, que aponta para a lista de argumentos (a variável `argumentos` no exemplo anterior). Esta macro deve ***sempre*** ser utilizada para encerrar o processamento da lista de argumentos variáveis; caso contrário, a função poderá apresentar um comportamento indefinido quando for executada.

Isto é realmente importante, ainda que se tenha um meio de determinar ***quando*** o processamento da lista de argumentos variáveis deve ser encerrado. No caso da função **printf()**, o processamento do argumentos é encerrado quando o string de formatação (primeiro argumento da função) é totalmente examinado.

O programa a seguir exemplifica o uso de uma função com argumentos variáveis.

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

/****
 *
 * Função SomaComArgumentosVariveis()
 *
 * Descrição: Calcula a soma de um número indeterminado de
valores
 *      inteiros. Pelo menos um valor inteiro deve ser
```

[9] Entenda-se por ***retorno normal da função*** a remoção completa de algum dado remanescente da função na pilha de execução.

```
 *      fornecido. O último argumento deve ser zero.
 *
 * Parâmetros:
 *      arg1 (entrada): primeira parcela da soma
 *      ... (entrada): as outras parcelas da soma
 *
 * Retorno: A soma dos argumentos recebidos.
 *
 ****/
int SomaComArgumentosVariveis(int arg1, ...)
{
   int      soma = arg1;
   va_list  argumentos;
   int      umArgumento;

   va_start(argumentos, arg1); // Inicia lista de argumentos
variáveis

      /* Lê cada argumento e acrescenta à soma.     */
      /* O valor 0 encerra o processamento da lista */
   while ((umArgumento = va_arg(argumentos, int)) != 0) {
      soma += umArgumento;
   }

   va_end(argumentos);     /*  Encerra  o  processamento  dos
argumentos */

   return soma;
}

int main(void)
{
   printf("A soma de 1+2+3+4 e': %d\n",
           SomaComArgumentosVariveis(1,2,3,4,0));
   return 0;
}
```

No programa acima, a função `SomaComArgumentosVariveis()` calcula e retorna a soma de um número indeterminado de valores inteiros (no mínimo um inteiro deve ser passado na chamada da função). O valor 0 encerra o processamento desses valores. Note que existe uma maneira bem definida de saber quais são os tipos dos argumentos variáveis (no caso, todos são do tipo **int**) e quando o processamento destes argumentos deve ser encerrado.

Além das macros apresentadas nesta seção para definição de funções com listas de argumentos variáveis, o padrão C99 introduz a macro **va_copy()**, que permite copiar o conteúdo de uma variável do tipo **va_list** para outra deste mesmo tipo. Esta macro é descrita no **Volume II**.

O programa apresentado nesta seção, apesar de ser simples e ilustrar bem a implementação de uma função com lista de argumentos variáveis, tem pouca utilidade prática. Na **Seção 3.8**, será apresentado um exemplo de natureza prática de uso desse tipo de função.

3.5 Recursão

3.5.1 Funções Recursivas

A linguagem C permite a escrita de funções que chamam, direta ou indiretamente, a si mesmas. Tais funções são denominadas **recursivas**. Uma função recursiva deve conter pelo menos duas partes (casos), a saber:

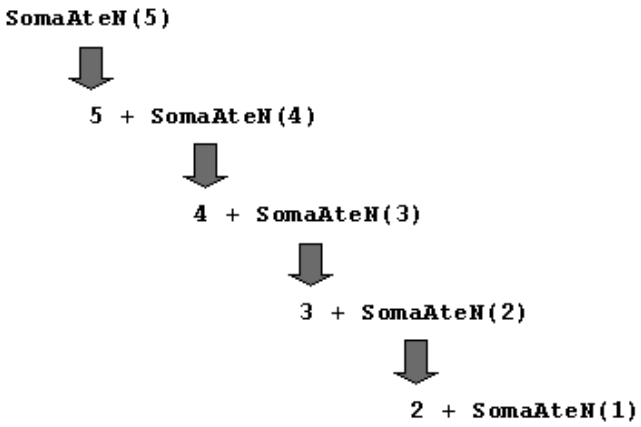
- **Caso não-recursivo**, que estabelece uma **condição de parada** da recursão e sem o qual a recursão será infinita. Esta parte da definição da função não deve fazer referência à própria função.
- **Caso recursivo**, no qual a função chama a si mesma. O programador deve garantir que uma das chamadas recursivas atinja eventualmente a condição de parada.

Uma função recursiva pode ter mais de um caso recursivo e mais de uma condição de parada.

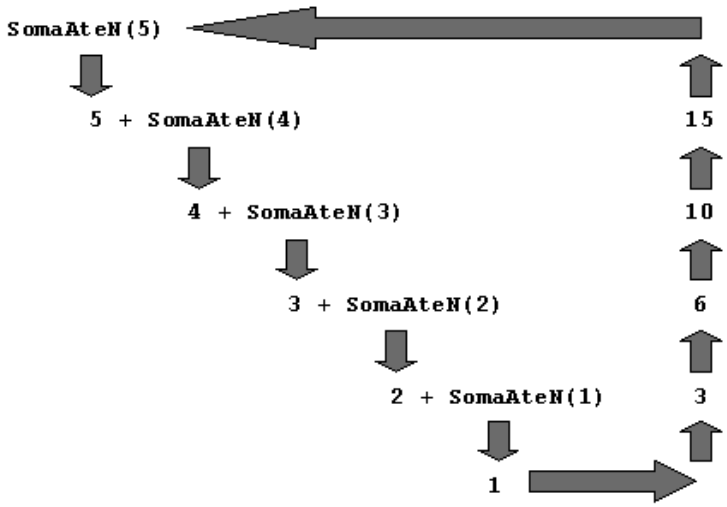
A função do exemplo a seguir ilustra o processo de recursão em C:

```
int SomaAteN(int n)
{
   if  (n <= 1)
      return n;           /* Condição de parada */
   else
      return (n + SomaAteN(n - 1)); /* Caso recursivo */
}
```

A função `SomaAteN()` retorna o valor da soma dos inteiros compreendidos entre `1` e `n`, sendo `n` o parâmetro de entrada da função. Por exemplo, a chamada `SomaAteN(5)` deve resultar em 15 (= 1 + 2 + 3 + 4 + 5). O esquema a seguir ilustra a seqüência de chamadas que ocorre quando é feita a chamada `SomaAteN(5)`:



No esquema acima, quando é feita a chamada `SomaAteN(1)`, a condição de parada é atingida e a função retorna `1`. Com este valor retornado, é possível voltar sucessivamente ao passo anterior na representação esquemática acima até que a chamada original seja atingida. Isto resulta no seguinte esquema final:



Note que, a cada chamada recursiva da função `SomaAteN()`, o valor do argumento `n` é cada vez menor, de modo que a condição de parada será eventualmente atingida. Entretanto, a função `SomaAteN()` produz resultados indesejáveis se o número introduzido for menor do que `1` (verifique isso). Uma forma de corrigir a função `SomaAteN()` é modificando-a de modo que ela alerte o usuário e seja encerrada quando `n < 1`. Isto é feito na versão da função `SomaAteN()` a seguir:

```
int   SomaAteN( int   n)
{
   if  (n < 1){
         printf("Erro: Numeros menores do que 1 nao sao
aceitos.");
      return  0;
   }
   else  if  (n == 1)
      return  n;                        /* Condição de parada */
   else
      return  (n + SomaAteN(n - 1));    /* Caso recursivo */
}
```

A função `SomaAteN()` serve como exemplo introdutório do uso de recursão em C, mas esta evidentemente não é a forma mais elegante de se resolver o problema da soma dos números inteiros compreendidos entre 1 e n. Isto é, este problema é muito mais fácil de ser resolvido utilizando um laço iterativo em vez de recursão. Além de ser mais legível, uma versão iterativa da função `SomaAteN()` irá provavelmente ser executada com um melhor desempenho, pois a versão recursiva envolve o uso de pilha para guardar parâmetros e variáveis locais a cada chamada recursiva.

3.5.2 O Problema das Torres de Hanói

Na maioria das vezes, os problemas encontrados pelo programador não precisam ser resolvidos de maneira recursiva. Isto é, a maioria dos problemas pode ser resolvida de maneira iterativa e o programador não tem que se preocupar em procurar soluções recursivas. Entretanto, existem problemas que possuem soluções naturalmente recursivas mais fáceis de serem encontradas. Um tal problema, conhecido como o ***Problema das Torres de Hanói***, será descrito a seguir.

Inicialmente, no problema das Torres de Hanói, existem três hastes e um determinado número de discos de diâmetros diferentes empilhados uns sobre os outros numa das hastes. O que o problema requer é que os discos sejam movidos de uma haste para outra obedecendo a duas restrições:

1. Nenhum disco pode ser colocado sobre um outro disco de diâmetro menor.

2. Apenas o disco do topo de uma haste pode ser movido num dado instante. Ou, em outras palavras, para mover-se um dado disco, deve-se primeiro mover os discos que estão sobre ele.

A **Figura 12** a seguir ilustra a condição inicial do problema das Torres de Hanói para três discos (os discos foram numerados para facilitar referências aos mesmos):

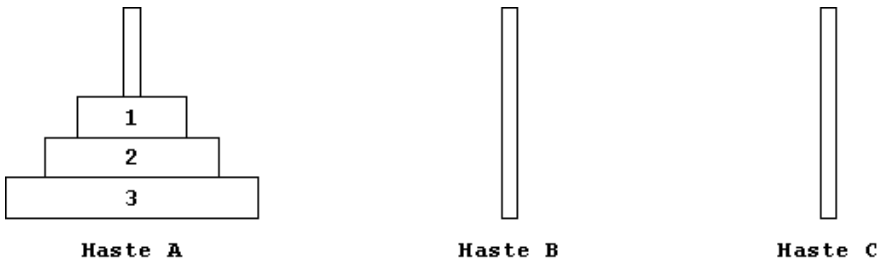
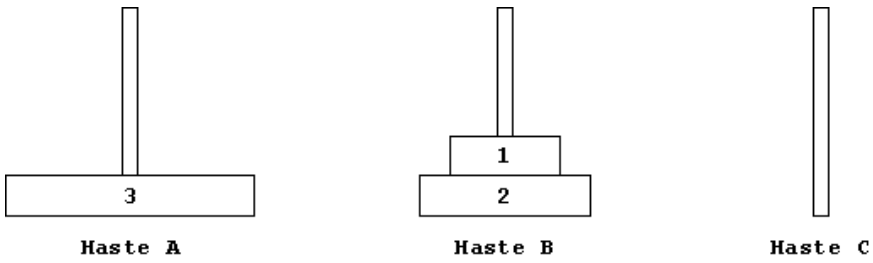


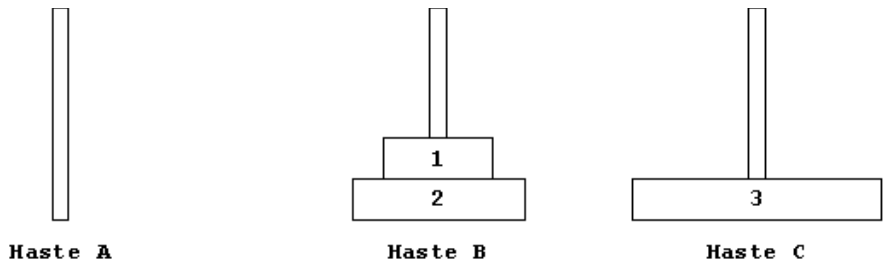
Figura 12: Problema das Torres de Hanói

Considerando o diagrama da **Figura 12**, o problema consiste em deslocar os três discos na `Haste A` (denominada de **haste de origem**) para a `Haste C` (denominada de **haste de destino**), utilizando a `Haste B` como haste auxiliar. Por enquanto, não se preocupe com entrada e saída de dados do programa a ser desenvolvido, concentre-se na solução do problema para o caso geral no qual existem `n` discos.

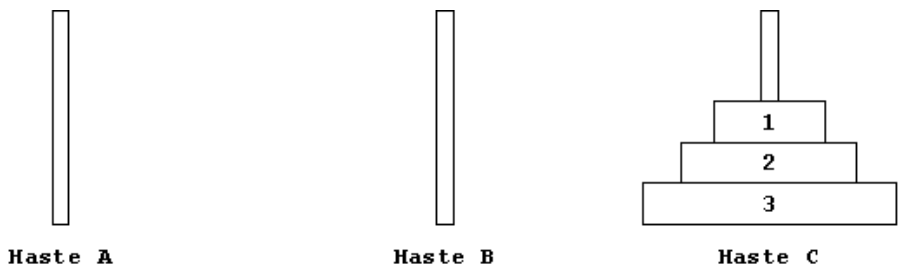
Para começar, suponha que a solução do problema para `n - 1` discos seja conhecida. Então, se for possível descrever a solução para `n` discos em termos da solução para `n - 1` discos, o problema será facilmente resolvido. De fato, isto é verdade porque, movendo-se `n - 1` discos para a `Haste B` (auxiliar) deixa-se apenas um disco para ser removido e, no caso trivial de um único disco, a solução é imediata: simplesmente mova este disco da `Haste A` para a `Haste C`. Para um melhor entendimento, considere novamente o caso particular onde `n = 3` e suponha que se pode mover dois discos de `A` para `C` utilizando `B` como auxiliar. Então, pode-se, de forma semelhante, movê-los de `A` para `B` utilizando `C` como auxiliar, o que resultaria no seguinte esquema:



Neste ponto, pode-se, então, mover o disco maior da `Haste A` para a `Haste C`, resultando na seguinte configuração:



Finalmente, pode-se aplicar a solução para dois discos aos discos 1 e 2 para movê-los de `B` para `C`, utilizando, agora, a `Haste A` como auxiliar. Isto resulta na configuração final apresentada a seguir:



Seguindo o raciocínio anterior, pode-se generalizar a solução para mover n discos da Haste A para a Haste C, utilizando a Haste B como auxiliar, por meio do seguinte algoritmo

1. Se n = 1, mova o único disco da haste A para a haste C e pare.

2. Mova os n - 1 discos do topo da haste A para a haste B utilizando C como auxiliar.

3. Mova o disco remanescente na haste A para a haste C.

4. Mova os n - 1 discos da haste B para a haste C utilizando A como auxiliar.

Não é difícil verificar que este algoritmo realmente produz a solução correta para o problema das Torres de Hanói para qualquer valor de `n` ≥ `1`. Observe que, para `n = 1`, o passo 1 resultará na solução esperada. Se `n = 2`, a solução para `n - 1` já é conhecida, de modo que os passos 2 e 4 podem ser executados sem problemas. Quando `n = 3`, a solução para `n - 1` já é conhecida e os passos 2 e 4 podem ser novamente executados. Prosseguindo com este raciocínio, pode-se mostrar que a solução fornecida pelo algoritmo acima funciona para `n = 1`, `2`, ..., `k`, onde `k` é um valor inteiro positivo arbitrário.

A solução algorítmica para o problema das Torres de Hanói ainda não está completa. É necessário ainda que se definam entrada e saída para o problema. A entrada é fácil de ser identificada, pois ela corresponde simplesmente ao número `n` de discos. A saída do algoritmo refere-se às representações de discos e hastes e como o movimento de discos de uma haste para outra deve ser apresentada. A escolha mais natural de saída parece ser de natureza gráfica, com os discos sendo movidos utilizando uma interface gráfica com animação. Este tipo de saída não será utilizado aqui, pois o interesse é entender as idéias fundamentais de recursão e não detalhes de interface gráfica. Portanto, a saída será apresentada de uma forma textual mais simples, contendo frases do tipo:

Mova o disco D da haste H1 para a haste H2

Também, serão adotadas as seguintes convenções:

(1) A haste de origem é denominada A, a haste de destino é denominada C e a haste auxiliar é denominada B.

(2) Os discos são numerados de 1 a n, a partir do disco do topo na haste de origem na situação inicial (em outras palavras, os discos são numerados de 1 a n do menor para o maior).

Um programa completo para o problema das Torres de Hanói é apresentado a seguir:

```
#include <stdio.h>

extern   void  TorresDeHanoi( int, char, char, char );

main()
{
   int  numeroDeDiscos;

   printf("Introduza o numero de discos: "); /* Solicita dado
ao usuário  */
   scanf("%d", &numeroDeDiscos); /* Não se esqueça de passar
o endereço aqui!*/
   printf("\n\n");

   TorresDeHanoi(numeroDeDiscos, 'A', 'C', 'B');
}
/****
 *
 * Função TorresDeHanoi()
 *
 * Descrição: Resolve o problema das Torres de Hanói.
 *
 * Parâmetros:
 *      numeroDeDiscos (entrada): o numero de discos
 *      hasteOrigem (entrada): a haste onde os discos se
 *                             encontram inicialmente
 *     hasteDestino (entrada): a haste onde os discos deverão
estar ao final
  *          hasteAuxiliar  (entrada):  a  haste  que  auxilia  o
movimento
 *
```

```
 * Retorno: Nada.
 *
 ****/

void TorresDeHanoi
  ( int numeroDeDiscos, /* Este estilo de cabeçalho é */
  char hasteOrigem, /* apropriado para funções cujos */
  char hasteDestino,  /* cabeçalhos são longos. Também é */
  char hasteAuxiliar )/* útil para comentar cada */
                       /* parâmetro na declaração da função.*/
                       /* Comentar parâmetros não é */
                       /* necessário neste caso. */
{
   if (numeroDeDiscos == 1){
      printf( "Mova o disco 1 da haste %c para a haste %c\n", /*
Passo 1:  */
     hasteOrigem, hasteDestino ); /* Imprime e encerra */
       return;
   }

      /* Move os n-1 discos de A para B usando C como auxiliar
(Passo 2)  */
   TorresDeHanoi(numeroDeDiscos-1,hasteOrigem,hasteAuxiliar,
hasteDestino);
   printf( "Mova o disco %d da haste %c para a haste %c\n",
/* Move último */
         numeroDeDiscos, hasteOrigem, hasteDestino );     /*
disco de A para */
                                                          /* C
(Passo 3).    */

     /* Move os n-1 discos de B para C usando A como auxiliar
(Passo 4)  */
   TorresDeHanoi(numeroDeDiscos-1,hasteAuxiliar,hasteDestino
,hasteOrigem);
}
```

O resultado da execução do programa acima com um número de discos igual a 4 seria:

```
Introduza o número de discos: 4

Mova o disco 1 da haste A para a haste B
Mova o disco 2 da haste A para a haste C
Mova o disco 1 da haste B para a haste C
```

```
Mova o disco 3 da haste A para a haste B
Mova o disco 1 da haste C para a haste A
Mova o disco 2 da haste C para a haste B
Mova o disco 1 da haste A para a haste B
Mova o disco 4 da haste A para a haste C
Mova o disco 1 da haste B para a haste C
Mova o disco 2 da haste B para a haste A
Mova o disco 1 da haste C para a haste A
Mova o disco 3 da haste B para a haste C
Mova o disco 1 da haste A para a haste B
Mova o disco 2 da haste A para a haste C
Mova o disco 1 da haste B para a haste C
```

Uma questão que pode surgir na definição da função `TorresDeHanoi()` acima é: ***Como os parâmetros desta função são escolhidos?*** Parece trivial entender por que o número de discos deve ser um parâmetro que é reduzido a cada chamada recursiva até que a condição de parada seja satisfeita. O uso das três hastes (`hasteOrigem`, `hasteDestino` e `hasteAuxiliar`) como parâmetros também não é difícil de entender. Basta perceber que soluções intermediárias do problema envolvem movimentos com as hastes `A`, `B` e `C` sendo ora origem, ora destino, ora auxiliar do movimento. Deve-se observar ainda que o programa foi facilitado pelo fato de a numeração dos discos ter sido feita do menor para o maior (verifique isto).

Note que, no programa acima, a alusão:

```
extern void TorresDeHanoi( int, char, char, char );
```

é necessária apenas porque a função `TorresDeHanoi()` foi definida após a função **main()**; i.e., se ela tivesse sido definida antes da função **main()**, a alusão não seria necessária.

A função `TorresDeHanoi()` poderia tornar-se mais geral se as apresentações de resultados, representadas pelas chamadas da função **printf()**, fossem colocadas separadamente em outra função. Esta função, que será aqui denominada `Mova()`, seria responsável pela apresentação dos resultados. A função `Mova()` deveria ter como argumentos os dados necessários para a apresentação dos resultados, que são o número do disco sendo movido e as hastes de origem e destino do movimento. A grande

vantagem desta mudança é que, com ela, a função `TorresDeHanoi()` tornar-se-ia independente das suposições sobre a apresentação dos resultados. Se, posteriormente, fosse decidido que a saída do programa seria efetuada com animação gráfica, em vez de textual, toda a mudança necessária seria restrita à função `Mova()`.

Para concluir, observe que, da mesma forma que na elaboração da função `SomaAteN()`, a solução obtida aqui para o problema das Torres de Hanói foi desenvolvida identificando-se um caso trivial (i.e., quando o número de discos é igual a 1) e uma solução para o caso geral (i.e., para `n` discos) em termos de um caso mais simples (i.e., para `n - 1` discos). Em termos de estratégia de resolução de problemas, entretanto, existe uma diferença fundamental entre os dois problemas aqui descritos. O problema de encontrar a soma dos números entre 1 e n pode facilmente ser resolvido sem o uso de recursão, e a solução iterativa não apenas é mais clara como também mais eficiente em termos de recursos computacionais. O problema das Torres de Hanói, por outro lado, não possui solução não-recursiva trivial e, portanto, representa uma situação prática na qual o uso de recursão é recomendável[10].

3.5.3 Cadeias Recursivas

Uma função pode ser recursiva sem que chame a si mesma diretamente. Isto é, uma função pode ser considerada recursiva se ela faz parte de uma **cadeia recursiva de funções**. Por exemplo, se uma função `f()` chama uma outra função `g()` que, por sua vez, chama `f()`, ambas as funções `f()` e `g()` são consideradas recursivas e são ditas formarem uma cadeia recursiva.

O perigo de se ter recursão infinita é maior em cadeias recursivas do que com funções que são diretamente recursivas. Também, em termos de estilo, cadeias recursivas não são fáceis de serem identificadas como tais,

[10] O quebra-cabeça das Torres de Hanói foi inventado pelo matemático francês Edouard Lucas em 1883 e possui propriedades matemáticas bastante curiosas. Por exemplo, utilizando o quebra-cabeça com 64 discos e movendo-se um disco por segundo, levar-se-ia 580 bilhões de anos para completar a tarefa.

pois examinando-se apenas uma das funções envolvidas não é possível perceber que a mesma chama indiretamente a si mesma.

3.6 Funções Inline (C99)

O padrão C99 permite que uma chamada de função possa ser substituída (**expandida**) adequadamente pelo corpo da função. Uma função que permite que suas chamadas sejam expandidas é denominada **função inline**. Para definir uma função *inline* utiliza-se a mesma sintaxe vista anteriormente, mas precede-se o cabeçalho da função com a palavra-chave **inline**. Por exemplo,

```
inline long F(long a, long b)
{
    return a + b;
{

int main(void)
{
    long x, y;
    ...
           // O compilador poderá expandir a seguinte
           // chamada de F() em x + y
    printf("%ld", F(x, y));
    ...
}
```

Uma função *inline* comporta-se como uma função qualquer sob todos os aspectos (por exemplo, as regras para passagem de parâmetros são as mesmas), mas as ligações entre as chamadas e a definição da função são resolvidas pelo compilador e não pelo *linker* como ocorre com funções comuns. Portanto, para ser capaz de expandir uma função no local da chamada, o compilador precisa ter conhecimento da definição completa da função neste local (i.e., neste caso, uma simples alusão da função não resolve). Isto significa dizer que a definição de uma função *inline* não pode residir num arquivo diferente daquele onde a mesma é chamada. Na prática, isto quer dizer que, se uma função *inline* é chamada em vários arquivos, a definição da função (e não apenas uma alusão à mesma) deve

ser colocada num arquivo de cabeçalho (v. **Seção 4.10**) que é incluído nos arquivos que a chamam[11].

Outra observação importante com relação a funções *inline* é que o compilador pode ignorar uma expansão desejada devido a dificuldades ou mesmo impossibilidade de atender à solicitação. Por exemplo, um compilador não pode fazer expansão de funções contendo variáveis locais de duração fixa (v. **Seção 4.2**).

Funções *inline* agilizam a execução do programa, pois reduzem o ônus associado com chamadas normais de funções. Em contrapartida, o uso de funções *inline* aumenta o tamanho do código final do programa, pois o código associado a uma função *inline* será inserido em cada local onde existe uma chamada da função. Portanto, funções *inline* são recomendadas apenas para funções contendo poucas instruções.

De acordo com o padrão C99, o compilador pode ignorar completamente o significado da palavra-chave **inline** apresentado aqui. A única função que não pode ser declarada como *inline* é a função **main()**.

3.7 Leitura e Validação de Dados I

Muitas vezes, um programa interativo requer que sejam lidos e processados vários dados até que um valor que indica o final do programa seja lido e, então, o programa seja encerrado. Portanto, um programa desta natureza segue o algoritmo geral aqui apresentado:

1. Leia um valor de acordo com as especificações do programa.

2. Enquanto o valor lido é diferente do valor que sinaliza o final do programa, faça o seguinte:

---

[11] As duas últimas sentenças parecem antagônicas, mas não o são. Quando um arquivo de cabeçalho contendo uma função inline é incluído num arquivo que contém uma chamada da função, o pré-processador faz a inclusão antes de o compilador processar o arquivo. Isso significa que, quando o arquivo que contém a chamada for compilado, a definição da função estará residindo neste arquivo.

[12] Provavelmente, o programa também deverá apresentar o resultado do processamento. Isto pode ser feito entre os passos 2.1 e 2.2 do algoritmo, quando cada valor lido produz um resultado, ou logo antes de o programa terminar, quando o conjunto de valores como um todo produz o resultado desejado.

2.1 Processe o valor lido.

2.2 Leia um valor de acordo com as especificações do programa.

3. Encerre o programa.

A essência daquilo que o programa se propõe a fazer está resumida no passo 2.1 do algoritmo acima, que corresponde ao processamento dos dados recebidos pelo programa[12]. Entretanto, é de suma importância para o bom funcionamento do programa assegurar que os dados introduzidos pelo usuário correspondem às expectativas do programa. Quer dizer, se os dados processados forem inadequados, obviamente o programa não produzirá resultados significativos.

O tema central desta seção são os passos 1 e 2.2 do algoritmo, ou seja, como ler dados e assegurar que os valores lidos realmente seguem as especificações do programa. Apesar de esta tarefa parecer simples, na maioria das vezes ela não é trivial. A estratégia a ser seguida aqui tem como base apenas o uso da função **scanf()** do módulo de biblioteca stdio. Existem outras abordagens mais sofisticadas (e mais complicadas) para resolver o problema, mas, apesar de apresentar algumas limitações, o método apresentado aqui é satisfatório em muitas situações práticas, além de ser simples e fácil de entender.

3.7.1 Sinalização do Final do Programa

Se o programa encerra quando um determinado valor (por exemplo, 0) é lido, deve-se especificar este valor utilizando uma constante simbólica. Por exemplo:

```
#define VALOR_TERMINAL  0
```

Algumas vezes o programa deve ser encerrado quando um dentre vários possíveis valores é lido, como, por exemplo, um programa que encerra quando um valor negativo é lido. Evidentemente, numa tal situação, você não pode especificar todos os valores que causam o encerramento do

programa usando constantes simbólicas, mas não terá dificuldades para adaptar a abordagem descrita aqui para a situação encontrada.

3.7.2 Leitura de Dados com Validação

A função `LeValor()` apresentada a seguir tenta ler valores no meio de entrada padrão usando **scanf()** e retorna um valor apenas quando a entrada é válida de acordo com os critérios do programa. As linhas de interesse da função foram numeradas para facilitar a discussão que segue a apresentação da função.

```
1.  long unsigned LeValor(void)
    {
2.     long     valor;
3.     unsigned teste;

4.     printf("\nIntroduza um numero inteiro positivo: ");
5.     teste = scanf("%ld", &valor);

6.     while (!teste || !VerificaValor(valor)) {
7.        if (teste) {
8.           printf("\nO valor %ld nao e' valido", valor);
9.        } else {
10.          printf("\nO valor introduzido nao e' valido");
          } /* if */

11.       printf("\nIntroduza um numero inteiro positivo: ");
12.       LimpaBuffer();
13.       teste = scanf("%ld", &valor);
       } /* while */

14.    LimpaBuffer();
15.    return valor;
    }
```

▶ Comentários sobre a função LeValor()

Nos comentários a seguir, a numeração dos itens a corresponde à numeração de linhas da função `LeValor()`.

1. Aqui, supõe-se que os valores válidos são do tipo **long unsigned**. Substitua o tipo retornado pela função por um tipo de dados adequado ao seu programa.

2. A variável `valor` é utilizada pela função **scanf()** na leitura de valores. Substitua o tipo desta variável por um tipo adequado ao seu programa.

3. A variável `teste` é utilizada para testar o valor retornado pela função **scanf()**. Não é necessário mudar nada aqui.

4. A chamada de **printf()** apresenta uma indicação de como deve ser o valor que o usuário irá introduzir. Modifique-a de acordo com as necessidades de seu programa.

5. Aqui é feita a primeira tentativa de leitura. Substitua o especificador de formato `%ld` por um especificador compatível com o tipo da variável `valor` se o tipo desta variável for modificado em sua definição (linha 2).

6. O corpo desse laço **while** será executado enquanto nenhum valor for lido ou quando o valor lido não satisfizer as especificações do programa. Você precisará modificar a condição do **while** apenas se o teste de validação da entrada for suficientemente simples para substituir a chamada da função `VerificaValor()` (v. adiante).

7. Se a condição do **if** for satisfeita, então um valor foi lido, mas este valor não é válido.

8. Informa ao usuário que a entrada introduzida não é válida, apresentando, inclusive, o valor lido. Você deve substituir adequadamente o especificador `%ld` usado em **printf()**.

9. A parte **else** corresponde a um valor introduzido que não é

do tipo especificado na chamada de **scanf()**; portanto, nada foi lido por esta função.

10. Simplesmente, informa ao usuário que a entrada introduzida não é válida. Se preferir, modifique o texto apresentado.

11. A chamada de **printf()** apresenta uma indicação de como deve ser o valor que o usuário irá introduzir. Modifique-o convenientemente para as necessidades de seu programa. Talvez o programa deva apresentar um prompt mais elaborado desta vez.

12. A função `LimpaBuffer()` esvazia o buffer associado com a entrada padrão (i.e., o teclado). É necessário limpar o buffer de entrada antes de fazer uma nova leitura porque talvez restem caracteres que não foram extraídos pela última chamada da função **scanf()**.

13. Aqui, tenta-se fazer uma nova leitura. Modifique esta instrução convenientemente conforme descrito anteriormente.

14. Novamente, a função `LimpaBuffer()` é chamada para esvaziar o buffer de entrada antes de retornar. Deste modo, garante-se que a próxima leitura (feita pela função `LeValor()` ou qualquer outra função de entrada de dados) irá começar com um buffer limpo.

15. O último valor lido satisfaz os critérios especificados. Logo, este valor é retornado.

3.7.3 Teste dos Valores Lidos

A função `VerificaValor()` apresentada a seguir verifica se o argumento recebido satisfaz os critérios do programa. Se este for o caso, ela retorna 1; caso contrário, ela retorna 0.

```
unsigned VerificaValor(long valor)
{
 return (valor >= 0);
}
```

► **Comentários sobre a função VerificaValor()**

Aqui, supõe-se que o critério que uma entrada válida do programa deve satisfazer é simplesmente que o valor introduzido seja maior do que ou igual a zero. Portanto, a função retorna 1 se o argumento recebido é maior do que ou igual a zero e 0 em caso contrário. Numa situação tão simples quanto esta, você não precisa de uma função de verificação. Isto é, você pode apenas fazer o teste de validade do valor lido diretamente na função `LeValor()`. Você pode, por exemplo, substituir, no laço **while** da função `LeValor()`, a condição:

```
(!teste || !VerificaValor(valor))
```

por:

```
(!teste || (valor < 0))
```

Em resumo, a função `VerificaValor()` é recomendada apenas quando a validação dos valores lidos é bem mais complicada do que o que se supõe aqui.

3.7.4 Limpeza do Buffer de Entrada

A função `LimpaBuffer()` lê e descarta todos os caracteres que porventura tenham sido deixados no buffer de entrada em alguma tentativa de leitura de dados. Uma limitação desta função é que, se não houver pelo menos um caractere `'\n'` (i.e., se o usuário não digitou `[ENTER]`), ela irá aguardar a introdução deste caractere antes de liberar a entrada. Portanto, antes de chamar esta função, certifique-se de que há realmente algum caractere no buffer de entrada.

```
void LimpaBuffer(void)
{
    (void) scanf("%*[^\n]");
    (void) getchar();
}
```

A primeira instrução da função `LimpaBuffer()` lê e descarta (***descartar*** corresponde a * no especificador de formato) todos os caracteres encontrados no buffer de entrada até que o caractere `'\n'` seja encontrado (`[^\n]` no especificador de formato corresponde a ler todos os caracteres até que `'\n'` seja encontrado). O próprio caractere `'\n'`, entretanto, não é lido. Portanto, a chamada de **getchar()** é necessária para completar a limpeza (i.e., ler o caractere `'\n'`).

Os operadores de *casting* **(void)** foram utilizados nas duas instruções da função `LimpaBuffer()` apenas por uma questão de legibilidade. Isto é, para indicar que as funções **scanf()** e **getchar()** retornam valores que se preferiu não usar (obviamente, por não serem necessários neste caso).

A função `LimpaBuffer()` apresentada aqui realiza exatamente a mesma tarefa que aquela apresenta na **Seção 2.6**. A diferença entre estas funções está na forma de implementação.

3.7.5 Chamada da Função de Leitura

A função **main()** a seguir lê valores apropriados para o programa e processa-os repetidamente até que um valor que sinaliza o final do programa seja lido. As linhas de interesse da função foram numeradas para facilitar a discussão que segue a apresentação da função.

```
    int main(void)
    {
1.      long unsigned valorLido;

2.      valorLido = LeValor();

3.      while(valorLido != VALOR_TERMINAL) {
4.          printf("\nO valor lido foi: %ld\n", valorLido);
5.               /*** Processe aqui o valor lido ***/
6.          valorLido = LeValor();
        }

        return 0;
    }
```

▶ **Comentários sobre a função main()**

1. A variável `valorLido` armazena o mais recente valor válido lido pelo programa. Aqui, supõe-se que os valores válidos são do tipo **long unsigned**. Substitua este tipo pelo tipo adequado ao seu programa.

2. Aqui, é feita uma leitura e validação de entrada introduzida pelo usuário por meio da função `LeValor()` descrita anteriormente.

3. O corpo deste laço **while** será repetido enquanto não se encontrar um valor, especificado pela constante `VALOR_TERMINAL`, que indique o término do programa. Se houver vários valores que sinalizem o término do programa, modifique esta condição adequadamente. Por exemplo, se o programa deve terminar quando for lido um valor maior do que 20, modifique a condição do **while** para:

```
while(valorLido <= 20)
```

Neste último exemplo, melhor ainda seria se a condição fosse escrita como: `valorLido <= VALOR_LIMITE`, onde `VALOR_LIMITE` é uma constante definida usando **#define**.

4. Esta instrução informa ao usuário qual foi o valor mais recentemente lido. Embora esta informação pareça óbvia e redundante para ser apresentada ao usuário, devido às limitações da abordagem descrita aqui, é necessário que o usuário seja notificado sobre que valor está sendo processado. Por exemplo, se o usuário digitar (acidentalmente) como entrada `2O` (leia-se dois seguido de o e não dois seguido de zero), a função `LeValor()` irá retornar 2 como sendo o valor válido lido (talvez, o usuário esperasse ter introduzido vinte).

5. Este é o espaço reservado para processamento do valor lido. Introduza aqui as instruções necessárias para este processamento, se o mesmo for simples ao ponto de não prejudicar a legibilidade do programa, ou uma chamada de função que realize o processamento desejado se este processamento tiver um certo grau de complexidade. A apresentação de resultados do processamento pode ser feita neste espaço logo após o processamento propriamente dito, quando o processamento de cada valor produz individualmente um resultado (e.g., o cálculo da raiz quadrada de cada valor), ou então logo antes da instrução **return** que encerra o programa, quando o conjunto de valores como um todo produz o resultado desejado (e.g., o cálculo da média aritmética de todos os valores introduzidos).

6. Esta instrução lê o próximo valor, conforme descrito no comentário da linha 2. O laço **while** é, então, recomeçado.

3.7.6 Limitações do Método Utilizado

O método de leitura e validação de dados apresentado acima contém outras limitações além daquelas já indicadas aqui. Estas limitações serão comentadas e superadas num método similar, no entanto mais poderoso, apresentado na **Seção 10.9**.

3.8 Interação Dirigida por Menus

Uma forma bastante comum de um programa interagir com o usuário é por meio de um **diálogo questão-resposta dirigido por menus**. Neste tipo de interação, o programa apresenta um menu de opções, cada uma das quais associada a um caractere, e espera-se que o usuário escolha uma destas opções digitando o caractere correspondente.

Suponha, por exemplo, que você deseja construir um programa que oferece as seguintes opções ao usuário:

1. Soma de dois números inteiros
2. Multiplicação de dois números inteiros

3. Divisão de dois números inteiros
4. Saída do programa

Este programa poderia ser codificado em C da seguinte maneira:

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

#define TITULO_OPCAO1 "Soma de dois numeros inteiros"
#define TITULO_OPCAO2 "Multiplicacao de dois números inteiros"
#define TITULO_OPCAO3 "Divisao de dois numeros inteiros"
#define TITULO_OPCAO4 "Saida do programa"

#define N_OPCOES   4
#define OPCAO1     '1'
#define OPCAO2     '2'
#define OPCAO3     '3'
#define OPCAO4     '4'

/****
 *
 * Função: LimpaBuffer()
 *
 * Descrição: Lê e descarta caracteres encontrados na entrada
padrão
 *
 * Parâmetros: nenhum
 *
 * Retorno: Nada
 *
 ****/
void LimpaBuffer(void)
{
   int valorLido; /* valorLido deve ser int! */

   do {
      valorLido = getchar();
   } while ((valorLido != '\n') && (valorLido != EOF));
}

/****
 *
 * Função: LeOpcao()
 *
 * Descrição: Lê e valida a opção digitada pelo usuário
```

```
 *
 * Parâmetros:
 *      menorValor (entrada): o menor valor válido
 *      maiorValor (entrada): o maior valor válido
 *
 * Retorno: A opção lida é validada
 *
 ****/
int LeOpcao(int menorValor, int maiorValor)
{
   int op;

   while (1) {
      printf("\nDigite sua opcao: ");
      op =  getchar();

      if (op >= menorValor && op <= maiorValor) {
         LimpaBuffer();
         break;
      } else {
         printf("\nOpcao invalida. Tente novamente.");
         printf("\nA opcao deve estar entre %c e %c.\n",
                 menorValor, maiorValor);
         LimpaBuffer();
      }
   }

   return op;
}

/****
 *
 * Função: LeInteiro()
 *
 * Descrição: Lê um número inteiro introduzido pelo usuário
 *
 * Parâmetros: nenhum
 *
 * Retorno: O inteiro lido
 *
 ****/
int LeInteiro(void)
{
   int  umInt, nValoresLidos;

   printf("Digite um valor inteiro: ");
   nValoresLidos = scanf("%d", &umInt);
```

```
    while (nValoresLidos == 0) { // Nenhum inteiro foi lido
ainda
      LimpaBuffer();
      printf("Entrada incorreta. Digite um valor inteiro: ");
      nValoresLidos = scanf("%d", &umInt);
   }

   return umInt;
}

/****
 *
 * Função: ApresentaMenu
 *
 * Descrição: Apresenta um menu com um número indeterminado de
opções
 *
 * Parâmetros:
 *      nItens (entrada):  número de itens no menu
 *       menorOpcao (entrada): caractere associado ao primeiro
item
 *
 * Retorno: Nada
 *
 ****/
void ApresentaMenu(int nItens, int menorOpcao, ...)
{
   int i;
   va_list   argumentos;

       /* Inicia lista de argumentos variáveis */
   va_start(argumentos, menorOpcao);
     /* Lê cada argumento e imprime na tela. Note que o  */
     /* tipo de cada argumento é char *, que é o tipo que */
     /* representa strings em C (v. Capítulo 8) */
   for (i = 0; i < nItens; ++i) {
       printf("%c-%s\n", menorOpcao++, va_arg(argumentos, char
*));
   }

   va_end(argumentos);     /*  Encerra  o  processamento  dos
argumentos */
}

int main(void)
{
```

```
   unsigned char op;
   int           inteiro1, inteiro2;
   unsigned int  saida = 0;

   do {
      ApresentaMenu( N_OPCOES, OPCAO1,
                     TITULO_OPCAO1, TITULO_OPCAO2,
                     TITULO_OPCAO3, TITULO_OPCAO4 );
      op = LeOpcao(OPCAO1, OPCAO1 + N_OPCOES - 1);

      switch(op) {
         case OPCAO1:
            inteiro1 = LeInteiro();
            inteiro2 = LeInteiro();
            printf("%d + %d = %d\n", inteiro1, inteiro2,
                    inteiro1 + inteiro2);
            break;

         case OPCAO2:
            inteiro1 = LeInteiro();
            inteiro2 = LeInteiro();
            printf("%d * %d = %d\n", inteiro1, inteiro2,
                    inteiro1 * inteiro2);
            break;

         case OPCAO3:
            inteiro1 = LeInteiro();
            inteiro2 = LeInteiro();
          printf("%d / %d = %d\n", inteiro1, inteiro2,
                    inteiro1 / inteiro2);
            break;

         case OPCAO4:
            saida = 1;
       printf("Obrigado por usar este programa. Bye.");
            break;
         default:
            printf("Este programa possui um bug.");
            return 1;
      }
   } while (!saida);

   return 0;
}
```

O programa apresentado aqui, apesar de relativamente longo, é bastante simples e espera-se que o aluno possa entender o modelo e utilizá-lo em situações práticas.

3.9 Exercícios de Revisão

1. Uma variável de um tipo elementar sempre contém um valor válido do tipo com o qual ela é declarada, mesmo quando ela não é iniciada. No entanto, uma variável que representa um ponteiro nem sempre contém um valor válido. Por quê?

2. O que significa indireção de um ponteiro?

3. (a) O operador de endereço pode ser aplicado a qualquer tipo de variável? (b) E o operador de indireção?

4. O que é e para que serve um ponteiro nulo?

5. Cite algumas vantagens decorrentes do uso de funções num programa em C.

6. O que são e para que servem os argumentos (ou parâmetros) de uma função?

7. (a) Qual é o propósito da instrução **return**? (b) Uma função pode conter mais de uma instrução **return**? Explique. (c) Uma função cujo tipo de retorno é **void** pode conter uma instrução **return**?

8. (a) O que são parâmetros formais? (b) O que são parâmetros reais? (c) Qual é a relação entre parâmetros formais e reais?

9. Que regras devem ser satisfeitas na passagem de parâmetros numa chamada de função?

10. (a) Quando uma função é chamada, os nomes dos parâmetros reais devem coincidir com os nomes dos respectivos parâmetros formais? (b) Quando uma alusão é feita por meio do protótipo de uma função, os nomes dos argumentos no protótipo devem coincidir com os nomes dos parâmetros formais na definição da função?

11. Por que se diz que a passagem de parâmetros em C se dá *apenas* por valor?

12. Em que situações é aconselhável utilizar um ponteiro como argumento de uma função?

13. (a) O que é um parâmetro de entrada? (b) O que é um parâmetro de saída? (c) O que é um parâmetro de entrada e saída?

14. Qual é o significado da instrução `return 0;` encontrada na maioria das funções **main()**?

15. Escreva protótipos para funções que possuam argumentos e retornos dos seguintes tipos:

(a) Retorno: nenhum
Argumentos: **float** e ponteiro para **char**
(b) Retorno: ponteiro para **unsigned int**
Argumento: nenhum

16. Explique o uso de **void** nos seguintes cabeçalhos de funções:

(a) `void f(int x)`
(b) `int g(void)`

17. (a) Determine o que a seguinte função recursiva realiza:

```
int   MinhaFuncao(int  x)
{
     if  (!n){
         return  0;
     }
     return  (n + MinhaFuncao(n - 1));
}
```

(b) Escreva uma função iterativa que tenha o mesmo efeito da função do item anterior.

18. Por que uma função recursiva é mais ineficiente do que uma função interativa equivalente?

19. Em que situações é recomendado o uso de funções recursivas?

20. (a) O que é uma função ***inline***?

(b) Que vantagem uma função ***inline*** oferece em relação a uma função convencional?

(c) Existe alguma desvantagem no uso de tais funções?

21. (a) O que é uma alusão a uma função?

(b) Quando uma alusão é requerida num programa em C?

(c) O que é protótipo de uma função?

(d) Qual é a relação entre protótipo e alusão de uma função? (e) Qual é a vantagem advinda do uso de protótipos de funções em relação ao uso do estilo antigo de alusões?

22. Escreva o corpo da função a seguir em uma única linha de instrução utilizando o operador condicional (**? :**).

```
unsigned  MinhaFuncao2( long   x )
{
    if ( x < 0)
        return  0;
    else
        return  1;
}
```

23. (a) Escreva uma versão iterativa da função `SomaAteN()` apresentada na **Seção 3.5.1** utilizando um laço de repetição **for** no corpo da função.
(b) Existe um modo ainda mais simples de calcular a soma de 1 até n. Você seria capaz de descobrir qual é?

24. Descreva a saída gerada por cada um dos seguintes programas:

(a)

```
#include <stdio.h>

int Funcao(int  x)
{
   int  y = 0;

   y += x;

   return  y;
}

int main()
{
    int a, contador;

    for (contador = 1; contador <= 5; ++contador)  {
       a = Funcao(contador);
       printf("%d  ", a);
    }

    return 0;
}
```

(b)

```
#include <stdio.h>

int Funcao1(int  a)
{
   int  b = 1;

   b += 1;

   return  b + a;
}

int Funcao2(int x)
{
   int  b;

   b = Funcao1(x);
```

```
   return  b;
}

int main()
{
   int  a = 0, b = 1, contador;

   for (contador = 1; contador <= 5; ++contador)  {
      b += Funcao1(a) + Funcao2(a);
      printf("%d  ", b);
   }

   return 0;
}
```

25. Apresente uma situação na qual uma interação por meio de menus seja adequada.

26. (a) O que é uma função com lista de parâmetros variáveis?

(b) Que critérios a lista de parâmetros de uma tal função precisam ser satisfeitos?

27. Explique por que o programa a seguir é abortado apenas quando a função `DivideInt()` é executada.

```
#include <stdio.h>

int DivideInt(int x, int y)
{
   return x/y;
}

float DivideFloat(float x, float y)
{
   return x/y;
}

int main()
{
   printf("Chamando DivideFloat(0, 0)...\n");
      DivideFloat(0,  0);    /*  Esta  chamada  NÃO  aborta  o
```

```
programa...*/

   printf("Chamando DivideInt(0, 0)...\n");
   DivideInt(0, 0);    /* Mas esta chamada SIM */

   printf("Programa bem sucedido\n");

   return 0;
}
```

3.10 Exercícios de Programação

EP3.1) Escreva um programa contendo as definições de variáveis a seguir que imprima o endereço de cada uma das variáveis:

```
char   c;
int    k;
float  f;
```

EP3.2) Escreva um programa em C que receba um número inteiro positivo no meio de entrada e responda como saída se o número é primo ou não.

EP3.3)

(a) Escreva uma função em C com dois parâmetros: (1) um parâmetro do tipo **double** e (2) um parâmetro do tipo **int**. Esta função deverá retornar o valor do primeiro parâmetro elevado ao segundo. Em outras palavras, se o primeiro parâmetro é denominado `x` e o segundo denominado `n`, esta função deverá retornar o valor x^n.

(b) Escreva um programa em C que receba como entrada um valor de ponto-flutuante e um valor inteiro. Utilize a função descrita em (a) para calcular e imprimir x^n, onde `x` e `n` são, respectivamente, os dois valores recebidos como entrada pelo programa.

EP3.4) Suponha que você precise classificar caracteres lidos no meio de entrada nas categorias apresentadas na tabela a seguir:

CATEGORIA	CARACTERES
Espaço em branco	espaço, '\n', '\r', '\t'
Pontuação	!, ?, ;, :, .
Letra	a-z, A-Z
Dígito	0-9
Desconhecido	qualquer outro caractere

(a) Defina constantes para cada uma das categorias acima (por exemplo, `ESPACO_EM_BRANCO`).

(b) Escreva uma função que receba um caractere como entrada e retorne uma das constantes definidas no item (a) de acordo com a classificação do caractere. Utilize apenas instruções **if-else** e **return** nesta função.

(c) Escreva uma função com o mesmo objetivo da função do item (b), mas que utilize apenas instruções **switch** e **return**.

(d) Qual das duas versões é melhor e por quê?

EP3.5) Escreva uma função recursiva em C, denominada `Multiplica()`, que avalie o produto de dois números inteiros não-negativos usando apenas adição.

EP3.6) Escreva uma função recursiva em C, denominada `Soma()`, que avalie a soma de dois números inteiros não-negativos usando a função `Sucessor()` definida como:

```
int  Sucessor(int  x)
{
   return  ++x;
}
```

EP3.7) (a) Escreva uma função recursiva que calcule o máximo divisor comum de dois números inteiros positivos. (b) Escreva uma versão iterativa da função solicitada no item (a).

EP3.8) Escreva um programa que lance um dado `n` vezes e imprima o percentual de aparecimento de cada face do dado. O valor n é introduzido pelo usuário, sendo que 0 encerra o programa. Exemplo de interação com o programa:

```
[Apresentação do programa]
Numero de lançamentos do dado (0 encerra o programa): 10
Percentagem da face 1:  20.00%
Percentagem da face 2:  10.00%
Percentagem da face 3:  30.00%
Percentagem da face 4:   0.00%
Percentagem da face 5:  20.00%
Percentagem da face 6:  20.00%
Numero de lançamentos do dado (0 encerra o programa):
5000
Percentagem da face 1:  17.06%
Percentagem da face 2:  16.32%
Percentagem da face 3:  17.22%
Percentagem da face 4:  16.04%
Percentagem da face 5:  17.08%
Percentagem da face 6:  16.28%

Numero de lançamentos do dado (0 encerra o programa): -1
-1 não é um valor válido
Numero de lançamentos do dado (0 encerra o programa): 0
```

SUGESTÕES

1) Você irá precisar de pelo menos duas funções: uma para ler valores e outra para lançar o dado e imprimir os resultados.

2) Utilize a sugestão apresentada no exercício **EP2.14** (**Capítulo 2**) para gerar números aleatórios entre 1 e 6 correspondentes às faces do dado.

3) Utilize o especificador de formato `%6.2f` na função `printf()` para imprimir as porcentagens com duas casas decimais.

4) Utilize o especificador de formato `%%` na função `printf()` para imprimir o símbolo de porcentagem.

EP3.9) Um número é **perfeito** se ele é igual à soma de seus divisores menores do que ele próprio. Assim, por exemplo, 6 é perfeito, pois `6 = 1 + 2 + 3`. (a) Escreva uma função **iterativa** denominada `EhPerfeito()`, que retorne 1 se o inteiro positivo recebido como argumento seja perfeito e 0 em caso contrário. (b) Escreva um programa que receba um número inteiro maior do que ou igual a zero como entrada e imprima se o número é perfeito ou não. O programa deve terminar quando o usuário introduzir o valor 0. Exemplo de interação com o programa:

```
[Apresentação do programa]
Introduza um número inteiro positivo: 10
10 não é um número perfeito
Introduza um número inteiro positivo: 28
28 é um número perfeito
Introduza um número inteiro positivo: -5
-5 não é um inteiro válido
Introduza um número inteiro positivo: 6.0
Entrada inválida
Introduza um número inteiro positivo: 0
```

EP3.10) Repita o exercício **EP3.9**, desta vez usando uma função **recursiva** para determinar se o número é perfeito ou não.

EP3.11) Escreva um programa em C que receba uma linha de texto do teclado e imprima este texto invertido. Exemplo de interação:

```
Introduza uma linha de texto: roma
amor
```

Sugestão

Escreva uma função recursiva que utilize as funções **getchar()** e **putchar()** para ler e imprimir caracteres, respectivamente. Não utilize **scanf()** ou **printf()**. Se você tentar resolver este problema sem usar recursão, terá uma grande dor de cabeça!

EP3.12) Modifique o programa para o problema das Torres de Hanói apresentado na **Seção 3.5.2** acima considerando que os discos são numerados do maior para o menor.

EP3.13) Implemente a função `Mova()` responsável pela apresentação de resultados, conforme descrição apresentada na **Seção 3.5.2**, e substitua as chamadas da função **printf()** na função `TorresDeHanoi()` por chamadas da função `Mova()`.

EP3.14) Um algoritmo, denominado **método de busca binária**, para calcular a raiz cúbica de um número `x` é o seguinte:

1. Comece com um limite inferior e outro superior.

   1.1 Se o número for maior do que 1, use 1 como limite inferior e o próprio número `x` como limite superior.

   1.2 Se o número for menor do que 1, use `x` como limite inferior e 1 como limite superior.

2. Se a diferença entre os limites inferior e superior for menor do que um certo valor de precisão (por exemplo, `0.000001`), então o resultado será a média aritmética desses limites e o problema estará resolvido.

3. Se e o problema ainda não estiver resolvido, verifique se a média dos limites inferior e superior é maior ou menor do que a raiz cúbica de `x` elevando esta média ao cubo.

   3.1 Se a média for menor do que a raiz cúbica de `x`, repita o processo a partir do passo 2 considerando agora a média como limite inferior e mantendo inalterado o limite superior.

   3.2 Se a média for maior do que a raiz cúbica de `x`, repita o processo a partir do passo 2 mantendo o mesmo limite inferior e considerando a média como limite superior.

(a) Implemente o algoritmo acima como uma função recursiva em C que receba como parâmetros um número de ponto-flutuante `x` e os limites inferior e superior descritos no algoritmo e que retorne a raiz cúbica de `x`.

(b) Escreva um programa que solicite um valor numérico do usuário, calcule a raiz cúbica deste valor utilizando a função descrita em (a) e imprima o resultado.

Sugestões

1) Defina a precisão como uma constante simbólica. Por exemplo:

```
#define PRECISAO 0.000001
```

2) Utilize o seguinte protótipo para a função que calcula a raiz cúbica:

```
double RaizCubica(double x, double inferior,
                  double superior);
```