Capítulo 5

Subprogramas

OBJETIVOS

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar a seguinte terminologia referente à linguagem C:
  - ❒ Ponteiro
  - ❒ Ponteiro nulo
  - ❒ Operador de endereço
  - ❒ Operador de indireção
  - ❒ Subprograma
  - ❒ Função
  - ❒ Cabeçalho de função
  - ❒ Corpo de função
  - ❒ Modo de parâmetro
  - ❒ Chamada de função
  - ❒ Parâmetro de entrada
  - ❒ Parâmetro de saída
  - ❒ Parâmetro de E/S
  - ❒ Parâmetros real e formal
  - ❒ Passagem de parâmetro
  - ❒ Alusão de função
  - ❒ Protótipo de função
  - ❒ Retorno de função
  - ❒ Duração de variável
  - ❒ Duração fixa
  - ❒ Duração automática
  - ❒ Escopo de bloco
  - ❒ Escopo de função
  - ❒ Escopo de arquivo
  - ❒ Escopo de identificador
  - ❒ Escopo de programa
  - ❒ Ocultação de identificador
- Descrever como são usados os símbolos ***** e **&** em C
- Explicar a relação existente entre a abordagem dividir e conquistar e o uso de subprogramas
- Esclarecer as situações nas quais o uso de subprogramas é recomendável num programa
- Relatar o papel desempenhado por parâmetros numa função
- Enumerar vantagens decorrentes do uso de funções num programa em C
- Descrever e usar as seguintes palavras-chave da linguagem C:
  - ❒ **void** ❒ **return** ❒ **extern** ❒ **auto**
- Definir um ponteiro e usá-lo para acessar o conteúdo de uma variável
- Especificar e implementar uma função em C
- Discutir como a duração de uma variável afeta seu valor
- Explicitar as situações em que um parâmetro formal deve ser declarado como ponteiro
- Explicar as regra de casamento entre parâmetros numa chamada de função
- Discorrer sobre como ocorre a iniciação de uma variável de acordo com sua duração
- Expor interação dirigida por menus

5.1 Introdução

UMA DAS ABORDAGENS mais utilizadas na construção de programas de pequeno porte é o método de refinamentos sucessivos (v. **Seção 2.2**). Utilizando essa abordagem, a descrição de um problema é dividida em subproblemas menos complexos do que o problema original. Então, cada subproblema é subdividido sucessivamente em outros subproblemas cada vez menos complexos até que cada um deles seja resolvido por operações elementares da linguagem de programação utilizada ou existam funções de biblioteca que realizem a tarefa.

Em C, **função** é o nome genérico dado a **subprograma**, **rotina** ou **procedimento** em outras linguagens de programação. Mais precisamente, uma função consiste num conjunto de instruções e declarações que executam *uma tarefa específica*, usualmente, mais complexa do que qualquer operação elementar da linguagem C.

Este capítulo visa explorar definições e usos 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 deste capítulo, é importante que a seção sobre endereços e ponteiros seja bem compreendida. Portanto leia e releia essa seção e convença-se de que realmente entendeu todos os conceitos e exemplos contidos nela antes de estudar os demais assuntos. Este capítulo apresenta ainda dois tópicos essenciais para a construção de programas interativos amigáveis ao usuário: leitura e validação de dados e interação dirigida por menus.

5.2 Endereços e Ponteiros

5.2.1 Endereços

Qualquer variável definida num programa em C possui um endereço que indica o local onde ela encontra-se armazenada em memória. Frequentemente, é necessário utilizar o endereço de uma variável num programa, em vez de seu próprio conteúdo, como será visto mais adiante neste capítulo.

O endereço de uma variável pode ser determinado por meio do uso do operador de endereço, representado por **&**. Suponha, por exemplo, a existência da seguinte definição de variável:

```
int  x;
```

então, a expressão:

```
&x
```

resulta no endereço atribuído à variável `x` quando ela é alocada.

5.2.2 Ponteiros

Ponteiro é uma 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 uma variável de qualquer tipo (p. ex., **int**, **double** ou outro tipo mais complexo) e ponteiros que apontam para variáveis de tipos diferentes são também considerados de tipos diferentes. Assim, para definir um ponteiro é preciso especificar o tipo de variável para a qual ele pode apontar.

Uma definição de ponteiro em C tem o seguinte formato:

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

Por exemplo,

```
int  *ponteiroParaInteiro;
```

define a variável `ponteiroParaInteiro` como um ponteiro capaz de apontar para qualquer variável do tipo **int**. Nesse exemplo, diz-se que o tipo da variável `ponteiroParaInteiro` é ponteiro para **int** ou, equivalentemente, que seu tipo é **int** *. Em geral, se uma variável é definida como:

```
umTipoQualquer * ptr;
```

seu tipo é `umTipoQualquer *` ou ponteiro para `umTipoQualquer`.

O asterisco que acompanha a definição de qualquer ponteiro é denominado **definidor de ponteiro** e a posição exata dele entre o tipo apontado e o nome do ponteiro é irrelevante. Isto é, aparentemente, a maioria dos programadores de C prefere definir um ponteiro assim:

```
int  *ponteiroParaInteiro;
```

enquanto outros preferem declarar assim:

```
int* ponteiroParaInteiro;
```

ou assim:

```
int * ponteiroParaInteiro;
```

Em suma, nenhum dos três formatos acima apresenta vantagens em relação ao outro. É apenas questão de gosto pessoal do programador. Mas, é importante salientar que se mais de um ponteiro de um mesmo tipo estiver sendo definido de forma abreviada, deve haver um asterisco para cada ponteiro. Por exemplo, na definição de variáveis:

```
int * ponteiroParaInteiro1, ponteiroParaInteiro2;
```

apesar de o nome da segunda variável demonstrar a intenção do programador, essa variável não será um ponteiro. Isto é, o definidor de ponteiro * aplica-se apenas à primeira variável. Portanto o correto nesse caso seria definir os dois ponteiros como:

```
int *ponteiroParaInteiro1, *ponteiroParaInteiro2;
```

Ponteiros podem ser iniciados da mesma forma que outros tipos de variáveis. Por exemplo, a segunda definição a seguir:

```
int    meuInteiro = 5;
int   *ponteiroParaInteiro = &meuInteiro;
```

define a variável `ponteiroParaInteiro` como um ponteiro para **int** e inicia seu valor 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.

Esquematicamente, as duas variáveis do último exemplo apareceriam em memória como na **Figura 5–1**.

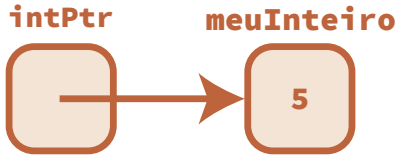

FIGURA 5–1: REPRESENTAÇÃO ESQUEMÁTICA DE PONTEIRO 1

Ao contrário do que ocorre com os tipos aritméticos (v. **Seção 3.10.1**), as regras para compatibilidade entre ponteiros são rígidas:

- Dois ponteiros só são compatíveis se eles forem exatamente do mesmo tipo. Por exemplo, dadas as seguintes definições de variáveis:

```
int    *ptrInt1, *ptrInt2;
double *ptrDouble;
```

  os ponteiros `ptrInt1` e `ptrInt2` são compatíveis entre si (i.e., as atribuições `ptrInt1 = ptrInt2` e `ptrInt2 = ptrInt1` são permitidas), mas o ponteiro `ptrDouble` não é compatível nem com `ptrInt1` nem com `ptrInt2`.

- Um ponteiro só é compatível com o endereço de uma variável se a variável for do tipo para o qual o ponteiro aponta. Considere, por exemplo, as seguintes definições de variáveis:

```
int     umInt;
int    *ptrInt;
double *ptrDouble;
```

  ao ponteiro `ptrInt` pode-se atribuir o endereço da variável `umInt`, mas não se pode atribuir esse endereço ao ponteiro `ptrDouble`.

Infelizmente, apesar de a desobediência a essas regras quase sempre causar graves problemas, alguns compiladores de C apenas emitem mensagens de advertência quando é o caso.

5.2.3 Indireção de Ponteiros

A importância de ponteiros em programação reside no fato de eles permitirem acesso ao conteúdo das variáveis para as quais eles apontam. O acesso ao conteúdo do espaço em memória apontado por um ponteiro é efetuado por meio de uma operação de **indireção**. Isto é, a indireção de um ponteiro resulta no valor contido no espaço em memória para onde o ponteiro aponta. Para acessar esse valor, precede-se o ponteiro com o **operador de indireção**, representado por `*`. Por exemplo, dadas as seguintes definições:

```
double  umDouble = 3.6;
double *ptrDouble = &umDouble;
```

o valor de `*ptrDouble` é `3.6`, enquanto o valor de `ptrDouble` é o endereço associado à variável `umDouble` quando ela é alocada. Nesse mesmo exemplo, o valor de `*ptrDouble` nesse ponto do programa será sempre `3.6`, mas o valor de `ptrDouble` (sem indireção) poderá variar entre uma execução e outra do programa. Esquematicamente, a relação entre as duas variáveis do exemplo pode ser representada como na **Figura 5–2**.

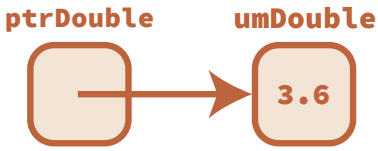

Figura 5–2: Representação Esquemática de Ponteiro 2

Neste ponto, não é mais novidade que o nome de uma variável representa seu conteúdo. Por exemplo, se `x` é uma variável, na expressão `x + 2`, é o conteúdo de `x` que é somado a `2`; na expressão `x = 5`, é o conteúdo de `x` que recebe o valor `5`; e assim por diante. O acesso ao conteúdo de uma variável usando seu nome é denominado **acesso direto**. Até o início da presente seção, esse era o único tipo de acesso que havia sido explorado.

Agora, surge um novo meio de acesso ao conteúdo de uma variável; i.e., por meio da aplicação do operador de indireção sobre um ponteiro que aponta para a variável. Esse novo tipo de acesso é denominado **acesso**

indireto porque ele requer dois passos: (1) acesso ao conteúdo do ponteiro para determinar o endereço da variável apontada e (2) de posse do endereço dessa variável, acesso ao seu conteúdo. A denominação *operador de indireção* é derivada do fato de o operador que usa esse nome permitir acesso indireto.

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 desses operadores é a segunda mais alta dentre todos os operadores da linguagem C e a associatividade deles é à direita.

O operando do operador de endereço deve ser uma variável e o operando do operador de indireção deve ser um endereço constante ou um ponteiro. Esses operadores compartilham a seguinte propriedade, que é válida para qualquer variável `x`:

***&x** é equivalente a* `x`

Pode-se alterar o conteúdo de uma variável apontada por um ponteiro utilizando o operador de indireção em conjunto com qualquer operador com efeito colateral. Por exemplo, considerando as definições:

```
double  umDouble = 3.6;
double *ptrDouble = &umDouble;
```

a instrução:

```
*ptrDouble = 1.6; /* Altera INDIRETAMENTE o conteúdo de umDouble */
```

atribuiria o valor `1.6` ao conteúdo da posição de memória apontada por `ptrDouble`. É interessante notar que essa última operação equivale a modificar o valor da variável `umDouble` sem fazer referência direta a ela. Isto é, a última instrução é equivalente a:

```
umDouble = 1.6; /* Altera DIRETAMENTE o conteúdo de umDouble */
```

As duas últimas instruções são funcionalmente equivalentes, mas a segunda é mais eficiente pois o acesso ao conteúdo da variável é feito diretamente.

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

```
double  *ptrDouble = &umDouble;
```

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

```
*ptrDouble = 1.6;
```

`*ptrDouble` recebe um valor do tipo **double**. Essa interpretação é errônea, entretanto, pois o asterisco na declaração de `ptrDouble` *não* representa o operador de indireção. Quer dizer, na iniciação, o endereço é atribuído ao ponteiro `ptrDouble` e não à expressão `*ptrDouble`. Apenas na instrução de atribuição acima, o operador de indireção é utilizado.

A essa altura, você deve apresentar a seguinte dúvida: *se é possível acessar o conteúdo de uma variável diretamente usando seu próprio nome, em que situação um ponteiro é necessário para efetuar esse acesso, se assim é mais ineficiente?*. Se, neste instante, você apresenta essa dúvida, saiba que ela é legítima, pois, na realidade, ponteiros não são utilizados conforme exposto nesta seção. Isto é, a exposição apresentada aqui é única e exclusivamente didática. Na prática, ponteiros são usados quando se tem um espaço em memória associado a uma variável, mas não se tem acesso ao nome dela (v. **Seção 5.5.3**) ou quando se tem um espaço em memória que não está associado a nenhum nome de variável. Nesse último caso, o espaço em memória é denominado **variável anônima** e esse assunto será discutido no **Capítulo 12**.

5.2.4 Ponteiro Nulo

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

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

Qualquer tentativa de acesso ao conteúdo apontado por um ponteiro nulo causa aborto de programa em qualquer hospedeiro no qual ele esteja sendo executado. Em outras palavras, a aplicação do operador de indireção a um ponteiro nulo é reconhecida como irregular por qualquer sistema operacional e causa um erro de execução (i.e., o programa é encerrado abruptamente). Considere, por exemplo, o seguinte programa:

```
#include <stdio.h> /* Entrada e Saída */
int main(void)
{
   int *ptr;
   ptr = 0;  /* Torna p um ponteiro nulo */
   printf("\n*ptr = %d\n", *ptr);
   return 0;
}
```

Se você compilar e tentar executar esse programa, observará que ele é abortado.

A seguinte definição de constante simbólica torna expressões de atribuição e comparação envolvendo ponteiros nulos mais legíveis:

```
#define NULL  0
```

Na realidade, você não precisa definir essa constante, pois ela é definida em vários cabeçalhos da biblioteca padrão de C, notadamente em `<stdio.h>`, `<string.h>` e `<stdlib.h>`. Portanto, para utilizá-la, você pode incluir em seu programa qualquer um desses cabeçalhos por meio da diretiva **#include**.

É importante salientar que, neste contexto, zero é um valor simbólico. Isto é, quando se atribui zero a um ponteiro, ele não recebe um endereço com esse valor, mas sim um valor que depende de implementação.

Parece estranho à primeira vista que o programador atribua um valor a um ponteiro que causará o aborto do programa. Acontece que, quando um ponteiro assume um valor inválido que não seja nulo e se tenta acidentalmente acessar o conteúdo apontado, as consequências podem ser desastrosas e o erro é muito mais difícil de detectar, pois o sistema operacional nem sempre é capaz de apontar a invalidade do ponteiro. Assim, na prática, enquanto um ponteiro é nulo, esse fato indica que não existe nenhum valor válido para lhe ser atribuído.

5.3 Funções

Conforme foi antecipado no início deste capítulo, em C, função é um subprograma que consiste num conjunto de instruções e declarações que executam *uma tarefa específica*. 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 se torna difícil, ela deve ser subdividida em funções menores e mais fáceis de ser entendidas.

Uma função pode ainda ser vista como uma abreviação para um conjunto de instruções. Se esse conjunto de instruções aparece mais de uma vez num programa, ele precisa ser definido apenas uma vez dentro do programa,

mas pode ser invocado nos vários pontos do programa em que sejam necessárias. Outros benefícios obtidos com o uso de funções num programa são:

- **Facilidade de manutenção**. Quando uma sequência de instruções que aparece repetidamente num programa é confinada em uma função, sua modificação, quando necessária, precisa ser efetuada apenas num único local.
- **Melhora de legibilidade**. Mesmo que uma sequência de instruções ocorra apenas uma vez num programa, às vezes, é preferível mantê-la confinada numa função, substituindo sua ocorrência por uma chamada da função. Desse modo, além de melhorar a legibilidade do programa, pode-se ter uma visão geral do programa no nível de detalhes desejado.

Funções podem aparecer de três maneiras diferentes num programa:

[1] Em forma de **definição**, que especifica aquilo que a função realiza, bem como os dados (**parâmetros**) que ela utiliza e produz como resultado.

[2] Em forma de **chamadas**, que causam a execução da função.

[3] Em forma de **alusões**, que contêm parte da definição da função e servem para informar o compilador que a função aludida é definida num local desconhecido por ele (frequentemente, num outro arquivo).

5.4 Definições de Funções

Uma **definição** de função representa a implementação da função e é dividida em duas partes:

[1] **Cabeçalho** que informa o nome da função, qual é o tipo do valor que ela produz e quais são os dados de entrada e saída (parâmetros) que ela manipula.

[2] **Corpo da função** que processa os parâmetros de entrada para produzir os resultados desejados.

5.4.1 Cabeçalho

O cabeçalho de uma função informa o tipo do **valor retornado** pela função, seu **nome** e quais são os **parâmetros** que ela utiliza.

O formato de cabeçalho de uma função é:

tipo-de-retorno nome-da-função(declarações-de-parâmetros)

A propósito, existe outro formato de cabeçalho de função, mas ele é obsoleto e não será abordado neste livro.

Tipo de Retorno de uma Função

O valor retornado por uma função corresponde ao que ela produz como resultado de seu processamento e que pode ser usado numa expressão. Considere, por exemplo, o programa a seguir:

```
#include <stdio.h>   /* printf()    */
#include "leitura.h" /* LeInteiro() */

int SomaAteN(int n)
{
   int i, soma = 0;

   for (i = 1; i <= n; ++i) {
      soma = soma + i;
   }

   return soma;
}
```

```
int main(void)
{
   int umInteiro, aSoma;

   printf("\nDigite um numero inteiro: ");
   umInteiro = LeInteiro();

   aSoma = SomaAteN(umInteiro);

   printf( "\nSoma de 1 ate' %d: %d\n", umInteiro, aSoma );

   return 0;
}
```

Nesse programa, o resultado da chamada da função `SomaAteN()` é atribuído à variável `aSoma` na expressão:

```
aSoma = SomaAteN(umInteiro);
```

Mas nem toda função retorna um valor que possa ser usado como operando numa expressão. Quando se deseja que uma função não retorne nenhum valor, utiliza-se o tipo primitivo void como tipo de retorno. Quando o tipo de retorno de uma função é void, uma chamada dela não pode fazer parte de uma expressão (incluindo atribuição). Isto é, uma função com tipo de retorno void só pode ser chamada isoladamente numa instrução. Por exemplo, considere o programa abaixo:

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

void ApresentaMenu(void)
{
   printf( "\nAs opcoes deste programa sao:\n\n"
           "\tOpcao 1\n"
           "\tOpcao 2\n"
           "\tOpcao 3\n"
           "\tOpcao 4\n"
         );
}

int main(void)
{
   ApresentaMenu();

   return 0;
}
```

Nesse programa, a chamada da função `ApresentaMenu()` aparece sozinha numa linha de instrução. Seria ilegal, por exemplo, incluí-la numa expressão, como: `2*ApresentaMenu()`.

Em padrões de C anteriores a C99, não era obrigatório incluir numa definição de função seu tipo de retorno e, quando esse não era incluído, o tipo assumido pelo compilador era **int**. A partir do padrão C99, tornou-se obrigatório a indicação do tipo de retorno de qualquer função. Portanto, para evitar problemas, siga sempre o conselho:

Recomendação *Nunca omita o tipo de retorno de uma função.*

Parâmetros de uma Função

Um **parâmetro** é semelhante a uma variável, pois ambos possuem nomes e são associados a espaços em memória. Além disso, uma **declaração de parâmetros** no cabeçalho de uma função é similar a um conjunto de definições de variáveis, mas iniciações ou abreviações não são permitidas numa declaração de parâmetros. Por exemplo, os seguintes cabeçalhos seriam considerados ilegais:

```
int F(int x = 5) /* Iniciação não é permitida */
void G(double x, y) /* Abreviação não é permitida */
```

Nesse último caso, o correto seria declarar os parâmetros como:

```
void G(double x, double y)
```

Quando a função não possui parâmetros, pode-se deixar vazio o espaço entre parênteses ou preencher esse espaço com a palavra-chave **void**. Essa segunda opção é mais recomendada, pois torna a declaração mais legível e facilita a escrita de alusões à função (v. **Seção 5.6**).

Nomes de parâmetros devem seguir as mesmas recomendações de estilo apresentadas na **Seção 3.8** para nomes de variáveis.

Nome de uma Função

Um nome de uma função é um identificador como outro qualquer (v. **Seção 3.2**), mas, em termos de estilo de programação, é recomendado que a escolha do nome de uma função siga as seguintes normas:

- **O nome de uma função deve refletir aquilo que a função faz ou produz**. Funções que retornam um valor devem ser denominadas pelo nome do valor retornado. Por exemplo, uma função que calcula o fatorial de um número deve ser denominada apenas como *Fatorial* (e não, por exemplo, *CalculaFatorial* ou, pior, *RetornaFatorial*). Por outro lado, o nome de uma função que não retorna nada (i.e., cujo tipo de retorno é **void**) deve indicar o tipo de processamento que a função efetua. Por exemplo, uma função que ordena uma lista de nomes pode ser denominada de forma sucinta como *OrdenaNomes* ou, se for necessário ser mais prolixo, *OrdenaListaDeNomes*. Também, funções que retornam um dentre dois valores possíveis (p. ex., *sim/não* ou *verdadeiro/falso*) podem ser nomeadas começando com *Eh* [representando *é* — p. ex., `EhValido()`] ou *Sao* [representando *são* — p. ex., `SaoIguais()`].
- **Cada palavra constituinte do nome de uma função deve começar por letra maiúscula e ser seguida por letras minúsculas**. Procedendo assim, não é necessário usar subtraços para separar as palavras. O uso de convenções diferentes para denominar diferentes componentes de um programa facilita a rápida identificação visual de cada componente, conforme já foi indicado antes.

5.4.2 Corpo de Função

O corpo de uma função contém declarações e instruções necessárias para implementar a função. O corpo de uma função deve ser envolvido por chaves, mesmo quando ele contém apenas uma instrução. Por exemplo:

```
int ValorAbsoluto(int x)
{
   return x < 0 ? -x : x;
}
```

A função `ValorAbsoluto()` contém apenas uma instrução, mas, mesmo assim, seu corpo deve estar entre chaves.

Em termos de estilo de programação, é recomendado usar endentação para ressaltar que uma instrução ou declaração faz parte de uma função. Por exemplo, a endentação da instrução **return** em relação ao cabeçalho da função `ValorAbsoluto()` deixa claro que essa instrução pertence à referida função.

Além de instruções, o corpo de uma função pode conter diversos tipos de declarações e definições, mas o mais comum é que ele contenha apenas definições de variáveis. Essas variáveis auxiliam o processamento efetuado pela função e não são reconhecidas fora do corpo da função (v. **Seção 5.10**). Considere, por exemplo, o seguinte programa:

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

int Fatorial(int n)
{
   int i, produto = 1;

   if (n < 0) {
      return 0;
   }

   for (i = 2; i <= n; ++i) {
      produto = produto * i;
   }

   return produto;
}
int main(void)
{
      /* ERRO: A variável 'produto' não é válida aqui */
   printf("produto = %d\n", produto);

   printf("O fatorial de 5 e' %d\n", Fatorial(5));

   return 0;
}
```

Esse programa não consegue ser compilado porque a variável `produto` é utilizada na instrução `produto = 0` no corpo da função **main**(). Como essa variável foi definida no corpo da função `Fatorial()`, ela não poderia ser referenciada fora dele.

Em C, funções não podem ser aninhadas (como em Pascal, por exemplo). Isto é, uma função não pode ser definida dentro do corpo de outra função.

O corpo de uma função pode ser vazio, mas isso só faz sentido durante a fase de desenvolvimento de um programa. Ou seja, o corpo de uma função pode permanecer vazio durante algum tempo de desenvolvimento de um programa, quando se deseja adiar a implementação da função. Nessa situação, o corpo deverá ser 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 UmaFuncao(int i)
{
   /* Corpo vazio a ser preenchido posteriormente */
}
```

5.4.3 Instrução return

A execução de uma instrução **return** encerra imediatamente a execução de uma função e seu uso depende do fato de a função ter tipo de retorno **void** ou não.

Funções com Tipo de Retorno void

Uma função com tipo de retorno **void** não precisa ter em seu corpo nenhuma instrução **return**. Quando uma função não possui instrução **return**, seu encerramento se dá, naturalmente, após a execução da última instrução no corpo da função. Mas, uma função com tipo de retorno **void** pode ter em seu corpo uma ou mais instruções **return** e isso ocorre quando a função precisa ter mais de uma opção de encerramento. Nesse caso, a função será encerrada quando uma instrução **return** for executada ou quando o final da função for atingido; i.e., a primeira alternativa de término que vier a ocorrer encerra a função.

Funções com tipo de retorno **void** não podem retornar nenhum valor. Isto é, numa função desse tipo, uma instrução **return** não pode vir acompanhada de nenhum valor. Ou seja, uma função com tipo de retorno **void** só pode usar uma instrução **return** como:

return;

Funções com Tipo de Retorno Diferente de void

Toda função cujo tipo de retorno não é **void** *deve* ter em seu corpo pelo menos uma instrução **return** que retorne um valor compatível com o tipo definido no cabeçalho da função (v. **Seção 5.4.1**). Para retornar um valor, utiliza-se uma instrução **return** seguida do valor que se deseja que seja o resultado da invocação da função. Esse valor pode ser representado por uma constante, variável ou expressão. Portanto a sintaxe mais geral de uma instrução **return** é:

return *expressão*;

O efeito da execução de uma instrução **return** no corpo de uma função é causar o final da execução da função com o consequente retorno, para o local onde a função foi chamada, do valor resultante da expressão que acompanha **return**.

Pode haver mais de uma instrução **return** no corpo de uma função, mas isso não quer dizer que mais de um valor pode ser retornado a cada execução da função. 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 parâmetros recebidos por ela. A primeira instrução **return** executada causará o término da execução da função e o retorno do respectivo valor associado a essa instrução. Como exemplo, considere a seguinte função:

```
int Fatorial(int n)
{
   int i, produto = 1;

   if (n < 0) {
      return 0;
   }

   for (i = 2; i <= n; ++i) {
      produto = produto * i;
   }

   return produto;
}
```

Nesse exemplo, a função retornará `0` quando o valor de seu parâmetro for negativo; caso contrário, ela retornará o fatorial dele. O fato de essa função retornar zero quando o parâmetro recebido é negativo não significa que o programador acredita que o fatorial de um número negativo seja zero. De fato, chamar essa função com um número negativo constitui uma condição de exceção (v. **Seção 1.3**). Ou seja, a chamada da função com um valor para o qual não existe fatorial transgride um pressuposto da função e o tratamento adequado seria abortar o programa para que o programador tentasse descobrir por que ela fora chamada indevidamente. Mas, lidar adequadamente com esse **tratamento de exceção** está além do escopo deste livro.

Toda função com tipo de retorno diferente de **void** deve retornar um valor compatível com o tipo de retorno declarado em seu cabeçalho. Quando o tipo de retorno declarado no cabeçalho da função não coincide com o tipo do valor resultante da expressão que acompanha **return** e é possível uma conversão entre esses tipos, o valor retornado será implicitamente convertido no tipo de retorno declarado. Por exemplo, na função a seguir:

```
int UmaFuncao(void)
{
   return  2*3.14;
}
```

o resultado da expressão `2*3.14` (i.e., `6.28`) será implicitamente convertido em **int** que é o tipo de retorno declarado. Assim, o valor retornado ao local onde essa função for chamada será `6`. É importante observar que, assim como ocorre com outras formas de conversão implícita, não há arredondamento (v. **Seção 3.10.1**).

5.4.4 Projetos de Funções

Uma função deve ser projetada de modo que, idealmente, ela realize uma única tarefa. Alguns indicativos de que uma função tenha sido especificada para realizar múltiplas tarefas são os seguintes:

- Você tem dificuldade para batizar a função com um nome sucinto e significativo (v. **Seção 5.4.1**).
- A função é longa demais (p. ex., ela ocupa mais de uma tela no editor de programas). Quanto menor e mais específica for uma função, mais ela promove o reúso de código.
- A função utiliza parâmetros demais. Uma função contendo mais de cinco parâmetros pode ser considerada exagerada nesse aspecto.

Se você se deparar com algum desses casos mencionados, tente dividir a função em funções menores.

Outros erros comuns de definição de funções cometidos por programadores inexperientes são:

- A função lê dados por meio do teclado, quando esses dados deveriam ser parâmetros de entrada da função. Normalmente, uma função lê dados introduzidos pelo usuário apenas quando essa é a finalidade precípua da função.
- A função escreve informação na tela, quando, de fato, essa informação deveria ser um parâmetro de saída ou um valor retornado pela função. Normalmente, uma função apresenta dados na tela apenas quando essa é sua única finalidade ou quando precisa apresentar prompt para leitura de dados (v. item anterior).

5.5 Chamadas de Funções

Chamar uma função significa transferir o fluxo de execução 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 ele existe, mas não há interesse em utilizá-lo. Por exemplo, a função **printf()** retorna um valor do tipo **int** que informa o número de caracteres escritos na tela, de modo que, por exemplo, ela pode ser usada assim:

```
printf("%d", x);
```

ou assim:

```
int nCaracteresEscritos = printf("%d", x);
```

No primeiro caso, o valor retornado por **printf()** é desprezado, enquanto, no segundo caso, ele é atribuído à variável `nCaracteresEscritos`.

Como mostra o último exemplo, chamadas de funções que retornam algum valor podem ser utilizadas como parte de expressões. Numa tal situação, a chamada de função sempre é avaliada antes da aplicação de qualquer outro operador.

5.5.1 Modos de Parâmetros

Parâmetros proveem o meio normal de comunicação de dados entre funções. Isto é, normalmente, uma função obtém os dados de entrada necessários ao seu processamento por meio de parâmetros e transfere dados resultantes do processamento também por meio de parâmetros. Do mesmo modo, é normal transferir o resultado de um processamento por meio de valor de retorno. O que não é normal é utilizar variáveis com escopo global ou de arquivo (v. **Seção 5.10**) para fazer essa comunicação de dados, a não ser que haja realmente um bom motivo para assim proceder.

O **modo** de um parâmetro de função refere-se ao papel que ele desempenha no corpo da função. Existem três modos de parâmetros:

- **Parâmetro de entrada**. Nesse caso, o parâmetro não é ponteiro ou, se ele for ponteiro, a função apenas consulta o valor da variável para a qual ele aponta, mas não altera esse valor. Assim, todo parâmetro que não é ponteiro é um parâmetro de entrada, mas um parâmetro que é ponteiro pode ser ou não ser um parâmetro de entrada, como será visto adiante.
- **Parâmetro de saída**. O parâmetro deve ser um ponteiro e a função apenas altera o valor da variável para a qual ele aponta, mas não consulta o valor dessa variável.
- **Parâmetro de entrada e saída**. O parâmetro deve ser um ponteiro e a função consulta e altera o valor da variável para a qual ele aponta.

Como foi descrito, um parâmetro de saída ou de entrada e saída deve, obrigatoriamente, ser declarado como ponteiro. Entretanto, um parâmetro pode ser declarado como ponteiro e ser um parâmetro de entrada (apenas). Esse último caso acontece em duas situações:

[1] A linguagem C requer que o parâmetro seja declarado como ponteiro, como é o caso quando o parâmetro representa um array (v. **Capítulo 8**).

[2] O tamanho do espaço em memória necessário para armazenar a variável para a qual o parâmetro aponta é bem maior do que o espaço ocupado por um ponteiro. É isso que tipicamente ocorre quando o parâmetro representa uma estrutura (v. **Capítulo 10**).

Saber classificar cada parâmetro de acordo com seu modo é fundamental em programação, mas essa tarefa nem sempre é trivial. Portanto pratique bastante.

5.5.2 Passagem de Parâmetros

Os parâmetros que aparecem numa definição de função são denominados **parâmetros formais**, enquanto os parâmetros 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 **regras de ligação**) entre parâmetros reais e formais. Em C, duas regras de casamento devem ser obedecidas para que uma chamada de função 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 definição da função, também não deve haver parâmetros reais numa chamada dela; se houver um parâmetro formal na definição da função, deve haver um parâmetro real numa respectiva chamada e assim por diante.

[2] **Os respectivos parâmetros reais e formais 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 casos nos quais 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. Considere como exemplo o seguinte programa:

```
#include <stdio.h>
int SomaInts(int a, int b)
{
   int soma;
   soma = a + b;
   return soma;
}
int main(void)
{
   double x = 2.5, y = 3.5, resultado;
   resultado = SomaInts(x, y);
   printf( "\n%f + %f = %f\n", x, y, resultado);
   return 0;
}
```

Esse programa apresenta o seguinte resultado:

```
2.500000 + 3.500000 = 5.000000
```

Esse resultado é obtido conforme descrito a seguir. Na chamada de `SomaInts()`:

```
resultado = SomaInts(x, y);
```

os parâmetros reais `x` e `y` são do tipo **double**. Como valores do tipo **double** podem ser convertidos em valores do tipo **int** (v. Seção 3.10), os valores de `x` e `y` (`2.5` e `3.5`, respectivamente) são convertidos implicitamente em **int**. Portanto os parâmetros `a` e `b` da função `SomaInts()` recebem, respectivamente, os valores `2` e `3` e o valor retornado pela função será `5` (i.e., a soma de `a` com `b`). Antes de o valor retornado pela função ser atribuído à variável `resultado` na função **main()**, ocorre uma outra conversão implícita, pois a variável `resultado` é do tipo **double** e o tipo do valor retornado pela função `SomaInts()` é **int**. Dessa vez, ocorre uma conversão de atribuição (v. Seção 3.10.1), de modo que essa variável recebe o valor `5.0`. Finalmente, a chamada de **printf()** na função **main()** é responsável pela apresentação do resultado na tela.

Conforme foi visto na Seção 5.2.2, regras rígidas de compatibilidade entre ponteiros devem ser observadas. Por exemplo, o seguinte programa:

```
#include <stdio.h>   /* Entrada e saída */
int SomaInts2(int *pa, int *pb)
{
   int soma;
   soma = *pa + *pb;
   return soma;
}
int main(void)
{
   double x = 2.5, y = 3.5, resultado;
      /* O resultado será imprevisível */
   resultado = SomaInts2(&x, &y);
```

```
   printf( "\n%f + %f = %f\n", x, y, resultado);

   return 0;
}
```

apresenta o seguinte resultado na tela:

```
2.500000 + 3.500000 = 0.000000
```

Para entender a razão do resultado aparentemente absurdo apresentado por esse programa, observe que a função `SomaInts2()` espera receber dois endereços de inteiros, mas na instrução:

```
resultado = SomaInts2(&x, &y);
```

ela é chamada com os parâmetros reais `&x` e `&y`, que são dois endereços de variáveis do tipo **double**. Como os parâmetros formais dessa função são ponteiros para **int**, quando a cada um deles é aplicado o operador de indireção o resultado é um valor do tipo **int**. Em outras palavras, na instrução da função `SomaInts2()`:

```
soma = *pa + *pb;
```

o conteúdo para o qual `pa` aponta é interpretado como sendo do tipo **int** (em vez de **double**, como deveria ser) e o mesmo ocorre com o conteúdo apontado por `pb`. Nesse exemplo específico, ocorre que ambos os respectivos conteúdos (`2.5` e `3.5`) resultam em zero quando interpretados como **int**. Isso explica o valor retornado pela função[1].

O que precipitou o resultado desastroso do último programa não foram conversões de tipos, mas sim **reinterpretações de conteúdos** de memória. Ou seja, os bits que ocupavam o espaço reservado à variável `x`, que originalmente eram interpretados como um valor do tipo **double**, passaram a ser interpretados como um valor do tipo **int**; e o mesmo ocorreu com a variável `y`. Para entender melhor a diferença entre conversão de tipos e reinterpretação de conteúdo, compile e execute o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   double  x = 2.5;
   int    *ptr = &x;
   int     y = x;

   printf( "\nValor de x interpretado como double: %f", x);
   printf( "\nValor de x convertido em int: %d", y);
   printf( "\nValor de x interpretado como int: %d", *ptr);

   return 0;
}
```

Executando esse último programa, você deverá obter como resultado:

```
Valor de x interpretado como double: 2.500000
Valor de x convertido em int: 2
Valor de x interpretado como int: 0
```

Para precaver-se contra surpresas desagradáveis, como o resultado da chamada da função `SomaInts2()`, é importante que se passem parâmetros reais dos mesmos tipos dos parâmetros formais correspondentes, ou então certifique-se de que uma dada conversão não produzirá um efeito indesejável. E, como você deve ter pressentido,

[1] Para descrever precisamente os detalhes dessa discussão, que é um tanto superficial, é necessário explorar como números inteiros e reais são representados em memória, o que está além do escopo deste livro.

o cuidado na passagem de parâmetros deve ser redobrado quando se lidam com parâmetros representados por ponteiros.

5.5.3 Simulando Passagem por Referência em C

Em algumas linguagens de programação (p. ex., Pascal), existem dois tipos de **passagens de** (ou **ligações entre**) **parâmetros**:

- **Passagem por valor**. Quando um parâmetro é passado por valor, o parâmetro formal recebe uma cópia do parâmetro real correspondente. Qualquer alteração sofrida pelo parâmetro formal no corpo da função fica restrita a essa cópia e, portanto, não é transferida para o parâmetro real correspondente. Essa é a única modalidade de passagem de parâmetros existente em C.
- **Passagem por referência**. Na passagem por referência, o parâmetro formal e o parâmetro real correspondente, que deve ser uma variável, compartilham o mesmo espaço em memória, de maneira que qualquer alteração feita pela função no parâmetro formal reflete-se na mesma alteração no parâmetro real correspondente. A linguagem C não possui passagem de parâmetros por referência.

Em linguagens que possuem essas duas modalidades de passagem de parâmetros, passagem por referência é requerida para um parâmetro de saída ou de entrada e saída, porque qualquer alteração do parâmetro formal deve ser comunicada ao parâmetro real correspondente. Por outro lado, parâmetros de entrada podem ser passados por valor ou por referência. Quer dizer, parâmetros de entrada são passados por referência por uma questão de economia de tempo e de espaço em memória, pois, quando um parâmetro é passado por valor, ele é copiado. Portanto, se o parâmetro ocupa muito espaço em memória, sua cópia poderá resultar em gasto excessivo de tempo e de espaço em memória.

Em C, estritamente falando, existe apenas passagem de parâmetros por valor, de forma que nenhum parâmetro real tem seu valor modificado como consequência da execução de uma função. Esse fato pode parecer estranho à primeira vista e uma questão que surge naturalmente é: *Como é que uma variável utilizada como parâmetro real numa chamada de função pode ter seu valor modificado em consequência da execução da função?* Ou, em outras palavras: *Como é possível existir parâmetro de saída ou entrada e saída em C se não há passagem por referência nessa linguagem?*

A resposta às questões acima é: mediante o uso de ponteiros e endereços de variáveis pode-se simular passagem por referência em C e modificar valores de variáveis, como será mostrado a seguir.

Suponha, por exemplo, que se deseje implementar uma função para trocar os valores de duas variáveis do tipo **int**. Então, um programador iniciante em C poderia ser induzido a escrever a seguinte função:

```
void Troca1(int inteiro1, int inteiro2)
{
   int aux = inteiro1; /* Guarda o valor do primeiro inteiro */

      /* A variável cujo valor foi guardado recebe o valor da  */
      /* outra variável e esta recebe o valor que foi guardado */
   inteiro1 = inteiro2;
   inteiro2 = aux;
}
```

Uma função **main()** que chama a função `Troca1()` poderia ser escrita como:

```
int main(void)
{
   int i = 2, j = 5;
```

```
    printf( "\n\t>>> ANTES da troca <<<\n"
            "\n\t    i = %d, j = %d\n", i, j );

    Troca1(i, j);

    printf( "\n\t>>> DEPOIS da troca <<<\n"
            "\n\t    i = %d, j = %d\n", i, j );

    return 0;
}
```

Após executar o programa composto pelas duas funções acima, o programador fica frustrado com o resultado obtido:

```
        >>> ANTES da troca <<<

            i = 2, j = 5

        >>> DEPOIS da troca <<<

            i = 2, j = 5
```

Note que não houve troca dos valores das variáveis, como seria esperado. Para entender melhor o que aconteceu com esse programa, acompanhe, a seguir, a sequência de eventos que ocorre durante a execução do programa.

Como qualquer programa em C que é executado sob a supervisão de um sistema operacional, o programa anterior começa sua execução na função **main**(). Essa função define duas variáveis, `i` e `j`, iniciadas, respectivamente, com `2` e `5`. Essas iniciações de variáveis causam suas alocações com os conteúdos especificados, como ilustrado na Figura 5–3.

FIGURA 5–3: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 1

Após as definições das variáveis `i` e `j`, é feita a primeira chamada de **printf**(), que causa a escrita do seguinte na tela:

```
        >>> ANTES da troca <<<

            i = 2, j = 5
```

Em seguida, é feita a seguinte chamada da função `Troca1()`:

```
Troca1(i, j);
```

Assim como variáveis, parâmetros também são alocados em memória quando uma função é chamada. A referida chamada da função `Troca1()` é perfeitamente legítima de acordo com as regras de ligação descritas na Seção 5.5.2, de modo que o parâmetro real `i` casa com o parâmetro formal `inteiro1` e o parâmetro real `j` casa com o parâmetro formal `inteiro2`. Como em C a passagem de parâmetros se dá por valor, os conteúdos dos parâmetros reais `i` e `j` são copiados, respectivamente, para os parâmetros formais `inteiro1` e `inteiro2`. Assim, a região de memória de interesse nessa discussão apresenta-se como mostra a Figura 5–4.

A função `Troca1()` define uma variável local denominada `aux`, iniciada com o valor do parâmetro `inteiro1`. Evidentemente, essa variável também é alocada em memória, de modo que, após a alocação dessa variável, as variáveis e parâmetros em uso no programa podem ser representados esquematicamente como na Figura 5–5.

Após a execução da instrução `inteiro1 = inteiro2` na função `Troca1()`, a situação em memória pode ser representada pela Figura 5–6.

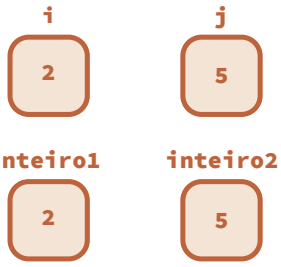

FIGURA 5–4: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 2

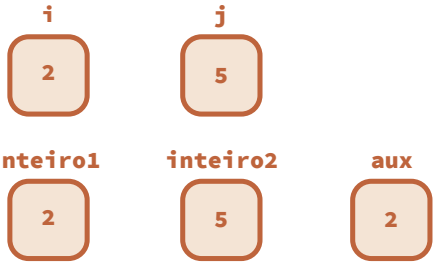

FIGURA 5–5: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 3

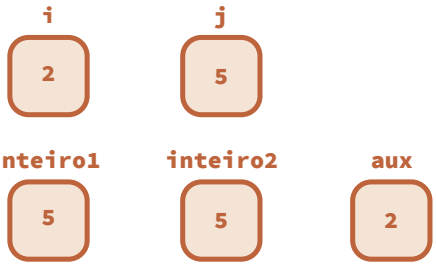

FIGURA 5–6: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 4

Finalmente, após a execução da última instrução da função `Troca1()`:

```
inteiro2 = aux;
```

a situação em memória pode ser representada como na Figura 5–7.

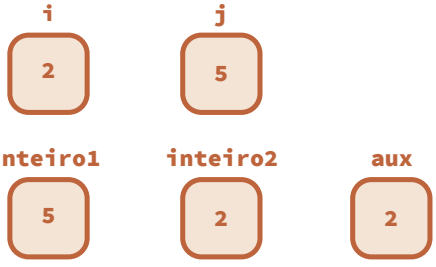

FIGURA 5–7: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 5

Como se pode observar na última ilustração, a chamada da função `Troca1()` troca os valores dos parâmetros formais `inteiro1` e `inteiro2`, mas os parâmetros reais `i` e `j` permanecem intactos. Portanto a função `Troca1()` não realiza a tarefa desejada. Por isso, a segunda chamada dessa função é responsável pela escrita da frustrante linha na tela:

```
i = 2, j = 5
```

Considere, agora, a função `Troca2()` que tem a mesma finalidade da função `Troca1()`, mas é implementada utilizando ponteiros:

```
void Troca2(int *ptrInt1, int *ptrInt2)
{
   int aux = *ptrInt1; /* Guarda o valor do primeiro inteiro que será alterado */

   /*                                                          */
   /* A variável cujo valor foi guardado recebe o valor da  */
   /* outra variável e esta recebe o valor que foi guardado */
   /*                                                          */

   *ptrInt1 = *ptrInt2;
   *ptrInt2 = aux;
}
```

Suponha que a função **main()** apresentada antes tenha sido alterada para chamar adequadamente a função `Troca2()`:

```
int main(void)
{
   int i = 2, j = 5;

   printf( "\n\t>>> ANTES da troca <<<\n"
           "\n\t    i = %d, j = %d\n", i, j );

   Troca2(&i, &j);

   printf( "\n\t>>> DEPOIS da troca <<<\n"
           "\n\t    i = %d, j = %d\n", i, j );

   return 0;
}
```

Quando executado, o programa composto pelas duas últimas funções apresenta o resultado desejado na tela:

```
        >>> ANTES da troca <<<

            i = 2, j = 5

        >>> DEPOIS da troca <<<

            i = 5, j = 2
```

Agora, acompanhe a chamada da função `Troca2()` e perceba a diferença entre o funcionamento dessa função e o funcionamento da função `Troca1()` descrito acima.

Como no exemplo anterior, no instante em que a função `Troca2()` é chamada, seus parâmetros formais são alocados em memória e cada um deles recebe uma cópia do parâmetro real correspondente. Então, nesse instante, a situação das variáveis e parâmetros do programa pode ser ilustrada na **Figura 5–8**.

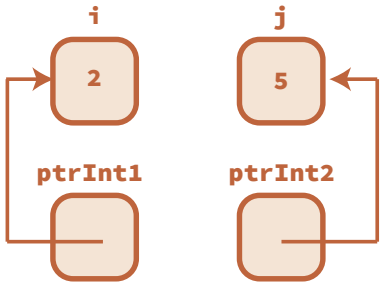

FIGURA 5–8: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 6

Observe na ilustração anterior que os parâmetros `ptrInt1` e `ptrInt2` recebem cópias dos endereços das variáveis `i` e `j`, conforme indicam as setas que emanam desses parâmetros.

Continuando com o acompanhamento da execução da função `Troca2()`, a variável `aux` é alocada e recebe o valor do conteúdo apontado pelo parâmetro `ptrInt1`, que é exatamente o valor da variável `i`. Portanto a situação nesse instante pode ser ilustrada como na Figura 5–9.

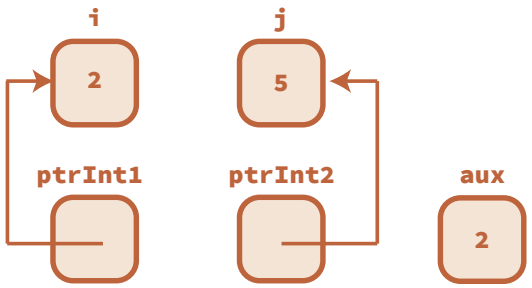

FIGURA 5–9: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 7

A próxima instrução da função `Troca2()` a ser executada é:

```
*ptrInt1 = *ptrInt2;
```

De acordo com essa instrução, o conteúdo apontado por `ptrInt1` recebe o conteúdo apontado por `ptrInt2`, o que equivale a substituir o valor da variável `i` pelo valor da variável `j` e a situação em memória passa a ser aquela mostrada na Figura 5–10.

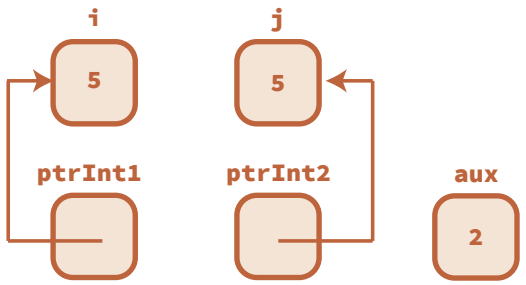

FIGURA 5–10: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 8

Finalmente, a última instrução da função `Troca2()` a ser executada é:

```
*ptrInt2 = aux;
```

Após a execução dessa instrução, o conteúdo apontado pelo parâmetro `ptrInt2` terá recebido o valor da variável `aux`, de modo que, logo antes do encerramento da função, a situação em memória é aquela ilustrada na Figura 5–11.

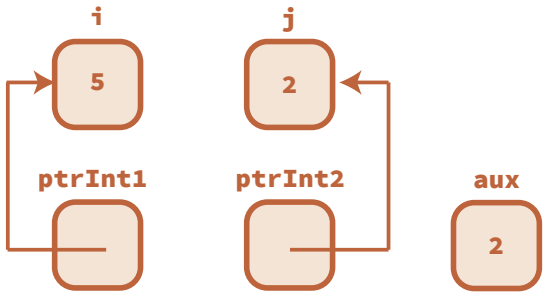

FIGURA 5–11: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 9

Ao encerramento da execução da função `Troca2()`, os espaços alocados para os seus parâmetros e para a variável local `aux` são liberados (v. Seção 5.12), restando em memória apenas as variáveis `i` e `j`, que foram alocadas na função **main()**, como se vê na Figura 5–12.

FIGURA 5–12: SIMULANDO PASSAGEM POR REFERÊNCIA EM C 10

Agora, o efeito desejado foi realmente obtido; i.e., os valores das variáveis i e j foram trocados pela função `Troca2()`.

A função `Troca2()` do último exemplo espera como parâmetros dois endereços de variáveis do tipo **int**. Pode-se, portanto, passar tanto endereços de variáveis do tipo **int**, como foi feito no último programa, quanto ponteiros para o tipo **int**. Por exemplo, a chamada da função `Troca2()` efetuada no corpo da função **main()**:

```
int main(void)
{
   int   i = 2, j = 5;
   int  *ptr = &i;
   ...
   Troca2(ptr, &j);
   ...
}
```

teria exatamente o mesmo efeito da chamada do exemplo anterior.

O problema na utilização de ponteiros em chamadas de funções ocorre quando eles não são previamente iniciados com endereços válidos. Por exemplo, suponha que, na chamada `Troca2(ptr, &j)` do último exemplo, o ponteiro `ptr` não tivesse sido iniciado com o endereço da variável `i`. Como qualquer variável definida no interior de uma função que não tem um valor explicitamente atribuído, `ptr` contém um valor indeterminado que, nesse caso, é um suposto endereço em memória. Os resultados de uma operação dessa natureza são imprevisíveis e desastrosos, pois ela modificará 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 esse 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, torne-o um ponteiro nulo (v. Seção 5.2.4). Dessa maneira, se você esquecer de atribuir um endereço válido a um ponteiro e tentar acessar o conteúdo desse endereço, o computador indicará uma operação inválida e impedirá que a execução do programa prossiga. Pode até parecer que isso não seja uma boa ideia, 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 (v. Seção 5.2.4).

5.6 Alusões e Protótipos de Funções

Uma **alusão** a uma função contém informações sobre a função que permitem ao compilador reconhecer uma chamada da função como sendo válida. Frequentemente, uma função aludida é definida num arquivo diferente daquele no qual é feita a alusão.

O formato de alusão de uma função é muito parecido com seu cabeçalho, mas, numa alusão, não é necessário especificar nomes de parâmetros (embora 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-parâmetros*);**

Por exemplo, a função `Troca2()` apresentada na **Seção 5.5.3** poderia ter a seguinte alusão:

```
extern void Troca2(int *, int *);
```

A sentença seguindo a palavra-chave **extern** numa alusão de função é conhecida como **protótipo** da função. Assim, a função `Troca2()` da **Seção 5.5.3** tem o seguinte protótipo:

```
void Troca2(int *, int *)
```

Quando incluídos numa alusão, nomes de parâmetros têm como único objetivo tornar a alusão mais clara. Eles não precisam coincidir com os nomes dos parâmetros formais na definição da função e são completamente ignorados pelo compilador. A palavra-chave **extern** é opcional, mas seu uso também é recomendado porque facilita a identificação de alusões.

Existe outra razão de natureza prática para inclusão de nomes de parâmetros em alusões de funções. Quando uma função é escrita antes de ser aludida, o programador pode criar uma alusão a ela copiando e colando o cabeçalho da função e, então, acrescentando **extern** ao início e ponto e vírgula ao final do cabeçalho colado. O caminho inverso também é facilmente percorrido: se uma alusão de função for escrita antes da definição da função, basta copiar a alusão sem **extern** e ponto e vírgula e tem-se imediatamente o cabeçalho da função.

Compiladores de C também aceitam alusões de funções sem informações sobre os tipos dos parâmetros. Isto é, o espaço entre os parênteses de uma alusão pode ser vazio e, nesse caso, tem-se uma **alusão sem protótipo**. Entretanto, essa prática não é recomendada, pois não permite que o compilador cheque se uma determinada chamada da função aludida satisfaz as regras de casamento de parâmetros. Por exemplo, pode-se aludir a função `Troca2()` da **Seção 5.5.3** como:

```
extern void Troca2();
```

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

```
Troca2(5);
```

é 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 (v. **Seção 3.18.6**).

Na **Seção 5.4.1**, afirmou-se que o uso de **void** entre os parênteses do cabeçalho de uma função era opcional. Mas, no caso de alusão, o uso de **void** não é opcional, pois uma alusão com espaço vazio entre parênteses é interpretada pelo compilador como uma alusão sem protótipo. Por outro lado, uma alusão com **void** entre parênteses deixa claro que se trata de uma alusão a uma função sem parâmetros.

5.7 Subprogramas em Linguagem Algorítmica

Do mesmo modo que um programa, um subprograma também tem, normalmente, entrada, saída e processamento. Mas, existem algumas diferenças entre programas e subprogramas:

- Um programa obtém seus dados de entrada do ambiente no qual é executado. Por exemplo, um programa interativo de console, tipicamente, obtém dados de um usuário via teclado. Por outro lado, uma função recebe dados de entrada da parte do programa que a invocou. Normalmente, esses dados são transferidos para uma função por meio de parâmetros.

- Um programa apresenta seus resultados no ambiente no qual ele é executado. Por exemplo, um programa interativo, tipicamente, apresenta os resultados que ele produz na tela. Uma função, por outro lado, transfere seus resultados para o trecho do programa que a invocou por meio de parâmetros ou de um valor de retorno.

Durante a etapa de refinamento de um algoritmo (v. **Seção 2.9.2**), se o programador se depara com um passo do algoritmo que requer muitos subpassos (refinamentos) ou que se repete em outros passos, então esse passo se torna candidato a um subalgoritmo cujo desenvolvimento será efetuado à parte. Ou seja, cada parte graúda ou que se repete no esboço de um algoritmo será um subalgoritmo; i.e., um algoritmo que faz parte de outro. Assim, se você começar a escrever o algoritmo em pseudocódigo e descobrir que um passo do algoritmo é muito complexo ou repetido, escreva esse passo como uma chamada de subalgoritmo.

Levando em consideração o que foi exposto, a linguagem algorítmica introduzida no **Capítulo 2** precisa admitir alguns acréscimos para acomodar o conceito de subalgoritmo.

Para criar subalgoritmos em linguagem algorítmica siga as seguintes etapas:

1. Na descrição do subproblema que precisa ser isolado, identifique quais são os dados de entrada, saída e entrada e saída. Essa etapa é semelhante às **Etapas 1.1** e **1.2** apresentadas na **Seção 2.9**, mas, aqui, esses dados serão os parâmetros de um subalgoritmo e não serão obtidos, por exemplo, via teclado ou apresentados na tela. Isto é, esses dados serão obtidos do algoritmo ou enviados para o mesmo algoritmo do qual o subalgoritmo faz parte.
2. Atribua nomes significativos para os parâmetros identificados na etapa anterior e classifique-os de acordo com seus modos (v. exemplos abaixo).
3. Se algum dado de saída identificado no passo 1 acima for um valor retornado pelo subalgoritmo em vez de por meio de um parâmetro de saída ou entrada e saída, especifique-o como tal, precedendo-o pela palavra *Retorno*. Se o subalgoritmo não retorna nenhum valor, escreva: *Retorno: nada* (ou algo equivalente).
4. Atribua um nome significativo ao algoritmo seguindo as recomendações apresentadas na **Seção 5.4.1**.
5. Escreva, sem muito detalhamento, os passos que você acha que são necessários para resolver o subproblema. Essa etapa é idêntica à **Etapa 1.3** discutida na **Seção 2.9.1** e deverá resultar num subalgoritmo preliminar.
6. Efetue o detalhamento do esboço do subalgoritmo obtido na etapa anterior. Essa etapa é idêntica à **Etapa 2** apresentada na **Seção 2.9.2** e deverá resultar num subalgoritmo bem detalhado.
7. Siga as demais recomendações para construção de algoritmos apresentadas no **Capítulo 2**.

Restam apenas mais dois acréscimos à linguagem algorítmica para adaptá-la ao conceito de subalgoritmo:

- Instrução **retorne**. Essa instrução encerra a execução de um subalgoritmo com o retorno de um valor, se o subalgoritmo do qual essa instrução faz parte retorna algum valor.
- Chamada do subalgoritmo. Uma chamada de subalgoritmo é realizada usando-se o nome do algoritmo acompanhado pelos parâmetros entre parênteses.

A seguir, são apresentados exemplos de algoritmos que usam subalgoritmos. Estes exemplos são meramente ilustrativos, pois, do ponto de vista pragmático, não há complexidade suficiente que justifique o uso de subalgoritmos. Na **Seção 5.11**, serão apresentados exemplos mais realistas.

Exemplo 5.1 O seguinte algoritmo principal troca os valores de duas variáveis e é semelhante àquele apresentado na **Seção 2.10.1**, mas, diferentemente daquele, este usa um subalgoritmo.

```
ALGORITMO TROCA
/* Troca os valores de duas variáveis inteiras */

ENTRADA/SAÍDA: x, y
RETORNO: nada

aux ← x
x ← y
y ← aux

ALGORITMO PRINCIPAL

leia(a, b)
Troca(a, b)
escreva(a, b)
```

FIGURA 5–13: ALGORITMO PRELIMINAR DE TROCA DE VARIÁVEIS

Exemplo 5.2 O exemplo de algoritmo a seguir calcula o fatorial de um número inteiro não negativo.

```
ALGORITMO FATORIAL
/* Calcula o fatorial de um número inteiro */

ENTRADA: n
RETORNO:
  ■ 0, se n < 0
  ■ O fatorial de n, em caso contrário

inteiro i, fat

se (n < 0)
  retorne 0 /* Indicação de erro */

fat ← 1
i ← 2

enquanto (i <= n) faça
  fat ← fat*i
  i ← i + 1

retorne fat

ALGORITMO PRINCIPAL

leia(x)

y ← Fatorial(x)

escreva(y)
```

FIGURA 5–14: ALGORITMO PRELIMINAR DE CÁLCULO DE FATORIAL

5.8 Interação Dirigida por Menus

Na Seção 4.6.3, foi apresentado um esboço de interação com usuário usando menus de opções. Resumidamente, este tipo de interação consiste no seguinte:

1. O programa apresenta um menu de opções ao usuário.
2. O programa solicita ao usuário que escolha uma das opções oferecidas.
3. O programa lê a opção escolhida pelo usuário.
4. Se a opção escolhida pelo usuário corresponder àquela especificada como saída do programa, o programa apresenta uma mensagem de despedida e é encerrado.
5. Se a opção escolhida pelo usuário não for aquela que representa encerramento do programa, executa-se a operação correspondente à opção escolhida e retorna-se ao passo 1 acima.

As seções a seguir mostram como implementar interação dirigida por menu.

5.8.1 Apresentação do Menu

A apresentação do menu é o passo mais simples de um programa dirigido por menus, porque ele consiste em uma ou mais chamadas da função **printf()**, como mostra a função `ApresentaMenu()` a seguir:

```
void ApresentaMenu(void)
{
   printf( "\n\n*** Opcoes ***\n"
           "\n1 - Opcao 1"
           "\n2 - Opcao 2"
           "\n3 - Opcao 3"
           "\n4 - Opcao 4"
           "\n5 - Encerra o programa\n" );
}
```

O menu de opções exibido pela função `ApresentaMenu()` é meramente ilustrativo, mas vale ressaltar que, quaisquer que sejam as opções apresentadas, deve haver uma que corresponda ao encerramento do programa. Também, faz sentido que essa opção de encerramento seja a última do menu.

5.8.2 Leitura de Opção

A maneira mais simples de efetuar a leitura da opção escolhida pelo usuário é por meio da função `LeOpcao()` da biblioteca LeituraFacil (v. Seção 3.13.2). Para usar essa função, o programador deve passar como parâmetro um string constante contendo todas as opções consideradas válidas. O valor retornado pela função será sempre um dos caracteres que constam desse string, como mostra o exemplo abaixo:

```
#include <stdio.h>   /* Entrada e saída */
#include "leitura.h" /* LeituraFacil    */

int main(void)
{
   int op;

   printf("\nDigite A, B ou C: ");
   op = LeOpcao("ABC");

   printf("\nVoce digitou: %c\n", op);

   return 0;
}
```

Como exemplo de execução desse programa, considere:

```
Digite A, B ou C: x
        >>> Opcao incorreta. Tente novamente
        > a
        >>> Opcao incorreta. Tente novamente
        > A
Voce digitou: A
```

Há dois fatos notáveis nesse exemplo de execução:

[1] A função `LeOpcao()` aceita como opção válida apenas um dos caracteres que constam no string constante recebido como parâmetro.

[2] Para ser simpático ao usuário, o programa deveria aceitar como válidas não apenas as opções representadas pelos caracteres `'A'`, `'B'` e `'C'`, como também os caracteres `'a'`, `'b'` e `'c'`.

Assim, para ser mais amigável ao usuário, a função `LeOpcao()` deveria ser chamada usando-se como parâmetro o string `"ABCabc"`, como mostra o programa adiante:

```
#include <stdio.h>   /* Entrada e saída */
#include "leitura.h" /* LeituraFacil    */

int main(void)
{
   int op;

   printf("\nDigite A, B ou C: ");
   op = LeOpcao("ABCabc");

   printf("\nVoce digitou: %c\n", op);

   return 0;
}
```

Exemplo de execução desse último programa:

```
Digite A, B ou C: a
Voce digitou: a
```

Como se pode constatar, o último programa considera como válida não apenas a opção representada pelo caractere `'A'`, como também aquela representada por `'a'`. Deve-se ainda ressaltar que a ordem dos caracteres que representam opções válidas no string usado com a função `LeOpcao()` não faz nenhuma diferença. Por exemplo, a chamada `LeOpcao("AB")` tem o mesmo efeito que `LeOpcao("BA")`.

5.8.3 Solicitando Confirmação Sim/Não

Uma situação muito comum em interação com o usuário é aquela na qual o programa solicita confirmação antes de executar uma operação. Isto é, nessas situações, o programa pergunta se o usuário deseja prosseguir ou não com a execução da operação. Obviamente, a função `LeOpcao()` pode ser utilizada com essa finalidade como mostra o programa a seguir:

```
int main(void)
{
   int op;

   printf("\n\t>>> Em seguida sera' executada tal operacao."
             "\n\t>>> Deseja continuar? ");
```

```
   op = LeOpcao("sSnN");

   if (op == 's' || op == 'S') {
      printf("\n\t>>> Operacao executada\n");
   } else {
      printf( "\n\t>>> A operacao NAO foi executada\n" );
   }
   return 0;
}
```

Mas, essa tarefa comum de solicitar confirmação pode ser simplificada definindo-se uma função como a função `LeOpcaoSimNao()` abaixo:

```
/****
 *
 * LeOpcaoSimNao(): Lê uma opção do tipo sim/não
 *
 * Parâmetros: Nenhum
 *
 * Retorno: 1, se a opção for 's' ou 'S'
 *          0, se a opção for 'n' ou 'N'
 *
 ****/
int LeOpcaoSimNao(void)
{
   int op;

   op = LeOpcao("sSnN");

   if (op == 's' || op == 'S') {
      return 1;
   }

   return 0;
}
```

Utilizando a função `LeOpcaoSimNao()`, a função **main()** apresentada acima pode ser escrita de modo mais sucinto como:

```
int main(void)
{
   printf("\n\t>>> Em seguida sera' executada tal operacao."
             "\n\t>>> Deseja continuar? ");
   if (LeOpcaoSimNao()) {
      printf("\n\t>>> Operacao executada\n");
   } else {
      printf( "\n\t>>> A operacao NAO foi executada\n" );
   }
   return 0;
}
```

5.8.4 Interação com Laço Infinito

Para completar uma interação dirigida por menu, é necessária uma função **main()** contendo um laço infinito que, repetidamente, apresenta o menu para o usuário, lê a opção escolhida por ele e executa a ação correspondente à opção. Na função **main()** a seguir, executar a operação correspondente à opção escolhida pelo usuário significa exibir uma mensagem, exceto quando a opção escolhida é aquela que encerra o programa.

```
int main(void)
{
   int opcao;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa faz isto, isso e aquilo."
           "\n\t>>> Escolha a opcao 5 para encerra-lo.\n" );

      /* O laço encerra quando o     */
      /* usuário escolhe a opção 'E' */
   while (1) {
      ApresentaMenu();

         /* Lê a opção do usuário */
      printf("\n\t>>> Escolha sua opcao: ");
      opcao = LeOpcao("12345");

         /* Se o usuário escolher a  */
         /* opção 5, encerra  o laço */
      if (opcao == '5') {
         break; /* Encerra o laço while */
      }

         /* Processa as demais opções */
      switch (opcao) {
         case '1':
            Opcao1();
            break;
         case '2':
            Opcao2();
            break;
         case '3':
            Opcao3();
            break;
         case '4':
            Opcao4();
            break;
         default:
            printf("\nEste programa contem um erro!!!");
            break;
      } /* switch */
   } /* while */

      /* Despede-se do usuário */
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

As funções `Opcao1()`, `Opcao2()`, `Opcao3()` e `Opcao4()` chamadas na função **main**() acima, apenas exibem uma sentença informando o usuário qual foi a opção escolhida. Por exemplo, a função `Opcao1()` é definida como:

```
void Opcao1(void)
{
   printf("\n\t>>> Voce escolheu a opcao 1\n");
}
```

Na prática, essas últimas funções mencionadas seriam substituídas por implementações de operações correspondentes às respectivas opções.

Cuidado: É importante notar que as opções da instrução **switch-case** da função **main()** acima são os dígitos `'1'` (e não o número inteiro `1`), `'2'` (e não o número inteiro `2`) etc. Isso significa, por exemplo, que se o trecho da instrução **switch-case**:

```
case '1':
   Opcao1();
   break;
```

for escrito como (note a ausência de apóstrofos):

```
case 1:
   Opcao1();
   break;
```

não haverá casamento quando o usuário escolher a opção `'1'` do menu.

Na **Seção 5.11.9**, será apresentado um programa completo que usa interação dirigida por menu e executa quatro operações aritméticas fundamentais com números inteiros.

5.9 Duração de Variáveis

Uma variável é **alocada** quando a ela se associa um espaço exclusivo em memória. Uma variável é **liberada** quando ela deixa de estar associada a qualquer espaço em memória. Enquanto uma variável permanece alocada, o espaço ocupado por ela jamais poderá ser ocupado por outra variável. Por outro lado, a partir do instante em que uma variável é liberada, o espaço antes ocupado por ela pode ser alocado para outras variáveis ou parâmetros.

Duração de uma variável é o intervalo de tempo decorrido entre sua alocação e sua liberação. De acordo com essa propriedade, variáveis são classificadas em duas categorias:

- Variável de **duração fixa**, que permanece alocada no mesmo espaço em memória durante toda a execução do programa.
- Variável de **duração automática**, que permanece alocada apenas durante a execução do bloco no qual ela é definida.

5.9.1 Duração Automática

Uma variável de duração automática pode ser definida apenas no corpo de uma função ou no interior de um bloco de instruções. Como blocos de instruções existem apenas dentro de funções, conclui-se que não existe variável de duração automática definida fora do corpo de uma função.

Na ausência de indicação explícita (v. **Seção 5.9.1**), toda variável definida dentro de um bloco tem duração automática. Uma variável com esse tipo de duração é alocada quando o bloco que contém sua definição é executado e é liberada quando encerra a execução do mesmo bloco. Portanto é possível que uma variável de duração automática ocupe diferentes posições em memória cada vez que o bloco que contém sua definição é executado. Assim, não existe nenhuma garantia de que uma variável de duração automática assuma o mesmo valor entre uma execução e outra do referido bloco.

A definição de uma variável de duração automática pode ser prefixada com a palavra-chave **auto**, mas essa palavra-chave raramente é utilizada porque ela é sempre redundante. Isto é, auto é incapaz de alterar a duração de uma variável.

5.9.2 Duração Fixa

Uma variável de duração fixa é alocada quando a execução do programa que contém sua definição é iniciada e permanece associada ao mesmo espaço de memória até o final da execução do programa.

Toda variável definida fora de qualquer função tem duração fixa, mas uma variável definida dentro de um bloco também pode ter duração fixa, desde que sua definição seja prefixada com a palavra-chave **static**.

No programa esquematizado abaixo, as variáveis `varFixa1` e `varFixa2` têm duração fixa, enquanto a variável `varAuto` tem duração automática.

```
#include <stdio.h>

int varFixa1;

void F(void)
{
   int        varAuto;
   static int varFixa2;
   ...
}

int main(void)
{
   ...
}
```

Nesse último exemplo, a variável `varFixa1` tem duração fixa porque foi definida fora de qualquer função e a variável `varFixa2` tem essa mesma duração porque sua definição usa **static**. A variável `varAuto` é definida no corpo de uma função e não usa **static**; portanto, ela tem duração automática.

Uma variável local de duração fixa é usada quando se deseja preservar seu valor entre uma chamada e outra da função que contém sua definição. Considere, por exemplo, o seguinte programa:

```
#include <stdio.h>

void QuantasVezesFuiChamada(void)
{
    static int contador = 0;

    ++contador;

    printf( "Esta funcao foi chamada %d vez%s\n",
            contador, contador == 1 ? "" : "es" );
}
int main(void)
{
   int i;

   for (i = 0; i < 3; ++i) {
      QuantasVezesFuiChamada();
   }

   return 0;
}
```

Quando executado, esse programa produz o seguinte resultado:

```
Esta funcao foi chamada 1 vez
Esta funcao foi chamada 2 vezes
Esta funcao foi chamada 3 vezes
```

A função `QuantasVezesFuiChamada()` do último programa apresenta o número de vezes que ela é chamada. A contagem é feita pela variável `contador` que tem duração fixa. Se essa variável tivesse duração automática, a referida contagem não seria possível, como mostra o seguinte programa:

```
#include <stdio.h>

void QuantasVezesFuiChamadaErrada(void)
{
    int contador = 0;

    ++contador;

    printf( "Esta funcao foi chamada %d vez%s\n",
            contador, contador == 1 ? "" : "es" );
}

int main(void)
{
   int i;

   for (i = 0; i < 3; ++i) {
      QuantasVezesFuiChamadaErrada();
   }
   return 0;
}
```

Diferentemente do programa anterior, esse último programa resulta na seguinte saída:

```
Esta funcao foi chamada 1 vez
Esta funcao foi chamada 1 vez
Esta funcao foi chamada 1 vez
```

Note que a única diferença substancial entre esse último programa e o anterior é que a variável `contador` tem duração fixa na função `QuantasVezesFuiChamada()` e duração automática na função `QuantasVezesFuiChamadaErrada()`.

5.9.3 Iniciação de Variáveis de Acordo com a Duração

Como mostram os exemplos apresentados na Seção 5.9.2, uma variável de duração fixa é iniciada apenas uma vez, enquanto uma variável de duração automática é iniciada sempre que o bloco que contém sua definição é executado. Se você ainda não notou a diferença, considere, por exemplo, o seguinte programa:

```
#include <stdio.h>

void Incrementa(void)
{
    int          i = 1;
    static  int  j = 1;

    i++;
    j++;

    printf("Valor de i = %d\t\t Valor de j = %d\n", i, j);
}

int main(void)
{
   int i;

   for (i = 0; i < 5; ++i) {
      Incrementa();
   }
   return 0;
}
```

O resultado de execução desse programa é:

```
Valor de i = 2           Valor de j = 2
Valor de i = 2           Valor de j = 3
Valor de i = 2           Valor de j = 4
Valor de i = 2           Valor de j = 5
Valor de i = 2           Valor de j = 6
```

Os resultados apresentados pela função `Incrementa()` são consequências das definições de `i` e `j` no corpo da função. A variável `i` tem duração automática e, portanto, é alocada e iniciada cada vez que a função é chamada. Por outro lado, a variável `j` tem duração fixa e é alocada e iniciada apenas uma vez. A cada chamada da função `Incrementa()`, é utilizado o valor atual da variável `j` que é mantido entre uma chamada e outra.

Outra importante diferença entre variáveis de duração fixa e automática é que, na ausência de iniciação explícita, variáveis de duração fixa são iniciadas com zero. Por outro lado, variáveis de duração automática *não* são automaticamente iniciadas. Isto é, uma variável de duração automática que não é explicitamente iniciada recebe, quando é alocada, um valor indefinido, que corresponde ao conteúdo encontrado no espaço alocado para ela.

No caso de variáveis de duração fixa, não são permitidas iniciações envolvendo expressões que não sejam constantes (i.e., que não possam ser resolvidas durante o processo de compilação). Essa exigência não se aplica ao caso de variáveis de duração automática. Ou seja, no caso de variáveis de duração automática, podem-se incluir variáveis numa expressão de iniciação, desde que essas variáveis já tenham sido previamente declaradas. Por exemplo:

```
{
   int i = 1;
   int j = 2*i + 7; /* Legal: j é de duração automática */
                    /* e i já é conhecida neste ponto   */
   static int k = i; /* ILEGAL: k é de duração fixa e sua */
                     /* iniciação deve ser determinada    */
                     /* em tempo de compilação            */
   static int n = sizeof(i); /* Legal: o valor de sizeof(i) */
                             /* pode ser determinado em     */
                             /* tempo de compilação         */
}
```

A Tabela 5–1 resume as diferenças entre variáveis de duração fixa e automática com respeito a iniciação.

Variável de Duração Fixa	Variável de Duração Automática
Iniciada implicitamente com zero	Não tem iniciação implícita
Iniciação deve ser resolvida em tempo de compilação	Iniciação pode ser resolvida em tempo de execução
Iniciada uma única vez	Pode ser iniciada várias vezes

Tabela 5–1: Iniciações de Variáveis de Duração Automática e Fixa (Comparação)

5.10 Escopo

Escopo de um identificador refere-se aos locais de um programa onde o identificador é reconhecido como válido. Em C, escopos podem ser classificados em quatro categorias, que serão descritas a seguir.

5.10.1 Escopo de Programa

Um identificador com **escopo de programa** é reconhecido em todos os arquivos e blocos que compõem o programa. Apenas identificadores que representam variáveis e funções podem ter esse tipo de escopo. Variáveis e funções com escopo de programa são denominadas **variáveis globais** e **funções globais**, respectivamente.

Qualquer variável declarada fora de funções tem escopo de programa, a não ser que ela seja precedida pela palavra-chave **static** (ver abaixo). Qualquer função que não seja precedida por **static** também tem esse tipo de escopo.

Como exemplo de variáveis e funções com escopo de programa, considere o seguinte esboço de programa:

```
#include <stdio.h>
...
double umDouble;

void MinhaFuncao(void)
{
   ...
}

int main(void){
   ...
   return 0;
}
```

Nesse esboço de programa, a variável `umDouble` e a função `MinhaFuncao()` têm escopo de programa.

5.10.2 Escopo de Arquivo

Um identificador com **escopo de arquivo** tem validade a partir do ponto de sua declaração até o final do arquivo no qual ele é declarado. Variáveis definidas fora de funções cujas definições sejam precedidas pela palavra-chave **static** têm esse tipo de escopo. Funções cujos cabeçalhos sejam qualificados com **static** também têm escopo de arquivo.

Usada nesse contexto, a palavra-chave **static** não tem o mesmo significado visto na **Seção 5.9.2**. Isto é, aqui, **static** refere-se a definição de escopo de variáveis e funções, e não a duração de variáveis como antes. Em qualquer circunstância, uma variável declarada fora de uma função tem duração fixa (quer ela venha acompanhada de **static** ou não). O significado de **static** aqui é o de delimitar o escopo de uma variável ao arquivo no qual ela é definida. Ou seja, sem ser qualificada com **static**, a variável é tratada como uma variável global. Esse mesmo significado de **static** é utilizado para delimitar escopos de funções.

Na prática, não há sensível diferença entre variáveis com escopo de arquivo e variáveis com escopo de programa quando se lidam com programas monoarquivos. E o mesmo é verdadeiro com respeito a funções. Isto é, a diferença entre esses dois tipos de escopos só é crucial quando se lidam com programas multiarquivos ou módulos de biblioteca (v. **Apêndice B**).

Uma variável com escopo de arquivo é útil quando existem várias funções num arquivo que a utilizam. Então, em vez de passar essa variável como parâmetro para as várias funções do arquivo, atribui-se a ela escopo de arquivo, de modo que todas as funções do arquivo possam compartilhá-la. Uma variável ou função com escopo de arquivo não pode ser acessada por funções em outros arquivos que constituem um programa multiarquivo.

No esboço de programa adiante, a variável `umDouble` e a função `MinhaFuncao()` têm escopo de arquivo:

```
#include <stdio.h>
...
static double umDouble;

static void MinhaFuncao(void)
{
   ...
}
```

CONTINUA

```
int main(void){
   ...
   return 0;
}
```

Identificadores associados a tipos de dados definidos pelo programador (v. **Seção 10.2**) e identificadores associados a constantes simbólicas (v. **Seção 3.15**) possuem esse tipo de escopo.

5.10.3 Escopo de Função

Um identificador com **escopo de função** tem validade do início ao final da função na qual ele é declarado. Apenas rótulos, utilizados em conjunto com instruções **goto** (v. **Seção 4.7.3**), têm esse tipo de escopo. Rótulos devem ser únicos dentro de uma função e são válidos do início ao final dela. Exemplos de escopo de função serão apresentados mais adiante.

5.10.4 Escopo de Bloco

Um identificador com **escopo de bloco** tem validade a partir de seu ponto de declaração até o final do bloco no qual ele é declarado. Parâmetros e variáveis definidas dentro do corpo de uma função têm esse tipo de escopo. Variáveis que possuem escopo de bloco são comumente denominadas **variáveis locais**.

Não se pode ter numa mesma função um parâmetro e uma variável local com o mesmo nome, a não ser que a variável seja definida dentro de um bloco aninhado no corpo da função. Considere a seguinte função como exemplo:

```
int UmaFuncao(int x)
{
   int x = 2; /* ILEGAL */
   return x;
}
```

Nesse exemplo, o compilador considerará ilegal a definição de variável:

```
int x = 2;
```

No entanto, se essa variável for definida dentro de um bloco interno à função, sua definição será considerada perfeitamente legal, conforme mostra o seguinte programa:

```
#include <stdio.h>
int OutraFuncao(int x)
{
   {
      int x = 2; /* Perfeitamente legal */
      printf("Valor da variavel x: %d\n", x);
   }
   return x; /* Retorna o valor do parâmetro */
}
int main(void)
{
   int i;
   i = OutraFuncao(0);
   printf("Valor retornado: %d\n", i);
   return 0;
}
```

Esse programa escreve o seguinte na tela:

```
Valor da variavel x: 2
Valor retornado: 0
```

No último exemplo, o parâmetro **x** é **ocultado** dentro do bloco que contém a definição da variável **x** e, portanto, não poderá ser acessado dentro desse bloco. Este tópico será discutido em detalhes na **Seção 5.10.5**.

À primeira vista, parece ser irrelevante considerar se parâmetros de funções possuem escopo de bloco ou de função, pois, nesse caso, as definições desses escopos, aparentemente, coincidem, mas tal raciocínio é equivocado. Conforme foi visto na seção anterior, um rótulo, que sempre tem escopo de função, tem validade do início ao final da função na qual ele é declarado. Isso significa que ele tem validade mesmo em blocos internos a uma função. Por exemplo:

```
int UmaFuncao(int x)
{
   int y = 2, z = 1;

umRotulo:
   z = x + y;

   if (z < 10) {
      goto umRotulo;
   }

   {
   umRotulo: /* ILEGAL: rótulos devem ser únicos numa função */
      z = z + x;
   }

   return x;
}
```

Um programa contendo a função do último exemplo não consegue ser compilado porque dois rótulos não podem ter o mesmo identificador dentro de uma mesma função, mesmo quando um rótulo é declarado dentro de um bloco interno à função e outro é declarado fora desse bloco. Isso não ocorre com parâmetros, conforme foi visto no penúltimo exemplo.

Outra diferença entre escopo de bloco e escopo de função é que um identificador com escopo de bloco tem validade a partir do ponto onde ele é declarado, enquanto um identificador com escopo de função tem validade em todo o bloco que constitui o corpo da função; i.e., ele tem validade mesmo antes de ser declarado. O exemplo a seguir ilustra esses fatos:

```
int UmaFuncao(int x)
{
   int y = z + 1, /* ILEGAL: Uso de z fora de seu escopo */
       z = 2; /* O escopo de z começa aqui */

   if (z < 10.0)
      goto umRotulo;

umRotulo: /* O rótulo é declarado aqui, mas seu escopo */
          /* começa no início do corpo da função      */
   z = x + y;
   /* ... */

   return 0;
}
```

5.10.5 Conflitos de Identificadores

É permitido o uso de identificadores iguais em escopos diferentes. Por exemplo, duas funções diferentes podem utilizar um mesmo nome de variável local sem que haja possibilidade de conflito entre os mesmos, como mostra o seguinte exemplo esquemático:

```
void  F1( void )
{
   int  x;
   ...
}
void  F2( void )
{
   double  x;
   ...
}
```

Menos evidente é o fato de também ser permitido o uso de identificadores iguais em escopos que se sobrepõem. Nesse caso, se dois identificadores podem ser válidos num mesmo local (i.e., se há **conflito de identificadores**), o identificador cuja declaração está mais próxima do ponto de conflito é utilizada. Por exemplo:

```
#include <stdio.h>
double  x = 2.5; /* x tem escopo de programa */
int main(void)
{
   int x = 1;  /* Definição de x que vale na função */
   printf("Valor de x = %f\n", x);
   return 0;
}
```

Nesse exemplo, o escopo da variável **x** declarada como **double** abrange todo o bloco da função **main()** e essa variável poderia ser utilizada dentro dessa função se não fosse o fato de uma nova variável **x** ser declarada como **int** no corpo da mesma função. Com isso, a variável **x** declarada como **int** será aquela considerada dentro do corpo de **main()**; i.e., a variável **x** declarada como **double** deixa de ser acessível no corpo dessa função.

Se você compilar e executar o último programa apresentado como exemplo, você poderá obter como resultado:

```
Valor de x = 0.000000
```

Esse resultado deve-se ao fato de a variável usada com **printf()** ser do tipo **int** e o especificador de formato **%f** usado com ela na chamada de **printf()** ser dirigido para valores do tipo **double**. Portanto o comportamento da função **printf()** nesse caso é indefinido.

5.11 Exemplos de Programação

5.11.1 Leitura de Números Naturais

Problema: Um **número natural** é um número inteiro não negativo. (a) Escreva uma função que lê um número natural via teclado e ofereça novas chances de digitação ao usuário, caso ele não introduza um valor esperado. (b) Escreva uma função **main()** que teste a função solicitada no item (a).

Solução de (a): O algoritmo aparece na **Figura 5–15** e a função vem logo em seguida.

ALGORITMO LêNATURAL

```
Algoritmo LeNatural /* Lê um número natural */

Parâmetros: Nenhum
Retorno: O valor lido

inteiro numNatural

leia(numNatural)

enquanto (numNatural < 0) faça
  escreva("Valor invalido. Tente novamente")
  leia(numNatural)

retorne numNatural
```

FIGURA 5–15: ALGORITMO DE LEITURA DE NÚMEROS NATURAIS

```
/****
 *
 * LeNatural(): Lê um número natural (i.e., um número inteiro não negativo)
 *
 * Parâmetros: Nenhum
 *
 * Retorno: O número natural lido
 *
 ****/
int LeNatural(void)
{
   int numNatural; /* O número a ser lido */

   numNatural = LeInteiro(); /* Primeira tentativa */

   while(numNatural < 0) {
      printf( "\a\n\t>>> Valor invalido. Tente novamente <<<\n\t> " );
      numNatural = LeInteiro(); /* Faz nova tentativa */
   }

   return numNatural;
}
```

Análise: Em programação, **validar dados** significa verificar se os dados recebidos por um programa satisfazem as expectativas do programa. Laços de repetição são utilizados em leitura de dados para oferecerem novas chances ao usuário após ele ter introduzido dados considerados anormais pelo programa, como faz a função `LeNatural()`.

Solução de (b): O algoritmo aparece abaixo e a função vem logo em seguida.

ALGORITMO PRINCIPAL

```
inteiro num

escreva("Digite um numero inteiro que não seja negativo:")
num ← LeNatural

escreva("Você digitou ", num)
```

```
/****
 *
 * main(): Testa a função LeNatural()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   int num;

      /* Apresenta o programa ao usuário e explica seu funcionamento */
   printf("\n\t>>> Este programa le um numero natural e apresenta-o na tela.\n");

      /* Força o usuário a digitar um número natural */
   printf( "\n\t>>> Digite um numero inteiro que nao seja negativo:\n\t> " );
   num = LeNatural(); /* Lê o número */

      /* Apresenta o número digitado */
   printf("\n\t>>> Voce digitou %d\n", num);

   return 0;
}
```

Exemplo de execução do programa:

```
>>> Este programa le um numero natural e apresenta-o na tela.

>>> Digite um numero inteiro que nao seja negativo:
> -1

>>> Valor invalido. Tente novamente <<<
> 0

>>> Voce digitou 0
```

5.11.2 Números Primos 1

Preâmbulo: Um **número primo** é um número natural (i.e., inteiro não negativo) maior do que `1` e divisível apenas por `1` e por si próprio.

Problema: Escreva uma função, denominada `EhPrimo()`, que retorna `1` se um número natural maior do que `1` recebido como parâmetro for primo e zero em caso contrário. (b) Escreva um programa que lê números inteiros não negativos como entrada e determina se cada um deles é primo ou não. O programa deve encerrar quando o usuário digitar zero ou um.

Solução de (a): O algoritmo aparece na Figura 5–16 e a função vem logo em seguida.

Algoritmo ÉPrimo

Entrada: n

Retorno:

- 1, se n for primo
- –1, se n for menor do que dois
- 0, se n não for primo

```
inteiro i

se (n ≤ 1) então
  retorne -1

i ← 2

enquanto (i ≤ n/2) faça
  se (n%i = 0) então
    retorne 0 /* Não é primo */

  i ← 1 + i

retorne 1 /* É primo */
```

Figura 5–16: Algoritmo de Verificação de Número Primo

```
/****
 *
 * EhPrimo(): Verifica se um número inteiro maior do que um é primo ou não
 *
 * Parâmetros:
 *      n (entrada): o número que será testado
 *
 * Retorno: 1, se o número for primo
 *          0, se o número não for primo
 *         -1, se for indefinido (i.e., se n <= 1)
 *
 ****/
int EhPrimo(int n)
{
   int i;

      /* O conceito de número primo não é definido */
      /* para números inteiros menores do que dois */
   if (n <= 1) {
      return -1; /* Indefinido */
   }

      /* Verifica se o número tem algum divisor */
   for (i = 2; i < n; ++i) {
      if (!(n%i)) {
         return 0; /* Encontrado um divisor */
      }
   }

      /* Não foi encontrado nenhum divisor para o número dado. Portanto ele é primo */
   return 1;
}
```

Solução de (b): O algoritmo aparece abaixo e a função vem logo em seguida.

```
ALGORITMO PRINCIPAL
inteiro num

enquanto (verdadeiro) faça
  escreva("Digite um numero inteiro que não seja negativo:")
  num ← LeNatural

  se (num ≤ 1) então
    pare

  se (EhPrimo(num) = 1) então
    escreva(num, "é primo")
  senão
    escreva(num, "não é primo")
```

Análise: Esse algoritmo chama o subalgoritmo LeNatural apresentado na Seção 5.11.1.

```
/****
 * main(): Determina se números inteiros positivos
 *         introduzidos via teclado são primos
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int num;

      /* Apresenta o programa e explica seu funcionamento */
   printf( "\n\t>>> Este programa verifica se numeros inteiros"
           "\n\t>>> nao-negativos sao primos ou nao. Para"
           "\n\t>>> encerra-lo, digite zero ou um.\n" );

      /* O laço principal do programa encerra quando o usuário introduz 0 ou 1 */
   while (1) {
      printf("\n\t>>> Digite um numero inteiro que nao seja negativo:\n\t> ");
      num = LeNatural(); /* Lê o número */

      if (num <= 1) { /* Encerra o laço */
         break;
      }

         /* Verifica se o número é primo ou não e apresenta a mensagem apropriada */
      printf( "\n\t>>> %d %s e' primo\n", num, EhPrimo(num) ? "" : "nao" );
   }

      /* Despede-se do usuário */
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Análise: A função **main**() chama a função `LeNatural()` definida na Seção 5.11.1. Portanto, para concluir esse programa, resta incluir essa última função e os cabeçalhos necessários:

```
#include <stdio.h>   /* Entrada e saída */
#include "leitura.h" /* LeituraFacil    */
```

Exemplo de execução do programa:

```
>>> Este programa verifica se numeros inteiros
>>> nao-negativos sao primos ou nao. Para
>>> encerra-lo, digite zero ou um.

>>> Digite um numero inteiro que nao seja negativo:
> -1

>>> Valor invalido. Tente novamente <<<
> 10

>>> 10 nao e' primo

>>> Digite um numero inteiro que nao seja negativo:
> 5

>>> 5  e' primo

>>> Digite um numero inteiro que nao seja negativo:
> 0

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

5.11.3 MDC por meio de Força Bruta

Problema: (a) Escreva uma função que calcule o máximo divisor comum (MDC) de dois números inteiros positivos. (b) Escreva uma função **main()** que lê dois valores inteiros positivos como entrada, calcula o máximo divisor comum deles e apresenta o resultado na tela.

Solução de (a): O algoritmo aparece na Figura 5–17 e a função vem logo em seguida.

ALGORITMO MDC

ENTRADA: x, y

RETORNO:

- MDC de x e y, se x e y forem maiores do que 0
- 0, em caso contrário

```
inteiro divisor

se ((x ≤ 0) ou (y ≤ 0)) então
  retorne 0

se (y < x)
  divisor ← y
senão
  divisor ← x

enquanto (x%divisor ≠ 0 ou y%divisor ≠ 0) faça
  divisor ← divisor - 1

retorne divisor
```

FIGURA 5–17: ALGORITMO DE CÁLCULO DE MDC

```
/****
 * MDC(): Calcula o MDC de dois números inteiros positivos
 *
 * Parâmetros:
 *      x, y (entrada): números cujo MDC será calculado
 *
 * Retorno: O referido MDC, se x e y forem positivos. Zero, em caso contrário.
 ****/
```

```
int MDC(int x, int  y)
{
   int divisor;

      /* MDC é definido apenas para números inteiros positivos */
   if ( (x <= 0) || (y <= 0) )
      return 0; /* 0 não é um MDC válido. Este valor */
                /* indica uma condição de erro.      */

   /* O MDC de dois números é no máximo igual ao menor dos dois números. */
   /* Portanto faz-se o valor inicial do divisor o menor deles. Então,   */
   /* enquanto 'divisor' não divide ambos, reduz-se seu valor de uma     */
   /* unidade. Se os números forem primos entre si, 'divisor' atingirá   */
   /* o valor 1, que é divisor de qualquer número inteiro positivo.      */

      /* Atribui a 'divisor' o menor dos parâmetros */
   divisor = (y < x) ? y : x;

      /* Enquanto 'divisor' não assumir um valor que divida */
      /* tanto x quanto y, continua decrementando seu valor */
   while ( (x%divisor) || (y%divisor) ) {
      divisor--;
   }

   return divisor;
}
```

Solução de (b): O algoritmo aparece abaixo e a função vem logo em seguida.

ALGORITMO PRINCIPAL

```
inteiro a, b
caractere op
faça
  escreva("Digite um numero inteiro que
       não seja negativo:")
  a ← LeNatural
  escreva("Digite outro numero inteiro que
       não seja negativo:")
  b ← LeNatural
  se (a = 0 ou b = 0) então
    escreva("Não existe MDC")
  senão
    escreva("O MDC é: ", MDC(a, b))
  escreva("Deseja continuar (s/n)? ")
  leia(op)
enquanto(op = 's' ou op = 'S')
```

Análise: Esse algoritmo chama o subalgoritmo LeNatural apresentado na Seção 5.11.1.

```
/****
 * main(): Lê continuamente dois valores inteiros não negativos,
 *         calcula o MDC deles e apresenta o resultado
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
```

```
{
   int a, b; /* Dois valores que terão o MDC calculado */

      /* Apresenta o programa */
   printf("\n\t>>> Este programa calcula o MDC de"
          "\n\t>>> dois numeros inteiros positivos.\n");

      /* O laço encerra quando o usuário escolhe 'n' ou 'N' */
   do {
         /* Lê dois valores não negativos */
      printf("\n>>> Digite um numero inteiro que nao seja negativo:\n\t> ");
      a = LeNatural();

      printf("\n>>> Digite outro numero inteiro que nao seja negativo:\n\t> ");
      b = LeNatural();

         /* Se um dos dois valores for 0, informa que não há MDC. */
         /* Caso contrário, calcula o MDC e o apresenta na tela.  */
      if (!a || !b) {
         printf("\n\t>>> Nao existe MDC de: %d e %d", a, b);
      } else {
         printf("\n\t>>> O MDC e': %d\n", MDC(a, b));
      }

         /* Verifica se o usuário deseja continuar */
      printf("\n\t>>> Deseja continuar (s/n)? ");
   } while (LeOpcaoSimNao());

      /* Despede-se do usuário */
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Análise: A função **main()** chama a função `LeNatural()`, apresentada na Seção 5.11.1 e a função `LeOpcaoSimNao()`, apresentada na Seção 5.8.3.

Exemplo de execução do programa:

```
 >>> Este programa calcula o MDC de
 >>> dois numeros inteiros positivos.

>>> Digite um numero inteiro que nao seja negativo:
> 7

>>> Digite outro numero inteiro que nao seja negativo:
> 28

>>> O MDC e': 7

>>> Deseja continuar (s/n)? n

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

5.11.4 MMC

Preâmbulo: Em aritmética, o **mínimo múltiplo comum** (**MMC**) de dois números naturais `x` e `y`, denotado como `MMC(x, y)`, é o menor número natural que é divisível simultaneamente por `x` e `y`. Por definição, se `x` ou `y` for zero, `MMC(x, y)` é zero.

Problema: (a) Escreva uma função que calcula o MMC de dois números naturais. (b) Escreva uma função **main()** que lê dois números naturais, chama a função descrita no item (a) e apresenta o resultado.

Solução de (a): O algoritmo aparece na Figura 5–18 e a função vem logo em seguida.

Algoritmo MMC

Entrada: x, y

Retorno:

- O MMC de x e y, se x e y forem maiores do que 0
- 0, em caso contrário

```
inteiro n

se (x ≤ 0 ou y ≤ 0) então
  retorne 0

n ← 1

enquanto(n ≤ a*b) faça
  se (n%a = 0 && n%b = 0)
    pare

  n ← n + 1

retorne n
```

Figura 5–18: Algoritmo de Cálculo de MMC

```
/****
 * MMC(): Calcula o mínimo múltiplo comum de dois números naturais
 *
 * Parâmetros:
 *      a, b (entrada) - os dois números usados no cálculo
 *
 * Retorno: Zero, se um dos parâmetros for menor do que ou igual
 *          a zero. O MMC dos dois parâmetros, em caso contrário.
 ****/
int MMC(int a, int b)
{
   int n;

      /* Se um dos dois valores for menor do que ou igual a zero, o MMC é zero */
   if (a <= 0 || b <= 0) {
      return 0;
   }

      /* O MMC de a e b é um número entre 1 e a*b */
      /* que é divisível tanto por a quanto por b */
   for(n = 1; n <= a*b; n++) {
      if(n%a == 0 && n%b == 0) {
         break; /* Encontrado o MMC */
      }
   }
   return n;
}
```

Solução de (b): O algoritmo aparece abaixo e a função vem logo em seguida.

```
ALGORITMO PRINCIPAL
inteiro a, b

escreva("Digite um numero inteiro que
      não seja negativo:")
a ← LeNatural

escreva("Digite outro numero inteiro que
       não seja negativo:")
b ← LeNatural

se (a = 0 ou b = 0) então
  escreva("Não existe MMC")
senão
  escreva("O MMC é: ", MMC(a, b))
```

Análise: Esse algoritmo chama o subalgoritmo LeNatural apresentado na Seção 5.11.1.

```
/****
 * main(): Determina o MMC de dois números naturais
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int x, y;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa determina o minimo multiplo"
           "\n\t>>> comum de dois numeros inteiros.\n" );

   printf("\n\t>>> Digite um inteiro nao-negativo: ");
   x = LeNatural();

   printf("\t>>> Digite outro inteiro nao-negativo: ");
   y = LeNatural();

   printf("\n\t>>> MMC de %d e %d: %d\n", x, y, MMC(x, y));

   return 0;
}
```

Análise: A função **main()** chama a função `LeNatural()` apresentada na Seção 5.11.1.

Exemplo de execução do programa:

```
>>> Este programa determina o minimo multiplo
>>> comum de dois numeros inteiros.

>>> Digite um inteiro nao-negativo: 4
>>> Digite outro inteiro nao-negativo: 6

>>> MMC de 4 e 6: 12
```

5.11.5 Série de Taylor para Cálculo de Seno

Preâmbulo: A função seno pode ser expressa pela seguinte série (infinita) de Taylor:

$$sen(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

ou, equivalentemente:

$$sen(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Problema: Escreva um programa que calcula o seno de um arco em radianos usando uma aproximação da fórmula acima até o enésimo termo.

Solução:

O algoritmo da **Figura 5–19** determina o sinal de cada termo.

ALGORITMO SINALDETERMODESÉRIEDETAYLOR

ENTRADA: n

RETORNO:

- -1, se n for ímpar
- 1, se n for par

```
se (n%2 ≠ 0)
  retorne  -1
senão
  retorne  1
```

FIGURA 5–19: ALGORITMO DE DETERMINAÇÃO DE SINAL DE TERMO DE SÉRIE DE TAYLOR

O algoritmo da **Figura 5–20** calcula o 'termo de ordem n da série.

ALGORITMO TERMODESÉRIEDETAYLOR

ENTRADA: x, n

RETORNO: o valor do termo de ordem n da série

```
se (n = 0) então
  retorne x

retorne Sinal(n)*pow(x, 2*n+1)/Fatorial(2*n + 1)
```

FIGURA 5–20: ALGORITMO DE CÁLCULO DE TERMO DE SÉRIE DE TAYLOR

O algoritmo da **Figura 5–21** calcula o seno de x com n termos.

ALGORITMO SENOUSANDOSÉRIEDETAYLOR

ENTRADA: x, n

RETORNO: o seno de x

```
real   soma
inteiro i

soma ← 0.0
i ← 0

enquanto (i < n) faça
  soma ← soma + Termo(x, i)
  i ← 1 + i

retorne soma
```

FIGURA 5–21: ALGORITMO DE CÁLCULO DE SENO USANDO SÉRIE DE TAYLOR

O algoritmo principal do programa que calcula seno usando série de Taylor é mostrado abaixo.

Algoritmo Principal

```
constante PI = 3.14

escreva("Seno de ", PI/4," com 3 termos: ", Seno(PI/4, 3))
escreva("Seno de ", PI/4," com 5 termos: ", Seno(PI/4, 5))
escreva("Seno de ", PI/4," usando sin(): ", sin(PI/4))
```

Observação: O algoritmo para cálculo de fatorial foi apresentado na Seção 5.7.

O programa a seguir implementa os algoritmos discutidos acima.

```
#include <stdio.h> /* Entrada e saída */
#include <math.h>  /* Função sin()    */

#define PI 3.141592

/****
 * Fatorial(): Calcula fatorial de um número inteiro não negativo
 *
 * Parâmetros: n (entrada): número cujo fatorial será calculado
 *
 * Retorno: O fatorial de n, se n >= 0. Zero, se n < 0
 ****/
int Fatorial(int n)
{
   int i,
       produto = 1; /* Acumula o produto */

      /* Se n for negativo, retorna 0. Este valor não representa um valor de   */
      /* fatorial válido; ele significa que a função foi chamada indevidamente */
   if (n < 0) {
      return 0; /* Chamada incorreta da função */
   }

      /* Calcula: 1*2*...*(n-1)*n */
   for (i = 2; i <= n; ++i) {
      produto = produto*i;
   }

   return produto;
}

/****
 *
 * Sinal(): Determina qual é o sinal do termo de ordem n da série de Taylor de seno
 *
 * Parâmetros: n (entrada): número de ordem do termo
 *
 * Retorno: -1, se o sinal do termo for negativo;
 *          1, se o sinal do termo for positivo
 *
 ****/
int Sinal(int n)
{
      /* Se o número de ordem do termo for par, o termo é positivo e a função */
      /* retorna 1. Caso contrario, ele é negativo e a função retorna -1.     */
   return n%2 ? -1 : 1;
}
```

```
/****
 *
 * Termo(): Calcula o valor do termo de ordem n da série de Taylor de seno
 *
 * Parâmetros:
 *      x (entrada): número cujo seno está sendo calculado
 *      n (entrada): número de ordem do termo
 *
 * Retorno: O valor do termo de ordem n da série
 *
 ****/
double Termo(double x, int n)
{
      /* Se n for igual a zero, o valor do termo é x */
   if (!n) {
      return x;
   }

      /* Calcula e retorna o valor do enésimo termo */
   return Sinal(n) * pow(x, 2*n + 1) / Fatorial(2*n + 1);
}

/****
 *
 * Seno(): Calcula o seno de um número real, que representa um ângulo em radianos,
 *         usando série de Taylor
 *
 * Parâmetros:
 *      x (entrada): número cujo seno será calculado
 *      n (entrada): número de termos da série de Taylor
 *
 * Retorno: O seno calculado
 *
 ****/
double Seno(double x, int n)
{
   double soma = 0.0; /* Soma dos termos da série */
   int    i;

      /* Soma os n termos da série */
   for (i = 0; i < n; ++i) {
      soma = soma + Termo(x, i);
   }

   return soma;
}

/****
 * main(): Testa a função Seno()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   printf( "\n\t>>> Este programa calcula senos usando"
           "\n\t>>> serie de Taylor e a funcao sin().\n" );
```

```
   printf( "\n\t>>> Seno de %3.2f com 3 termos: %3.2f", PI/4, Seno(PI/4, 3) );
   printf( "\n\t>>> Seno de %3.2f com 5 termos: %3.2f", PI/4, Seno(PI/4, 5) );
   printf( "\n\t>>> Seno de %3.2f usando sin(): %3.2f\n", PI/4, sin(PI/4) );
   return 0;
}
```

Resultado de execução do programa:

```
        >>> Este programa calcula senos usando
        >>> serie de Taylor e a funcao sin()

        >>> Seno de 0.79 com 3 termos: 0.71
        >>> Seno de 0.79 com 5 termos: 0.71
        >>> Seno de 0.79 usando sin(): 0.71
```

5.11.6 Comparando Números Reais

Preâmbulo: Raramente, operadores relacionais são adequados para comparar números reais em virtude da forma aproximada com que esses números são representados em computadores. Por exemplo, o resultado de uma operação real que, matematicamente, deve ser exatamente `0.1` pode ser representado como `0.10000000149...` num computador. O problema decorrente da comparação de números reais por meio de operadores relacionais pode ser ilustrado no seguinte programa:

```
#include <stdio.h>

int main(void)
{
   double d;

      /* Em Matemática, o resultado da seguinte operação é exatamente igual a 0.1 */
   d = 10.0 - 9.9;

   /*                                              */
   /* Compara o valor de 'd' com 0.1 usando '==' */
   /*                                              */

   printf("\n>>> Usando o operador '==' <<<\n");

   if (d == 0.1) {
      printf("\n\t>>> Os valores sao iguais\n");
   } else {
      printf("\n\t>>> Os valores sao diferentes\n");
   }

   return 0;
}
```

Quando executado, esse programa apresenta o seguinte resultado:

```
>>> Usando o operador '==' <<<

        >>> Os valores sao diferentes
```

Em bom português, o resultado do programa acima informa que `0.1` não é igual a `0.1`!

Problema: (a) Escreva uma função que recebe dois números reais como parâmetros e retorna `0`, se eles forem considerados iguais, um valor positivo se o primeiro parâmetro for maior do que o segundo parâmetro e um valor negativo se o primeiro parâmetro for menor do que o segundo parâmetro. (b) Escreva um programa que teste a função especificada no item (a).

Observação: Os algoritmos necessários para implementação do programa proposto nesse exemplo são deixados como exercício para o leitor.

Solução de (a): Uma solução para o problema descrito no preâmbulo consiste em usar uma constante com um valor bem pequeno tal que, quando a diferença absoluta entre os números reais sendo comparados for menor do que ou igual a essa constante, eles serão considerados iguais. Caso contrário, eles serão considerados diferentes. O valor dessa constante depende da precisão que se deseja obter para um programa Aqui, será utilizado o valor `1.0E-14`, que é bastante razoável para as pretensões deste livro (v. **Seção 7.5**).

A função `ComparaDoubles()` a seguir implementa o que foi exposto.

```
/****
 * ComparaDoubles(): Compara dois valores do tipo double
 *
 * Parâmetros: d1, d2 (entrada): valores que serão comparados
 *
 * Retorno: 0, se os números forem considerados iguais
 *          -1, se d1 for considerado menor do que d2
 *          1, se d1 for considerado maior do que d2
 ****/
int ComparaDoubles(double d1, double d2)
{
      /* Verifica se o valor absoluto da diferença entre os números é menor  */
      /* do que ou igual à precisão determinada pela constante DELTA. Se for */
      /* o caso, eles são considerados iguais. Caso contrário, verifica qual */
      /* deles é menor ou maior.                                             */
   if (fabs(d1 - d2) <= DELTA) {
      return 0; /* d1 e d2 são considerados iguais */
   } else if (d1 < d2) {
      return -1; /* d1 é menor do que d2 */
   } else {
      return 1; /* d1 é maior do que d2 */
   }
}
```

Análise:

- Apesar de essa abordagem ser comumente utilizada por muitos programadores de C e de outras linguagens, ela não constitui a melhor opção para comparação de números reais. Isto é, essa abordagem leva em consideração o **erro de precisão absoluto**, quando o mais correto seria adotar uma estimativa de erro que levasse em consideração a ordem de grandeza dos números sendo comparados (i.e., o **erro de precisão relativo**). Mas, uma discussão mais aprofundada sobre esse tema está além do escopo de um livro de introdução à programação. Para obter maiores detalhes, consulte: Knuth, D., *The Art of Computer Programming Volume 2* (v. **Bibliografia**).
- A função **fabs()** é usada para calcular o valor absoluto da diferença entre os dois valores ora comparados. Para usar essa função, o cabeçalho `<math.h>` deve ser incluído.

Solução de (b):

```
/****
 * main(): Testa a função ComparaDoubles()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
```

```
int main(void)
{
   double d = 10.0 - 9.9;

   /*                                                        */
   /* Compara o valor de 'd' com 0.1 usando ComparaDoubles() */
   /*                                                        */

   printf("\n\n>>> Usando ComparaDoubles() <<<\n");

   if (ComparaDoubles(d, 0.1) == 0) {
      printf("\n\t>>> Os valores sao iguais\n");
   } else {
      printf("\n\t>>> Os valores sao diferentes\n");
   }

   return 0;
}
```

Para completar o programa é necessária a inclusão das seguintes diretivas no início do arquivo-fonte que contém as funções apresentadas acima:

```
#include <stdio.h> /* Função printf() */
#include <math.h>  /* Função fabs()   */

   /* Precisão utilizada na comparação de valores do tipo double */
#define DELTA 1.0E-14
```

Resultado de execução do programa:

```
>>> Usando ComparaDoubles() <<<

        >>> Os valores sao iguais
```

Análise: Compare a saída do programa acima com aquela apresentada pelo programa visto no preâmbulo e note que, agora, o resultado é aceitável.

5.11.7 Calculando Raiz Quadrada por meio de Busca Binária

Preâmbulo: Uma técnica utilizada para calcular a raiz quadrada de um número real não negativo `x`, denominada **busca binária**, segue o seguinte algoritmo:

1. Estime um limite inferior e outro superior:
   - 1.1 Se o número `x` for maior do que `1`, use `1` como limite inferior e o próprio número `x` como limite superior.
   - 1.2 Se o número `x` 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 certo valor de precisão (p. ex., `1.0E-10`), então o resultado será a média aritmética desses limites.
3. Caso contrário, verifique se a média aritmética dos limites inferior e superior elevada ao quadrado é maior ou menor do que `x`. Seja `M` o valor dessa média. Então:
   - 3.1 Se M^2 for menor do que `x`, repita o processo a partir do passo `2` considerando agora `M` como limite inferior; o limite superior permanece o mesmo.
   - 3.2 Se M^2 for maior do que `x`, repita o processo a partir do passo `2` considerando agora `M` como limite superior; o limite inferior permanece o mesmo.

Problema: (a) Implemente o algoritmo acima como uma função que retorna a raiz quadrada de um número real positivo `x`, recebendo como parâmetros o número `x` e os limites inferior e superior entre

os quais se encontra a referida raiz. (b) Escreva um programa que solicita um valor numérico do usuário, calcula a raiz quadrada desse valor utilizando **sqrt()** e a função descrita em (a) e apresenta o resultado na tela.

Solução: Os itens (a) e (b) do enunciado são apresentados em conjunto no programa abaixo. Os algoritmos necessários para implementação do programa proposto nesse exemplo são deixados como exercício para o leitor.

```
#include <stdio.h>   /* printf() */
#include <math.h>    /* sqrt()   */
#include "leitura.h" /* LeReal() */

#define TOLERANCIA 1.0E-14 /* Tolerância de erro admitida no cálculo da raiz */

/****
 *
 * RaizQuadrada(): Calcula a raiz quadrada de um número
 *                 real não negativo usando busca binária
 * Parâmetros:
 *      x (entrada): número cuja raiz será calculada
 *      inferior (entrada): limite inferior do intervalo
 *      superior (entrada): limite superior do intervalo
 * Retorno: A raiz quadrada positiva do primeiro parâmetro, se inferior <= superior
 *          e o número não for negativo. Caso contrário, um valor negativo que
 *          indica ocorrência de erro.
 *
 ****/
double RaizQuadrada(double x, double inferior, double superior)
{
   double meio; /* Valor médio do intervalo */

      /* Se o limite inferior é maior do que o limite superior ou o valor    */
      /* que se deseja calcular a raiz é negativo, esta função foi chamada   */
      /* incorretamente. Este é o significado do valor retornado neste caso. */
   if (inferior > superior || x < 0) {
      return -1.0;
   }

      /* O laço encerra quando a largura do intervalo */
      /* tornar-se menor do que a tolerância de erro  */
   while (1) {
         /* Calcula o meio do intervalo */
      meio =(inferior + superior)/2.0;

         /* Verifica se a raiz quadrada foi encontrada  */
         /* de acordo com a tolerância de erro admitida */
      if (superior - inferior <= TOLERANCIA) {
         break; /* A diferença entre 'superior' e 'inferior' é */
                /* tão pequena que se pode considerar o meio   */
                /* deste intervalo como sendo a raiz procurada */
      }

      /*******************************/
      /* >>> Redução do intervalo <<< */
      /*******************************/

         /* Se o quadrado do meio for maior do que o número, a raiz está */
         /* na metade inferior do intervalo. Caso contrário, a raiz está */
         /* na metade superior do intervalo.                             */
```

```
      if (meio*meio > x) {
         superior = meio; /* A raiz está na metade inferior */
      } else {
         inferior = meio; /* A raiz está na metade superior */
      }
   }

      /* A variável 'meio' armazena a raiz procurada, */
      /* de acordo com a tolerância admitida          */
   return meio;
}

/****
 *
 * main(): Testa RaizQuadrada() comparando o resultado apresentado
 *         por esta função com aquele apresentado por sqrt()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   double num;
      /* Apresenta o programa */
   printf( "\n\t>>> Este programa calcula a raiz quadrada"
           "\n\t>>> de um numero nao negativo usando busca binaria.\n" );

      /* Lê o número cuja raiz quadrada será calculada */
   printf("\n\t>>> Digite um numero nao negativo> ");
   num = LeReal();

      /* Se a raiz quadrada existir, calcula-a e exibe-a */
   if(num < 0) { /* Usuário desobediente */
      printf("\n\t>>> Nao ha raiz quadrada de numero negativo\n");
   } else { /* Usuário bacana */
      printf( "\n\t>>> Raiz quadrada aproximada de %3.2f: ", num );

         /* Determina o intervalo no qual se encontra a raiz quadrada, chama a */
         /* função RaizQuadrada() de acordo com o intervalo e exibe resultado  */
      if(num < 1) { /* A raiz está entre 'num' e 1.0 */
         printf( "%3.2f\n", RaizQuadrada(num, num, 1.0) );
      } else { /* A raiz está entre 1.0 e 'num' */
         printf( "%3.2f\n", RaizQuadrada(num, 1.0, num) );
      }

         /* Apresenta o resultado calculado usando sqrt()*/
      printf("\t>>> Raiz quadrada de %3.2f usando sqrt(): %3.2f\n", num, sqrt(num));
   }
   return 0;
}
```

Análise: Espera-se que os comentários apresentados no programa sejam esclarecedores e não deixem dúvidas para o leitor.

Exemplo de execução do programa:

```
>>> Este programa calcula a raiz quadrada
>>> de um numero nao negativo usando busca binaria.

>>> Digite um numero nao negativo> 2

>>> Raiz quadrada aproximada de 2.00: 1.41
>>> Raiz quadrada de 2.00 usando sqrt(): 1.41
```

5.11.8 Gerando Números Aleatórios

Problema: (a) Escreva uma função que retorna um número aleatório entre os limites inteiros recebidos como parâmetros. (b) Escreva uma função **main()** que teste a função solicitada no item (s).

Solução de (a): O algoritmo da Figura 5–22 retorna um número aleatório entre os valores recebidos. Depois dessa figura aparece a função que implementa esse algoritmo.

ALGORITMO NUMEROALEATÓRIOENTREDOISLIMITES

ENTRADA: m, n /* Limites do intervalo */

```
booleano primeiraChamada

primeiraChamada ← verdadeiro

se (primeiraChamada) então
  srand
  primeiraChamada ← falso

se (n < m) então
  retorne rand%(m - n + 1) + n
senão
  retorne rand%(n - m + 1) + m
```

FIGURA 5–22: ALGORITMO DE OBTENÇÃO DE NÚMERO ALEATÓRIO ENTRE DOIS LIMITES

```
/****
 * NumeroAleatorio(): Retorna um número aleatório entre os valores recebidos
 *
 * Parâmetros:
 *      m, n (entrada) - valores que definem o intervalo de números sorteados
 *
 * Retorno: O número aleatório gerado
 *
 * Observações:
 *      1. É indiferente se m <= n ou n <= m
 *      2. Quando chamada pela primeira vez, esta função
 *         alimenta o gerador de números aleatórios
 ****/
int NumeroAleatorio(int m, int n)
{
   static int primeiraChamada = 1; /* Esta variável checa se a função está   */
                                   /* sendo chamada pela primeira vez e deve */
                                   /* ter duração fixa                       */

      /* Se esta for a primeira chamada da função, */
      /* alimenta o gerador de números aleatórios  */
   if (primeiraChamada) {
         /* Alimenta o gerador de números aleatórios */
      srand(time(NULL));
```

```
         /* A próxima chamada não será mais a primeira */
      primeiraChamada = 0;
   }

      /* Leva em consideração o fato de n poder ser menor do que m */
   if (n < m) {
      return rand()%(m - n + 1) + n;
   }

      /* m é menor do que ou igual a n */
   return rand()%(n - m + 1) + m;
}
```

Análise: A função `NumeroAleatorio()` alimenta o gerador de números aleatórios quando é chamada pela primeira vez e ela obtém essa informação por meio da variável local de duração fixa `primeiraChamada`.

Solução de (b): O algoritmo aparece abaixo e a função **main()** vem logo em seguida.

ALGORITMO PRINCIPAL

```
inteiro i1, i2; /* Valores que definem o intervalo */

escreva("Primeiro limite do intervalo: ");
leia(i1)

escreva("Segundo limite do intervalo: ");
leia(i2)

escreva("Número aleatório gerado: ", NumeroAleatorio(i1, i2));
```

```
/****
 *
 * main(): Lê dois valores que representam limites de um intervalo, gera um número
 *         aleatório entre os limites introduzidos e apresenta o resultado
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   int i1, i2; /* Valores que definem o intervalo */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa gera um numero aleatorio"
           "\n\t>>> entre os valores introduzidos.\n" );

   printf("\n\t>>> Primeiro limite do intervalo: ");
   i1 = LeInteiro();

   printf("\n\t>>> Segundo limite do intervalo: ");
   i2 = LeInteiro();

   printf( "\n\t>>> Numero aleatorio gerado: %d\n", NumeroAleatorio(i1, i2) );

   return 0;
}
```

Para completar o programa, é necessário incluir os seguintes cabeçalhos:

```
#include <stdio.h>   /* printf()         */
#include <stdlib.h>  /* rand() e srand() */
```

```
#include <time.h>    /* time()           */
#include "leitura.h" /* LeInteiro()      */
```

Exemplo de execução do programa:

```
>>> Este programa gera um numero aleatorio
>>> entre os valores introduzidos.

>>> Primeiro limite do intervalo: 33

>>> Segundo limite do intervalo: 55

>>> Numero aleatorio gerado: 48
```

5.11.9 Calculadora Dirigida por Menu

Problema: Escreva um programa que apresenta um menu no qual cada opção representa encerramento do programa ou uma operação aritmética básica, lê a opção do usuário e efetua a respectiva operação.

Solução:

Observação: Os algoritmos seguidos pelo programa apresentado adiante são deixados como exercício para o leitor.

```
#include <stdio.h>   /* Entrada e saída */
#include "leitura.h" /* LeituraFacil    */

/****
 * ApresentaMenu(): Apresenta um menu de opções
 *
 * Parâmetros: nenhum
 *
 * Retorno: Nada
 ****/
void ApresentaMenu(void)
{
   printf( "\n\n\t****** Opcoes ******\n"
           "\n\t[A]dicao"
           "\n\t[M]ultiplicacao"
           "\n\t[S]ubtracao"
           "\n\t[D]ivisao"
           "\n\t[E]ncerra o programa\n" );
}

/****
 *
 * main(): Apresenta um menu no qual cada opção representa
 *         encerramento do programa ou uma operação aritmética
 *         básica, lê a opção e efetua a respectiva operação
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   int    opcao;
   double x, y;

      /* Apresenta o programa */
```

```
   printf( "\n\t>>> Este programa e' uma calculadora mambembe."
           "\n\t>>> Escolha a opcao 'E' para encerra-lo." );

      /* O laço encerra quando o usuário escolhe a opção 'E' */
   while (1) {
      ApresentaMenu();

         /* Lê a opção do usuário */
      printf("\n\t>>> Escolha sua opcao: ");
      opcao = LeOpcao("AMSDEamsde");

         /* Se a opção for 'E', encerra o laço */
      if (opcao == 'E' || opcao == 'e') {
         break; /* Encerra o laço while */
      }

      /*                                                                  */
      /* Lê os valores sobre os quais a operação aritmética será efetuada */
      /*                                                                  */

      printf("\n\t>>> Digite o primeiro valor: ");
      x = LeReal();

      printf("\n\t>>> Digite o segundo valor: ");
      y = LeReal();

         /* Efetua a operação escolhida */
      switch (opcao) {
         case 'A':
         case 'a': /* Adição */
            printf("\n\t> %f + %f = %f", x, y, x + y);
            break;
         case 'M':
         case 'm': /* Multiplicação */
            printf("\n\t> %f * %f = %f", x, y, x * y);
            break;
         case 'S':
         case 's': /* Subtração */
            printf("\n\t> %f - %f = %f", x, y, x - y);
            break;
         case 'D':
         case 'd': /* Divisão */
            if (y) {
               printf("\n\t> %f / %f = %f", x, y, x / y);
            } else {
               printf("\n\t> O divisor nao pode ser zero");
            }
            break;
         default: /* Não se deve chegar até aqui */
            printf("\nEste programa contem um erro!!!");
            break;
      } /* switch */
   } /* while */

      /* Despede-se do usuário */
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Exemplo de execução do programa:

```
>>> Este programa e' uma calculadora mambembe.
>>> Escolha a opcao 'E' para encerra-lo.

****** Opcoes ******

[A]dicao
[M]ultiplicacao
[S]ubtracao
[D]ivisao
[E]ncerra o programa

>>> Escolha sua opcao: a

>>> Digite o primeiro valor: 2.5
>>> Digite o segundo valor: 3.5

> 2.500000 + 3.500000 = 6.000000

****** Opcoes ******

[A]dicao
[M]ultiplicacao
[S]ubtracao
[D]ivisao
[E]ncerra o programa

>>> Escolha sua opcao: d

>>> Digite o primeiro valor: 5
>>> Digite o segundo valor: 0

> O divisor nao pode ser zero

****** Opcoes ******

[A]dicao
[M]ultiplicacao
[S]ubtracao
[D]ivisao
[E]ncerra o programa

>>> Escolha sua opcao: d

>>> Digite o primeiro valor: 5
>>> Digite o segundo valor: 2.5

> 5.000000 / 2.500000 = 2.000000

****** Opcoes ******

[A]dicao
[M]ultiplicacao
[S]ubtracao
[D]ivisao
[E]ncerra o programa

>>> Escolha sua opcao: e

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

5.11.10 Conjectura de Collatz 1

Preâmbulo: Seja `n` um número inteiro positivo. Então, considere uma sequência de números inteiros tendo `n` como primeiro termo e tal que cada termo seguinte é obtido dividindo-se o termo corrente por `2`, se ele for par, ou multiplicando-se por `3` e somando-se `1`, se ele for ímpar. A **Conjectura**

de Collatz propõe que, eventualmente, um termo dessa sequência assume **1** como valor. Essa proposição nunca foi provada como verdadeira ou falsa (por isso ela é considerada uma *conjectura* e não um *teorema*).

Problema: Escreva um programa que determina a sequência de Collatz que começa com um número inteiro positivo recebido como entrada. O programa deve exibir a referida sequência de Collatz e contar o número de termos da mesma.

Solução: Os algoritmos seguidos pelo programa apresentado a seguir são deixados como exercício para o leitor enquanto o programa é apresentado a seguir.

```
#include <stdio.h>   /* printf() e putchar() */
#include "leitura.h" /* LeInteiro()          */

/****
 * Collatz(): Apresenta na tela a sequência de Collatz a partir de um dado número
 *            inteiro positivo e conta o número de termos dessa sequência
 *
 * Parâmetros: n (entrada) - termo inicial da sequência
 *
 * Retorno: O número de termos da sequência
 ****/
int Collatz(int n)
{
   int cont = 1; /* Conta o número de termos da sequência e é iniciado com 1 pois */
                 /* a sequência tem pelo menos um termo, que é o parâmetro       */

      /* O primeiro termo deve ser positivo */
   if (n <= 0) {
      return 0; /* Nada será escrito, pois não existe sequência */
   }

      /* Apresenta o primeiro termo na tela */
   printf("\n>>> %d%s", n, n > 1 ? " -> " : "\n");

      /* Determina e apresenta na tela os demais termos. Se o laço a seguir */
      /* não encerrar, i.e., se não for encontrado um termo igual a 1, você */
      /* terá encontrado um número que não satisfaz a conjectura de Collatz */
   while (1) {
         /* Gera o próximo termo */
      if (!(n%2)) { /* O termo corrente é par */
         n = n/2; /* Calcula o próximo termo */
      } else { /* O termo corrente é ímpar */
         n = 3*n + 1; /* Calcula o próximo termo */
      }

      printf("%d", n); /* Apresenta o novo termo */

      ++cont; /* Mais um termo foi gerado */

         /* Se o novo termo for 1, a sequência está encerrada */
      if (n != 1) { /* Existe, pelo menos, mais um termo */
            /* Escreve seta que antecede o próximo termo */
         printf(" -> ");
      } else { /* Não existem mais termos */
         putchar('\n'); /* Embelezamento */
         break; /* Encerra o laço */
      }
   }
   return cont;
}
```

```
/****
 * main(): Encontra a sequência de Collatz a partir de um dado
 *         número inteiro positivo
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int n, /* Número introduzido pelo usuário */
       nTermos;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa encontra a sequencia de Collatz"
           "\n\t>>> a partir de um dado valor inteiro positivo."
           "\n\t>>> Digite zero para encerrar o programa.\n" );

      /* O laço encerra quando o usuário digitar 0 */
   while (1) {
         /* Lê o termo inicial da sequência */
      printf( "\n\t>>> Digite um numero inteiro positivo"
              "\n\t>>> ou zero para encerrar o programa: ");
      n = LeInteiro();

      if (!n) { /* O usuário digitou zero */
         break; /* Encerra o laço */
      } else if (n < 0) { /* Valor digitado é negativo */
         printf("\a\n\t>>> Nao sao aceitos numeros negativos");
      } else {
            /* Apresenta a sequência e obtém o número de termos */
         nTermos = Collatz(n);

            /* Apresenta o número de termos da sequência */
         printf( ">>> A sequencia possui %d termos\n", nTermos );
      }
   }

      /* Despede-se do usuário */
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Exemplo de execução do programa:

```
        >>> Este programa encontra a sequencia de Collatz
        >>> a partir de um dado valor inteiro positivo.
        >>> Digite zero para encerrar o programa.

        >>> Digite um numero inteiro positivo
        >>> ou zero para encerrar o programa: 20

>>> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
>>> A sequencia possui 8 termos

        >>> Digite um numero inteiro positivo
        >>> ou zero para encerrar o programa: 0

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

5.12 Exercícios de Revisão

Introdução (Seção 5.1)

1. (a) O que é um subprograma? (b) O que é uma função?
2. Qualquer programa de console escrito em C deve usar funções?
3. Que relação existe entre a abordagem dividir e conquistar discutida no **Capítulo 2** e o uso de funções?

Endereços e Ponteiros (Seção 5.2)

4. Como se obtém o endereço de uma variável?
5. Uma variável de um tipo primitivo sempre contém um valor válido do tipo com o qual ela é definida, mesmo quando ela não é iniciada. No entanto, uma variável que representa um ponteiro nem sempre contém um valor válido. Por quê?
6. (a) Para que serve o operador de indireção? (b) Como deve ser um operando desse operador? (c) Qual é o resultado da aplicação desse operador? (d) Por que o operador de indireção recebe essa denominação?
7. (a) O operador de endereço pode ser aplicado a qualquer tipo de variável? (b) E o operador de indireção?
8. O que é e para que serve um ponteiro nulo?
9. O que há de errado nas seguintes definições de variáveis?

```
int  x1 = -1, x2 = 5;
int *p1 = &x1, p2 = &x2;
```

10. (a) O seguinte trecho de programa é legal? (b) Em caso afirmativo, qual é o seu efeito?

```
int x = 5;
int *p = &x;

*p = 2*x;
```

11. Suponha que `x` seja uma variável. (a) Qual é o significado da expressão `&*&x`? (b) Qual é o significado da expressão `*&*&x`?
12. Por que o programa a seguir é abortado?

```
"Este e' um string muito grande para "
"ser contido numa linha do meu programa"
```

13. Qual é a saída do seguinte programa? [**Dica**: Este programa é sintaticamente correto, mas apresenta um erro de lógica que não é de fácil visualização. Um bom compilador apresentará uma mensagem de advertência sugerindo qual é o erro.]

```
#include <stdio.h>

int main(void)
{
   int  y = 100, x = 10;
   int *p = &x;

   y = y/*p; /* Divide y por *p */;

   printf("y = %d", y);

   return 0;
}
```

14. Por que a iniciação a seguir:

```
int  x;
int *p = &x;
```

é legal, mas a instrução:

```
*p = &x;
```

é problemática?

15. Por que, do ponto de vista sintático, a iniciação:

```
int *p = 5;
```

é ilegal, mas a instrução:

```
*p = 5;
```

é perfeitamente legal, apesar de também ser problemática?

Funções (Seção 5.3)

16. Em que situações o uso de funções, além de **main()**, é recomendável num programa?

17. Como o uso de funções facilita a escrita de programas?

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

Definições de Funções (Seção 5.4)

19. Descreva as partes que constituem uma definição de função?

20. (a) Por que se diz que parâmetros constituem o meio normal de comunicação de dados de uma função? (b) Que outros meios de comunicação de dados uma função possui?

21. Na linguagem C, uma função pode ser definida no corpo de outra função?

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

(a) `void F(int  x)`

(b) `int G(void)`

23. O que há de errado com o seguinte cabeçalho de função?

```
void F(int x, y)
```

24. Qual é o propósito de uma instrução **return**?

25. (a) Uma função cujo tipo de retorno é **void** pode ter em seu corpo uma instrução **return**? (b) Nesse caso, essa instrução pode ser usada acompanhada de um valor?

26. (a) Uma função cujo tipo de retorno não é **void** tem obrigação de ter instrução **return**? (b) Nesse caso, essa instrução pode não vir acompanhada de um valor?

27. O tipo da expressão que acompanha uma instrução **return** precisa coincidir com o tipo de retorno da função no corpo da qual a referida instrução se encontra? Explique.

28. Uma função pode ter mais de uma instrução **return**? Explique.

29. Escreva o corpo da função a seguir usando uma única linha de instrução por meio do operador condicional (`? :`).

```
int MinhaFuncao(int x)
{
    if (x < 0)
        return  0;
    else
        return  1;
}
```

30. A função `TrocaErrada()`, apresentada abaixo, propõe-se a trocar os valores de duas variáveis inteiras. Entretanto, esta função contém um grave erro. Qual é o erro?

```
void TrocaErrada(int * x, int * y)
{
   int *aux;

   *aux = *x;
   *x   = *y;
   *y   = *aux;
}
```

31. Interprete o valor retornado pela função `F()` a seguir:

```
int Abs(int x)
{
   return x >= 0 ? x : -x;
}
```

32. O que há de errado com a seguinte definição de função?

```
int F(int x)
{
   if (x > 0) {
      return x;
   }
}
```

33. De acordo com Matemática, o valor absoluto de um número pode ser definido como a raiz quadrada positiva do número elevado ao quadrado. Seguindo essa definição, uma função que calcula o valor absoluto de uma variável do tipo **int** poderia ser definida como:

```
int Abs(int x)
{
   return sqrt(x*x);
}
```

Apresente um argumento que demonstre que essa definição matemática de valor absoluto é inadequada em programação. [**Dica**: O que aconteceria se essa função fosse chamada recebendo como parâmetro um valor próximo ao maior valor permitido para o tipo **int**?]

Chamadas de Funções (Seção 5.5)

34. Em que situações um parâmetro formal deve ser declarado como ponteiro?

35. Em que situações é aconselhável (mas não obrigatório) utilizar um ponteiro como parâmetro de uma função?

36. O que é modo de um parâmetro?

37. (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?

38. Um parâmetro formal que não é definido como ponteiro pode ser um parâmetro de saída?

39. (a) O que é parâmetro formal? (b) O que é parâmetro real? (c) Que relação existe entre parâmetros reais e formais de uma função?

40. (a) Parâmetro real precisa ter nome? (b) Se um parâmetro real tiver nome, ele deve coincidir com o nome do parâmetro formal correspondente?

41. Qual é a regra de casamento que deve ser obedecida entre um parâmetro real e um parâmetro formal correspondente durante uma chamada de função?

42. O que ocorre quando, numa chamada de função, parâmetros reais e formais correspondentes não são do mesmo tipo?

43. (a) O que é passagem de parâmetros? (b) Que regras devem ser satisfeitas durante a passagem de parâmetros numa chamada de função?

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

45. Uma chamada de função cujo tipo de retorno é **void** pode fazer parte de uma expressão?

46. Uma função cujo tipo de retorno não é **void** pode ser chamada numa linha isolada de instrução?

47. Qual é o significado de **(void)** precedendo uma chamada de função, como no exemplo abaixo? [**Sugestão**: v. **Seção 3.10.2**.]

```
(void) F(5);
```

48. Suponha que o cabeçalho `<stdio.h>` seja incluído em cada um dos seguintes programas. Qual é a saída exibida por cada um deles.

(a)

```
int Funcao1(int  x)
{
   int  y = 0;

   y = y + x;

   return  y;
}

int main(void)
{
   int a, contador;

   for (contador = 1; contador <= 5; ++contador) {
      a = Funcao1(contador);
      printf("%d  ", a);
   }
   return 0;
}
```

(b)

```
int Funcao2(int  x)
{
   static int  y = 0;

   y = y + x;

   return  y;
}

int main(void)
{
   int a, contador;

   for (contador = 1; contador <= 5; ++contador) {
      a = Funcao2(contador);
      printf("%d  ", a);
   }
   return 0;
}
```

(c)

```
int Funcao3(int  a)
{
   int  b = 1;

   b = b + 1;

   return  b + a;
}
int Funcao4(int x)
{
   int  b;

   b = Funcao3(x);

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

   for (contador = 1; contador <= 5; ++contador)  {
      b = b + Funcao3(a) + Funcao4(a);
      printf("%d  ", b);
   }

   return 0;
}
```

Alusões e Protótipos de Funções (Seção 5.6)

49. Suponha que uma função `F1()` chama uma função `F2()`. É importante para o compilador a ordem na qual essas funções são definidas?

50. (a) O que é uma alusão a uma função? (b) Quando uma alusão é requerida num programa em C?

51. Explique por que, mesmo quando não são estritamente necessárias, alusões facilitam o trabalho do programador. [**Dica**: Pense num programa contendo muitas funções.]

52. (a) O que é protótipo de uma função? (b) Que relação existe entre protótipo e alusão de uma função? (c) Qual é a vantagem advinda do uso de protótipos de funções em relação ao uso do estilo antigo de alusões que não usa protótipos?

53. (a) Como é possível escrever uma alusão de função sem o uso do protótipo da função? (b) Por que essa prática não é recomendada?

54. Quando uma alusão é escrita usando o protótipo de uma função, os nomes dos parâmetros no protótipo devem coincidir com os nomes dos parâmetros formais na definição da função?

55. Escreva o protótipo de uma função denominada *F* para cada uma das seguintes situações:

(a) `F()` possui um parâmetro de entrada do tipo **int** denominado `dia`, um parâmetro de saída do tipo **int** denominado `mes` e não retorna nenhum valor.

(b) `F()` possui um parâmetro de saída do tipo **int** denominado `x`, um parâmetro de entrada e saída do tipo **int** denominado `y` e retorna o endereço de uma variável do tipo **int**.

56. Escreva protótipos para funções que possuam parâmetros e valores de retorno dos seguintes tipos:

(a) Retorno: nenhum; parâmetros: **double** e ponteiro para **char**

(b) Retorno: ponteiro para **int**; parâmetro: nenhum

57. Justifique a seguinte afirmação: *o uso de* ***void*** *entre parênteses numa definição de função é opcional, mas isso não ocorre no caso de uma alusão de função.*

58. Com relação ao uso de nomes de parâmetros numa alusão responda as seguintes questões:
(a) É obrigatório o uso de nomes de parâmetros?
(b) Os nomes dos parâmetros numa alusão devem corresponder aos nomes dos respectivos parâmetros na definição da função?
(c) Se nomes de parâmetros não forem necessários numa alusão, por que eles são tipicamente utilizados?

59. (a) O que há de errado com o seguinte programa? (b) Como esse problema pode ser corrigido?

```
#include <stdio.h>

int main(void)
{
   ApresentaMenu();

   return 0;
}

void ApresentaMenu(void)
{
   printf( "\nOpcoes:"
           "\n\tOpcao 1"
           "\n\tOpcao 2"
           "\n\tOpcao 3\n" );
}
```

60. Observando o protótipo de função a seguir, que informações você pode inferir em relação a função `G()`?

```
int G(double *, int)
```

61. (a) Para que serve a palavra-chave **extern**. (b) Essa palavra-chave é estritamente necessária em C?

62. Quando um programa contém muitas funções, por que é útil incluir alusões a todas elas precedendo suas definições?

Subprogramas em Linguagem Algorítmica (Seção 5.7)

63. Quais são as diferenças e semelhanças entre programas e subprogramas?

64. Quais são as etapas que devem ser seguida para a criação de um subprograma em pseudolinguagem?

65. Compare as etapas envolvidas na criação de subalgoritmos em linguagem algorítmica com aquelas utilizadas na construção de algoritmos apresentadas no **Capítulo 2**.

Interação Dirigida por Menus (Seção 5.8)

66. (a) O que é interação dirigida por menus? (b) Cite um exemplo cotidiano do uso desse tipo de interação?

67. Por que, tipicamente, um programa que implementa interação dirigida por menus usa uma instrução **switch-case**?

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

69. Para que serve a função `LeOpcao()` da biblioteca LeituraFacil e como ela deve ser usada?

70. Que facilidade adicional a função `LeOpcaoSimNao()`, discutida na **Seção 5.8.3**, apresenta em relação à função `LeOpcao()`?

Duração de Variáveis (Seção 5.9)

71. O que é uma variável de duração automática? (b) Como uma variável de duração automática é definida? (c) Qual é o escopo de uma variável de duração automática? (d) O que acontece quando uma variável de duração automática não é explicitamente iniciada?

72. (a) Qual é o significado da palavra-chave **auto**? (b) Essa palavra-chave é necessária em C? (c) Caso a resposta ao item (b) seja negativa, faria sentido remover **auto** do rol de palavras-chaves de C?

73. (a) O que significa liberar uma variável ou parâmetro? (b) Quando uma variável de duração automática é liberada? (c) Quando uma variável de duração fixa é liberada?

74. (a) O que acontece quando uma variável de duração automática não é explicitamente iniciada? (b) O que acontece quando uma variável de duração fixa não é explicitamente iniciada?

75. Que restrições são aplicadas a iniciações de variáveis de duração fixa?

76. (a) Por que uma variável de duração fixa não pode ser iniciada com o valor de uma variável de duração automática? (b) Por que uma variável de duração fixa não pode ser iniciada com o valor de outra variável de duração fixa?

77. (a) Uma variável de duração automática retém seu valor entre duas chamadas da função na qual ela é definida? (b) Uma variável de duração fixa definida no corpo de uma função retém seu valor entre duas chamadas da mesma função?

78. O que o seguinte programa escreve na tela?

```
#include <stdio.h>

int main(void)
{
   int i;

   for (i = 0; i < 5; ++i) {
      int        x = 1;
      static int y = 1;

      printf("x = %d \t y = %d\n", x, y);

      ++x;
      ++y;
   }

   return 0;
}
```

79. A iniciação da variável `y` no trecho de programa a seguir é legal?

```
void F(void)
{
   double     x;
   static int y = sizeof(x)/2;
   ...
}
```

80. Por que o programa a seguir não consegue ser compilado?

```
#include <stdio.h>

int main(void)
{
   int        i = 1;
   int        j = 2*i + 5;
   static int k = i;

   printf("i = %d  j = %d  k = %d", i, j, k);

   return 0;
}
```

81. O que exibe na tela o programa abaixo?

```
#include <stdio.h>

int F(int a)
{
   static int b;

   b = b + 1;

   return b + a;
}

int G(int a)
{
   int  b;

   b = F(a);

   return b;
}

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

   for (i = 1; i <= 3; ++i) {
      b = b + G(a) + F(a);
      printf("%d\t", b);
   }
}
```

82. A seguinte função foi criada para calcular a soma compreendida entre `1` e o número inteiro positivo recebido como parâmetro. No entanto, essa função contém um erro que faz com que ela sempre retorne `1` quando seu parâmetro for positivo. (a) Qual é esse erro? (b) Como corrigi-lo?

```
int SomaAteN(int n)
{
   if (n <= 0) {
      return 0;
   }

   while (1) {
      int soma = 0;

      soma = soma + n--;

      if (!n) {
         return soma;
      }
   }
}
```

83. A solução de um estudante de programação para o problema apresentado pela função `SomaAteN()` da questão anterior foi qualificar a definição da variável `soma` com **static**. Ou seja, a função apresentada como solução para a questão anterior foi:

```
int SomaAteN2(int n)
{
   if (n <= 0) {
      return 0;
   }

   while (1) {
      static int soma = 0;

      soma = soma + n--;

      if (!n) {
         return soma;
      }
   }
}
```

Explique por que a solução proposta pelo estudante não funciona se a função `SomaAteN2()` for chamada mais de uma vez num mesmo programa.

Escopo (Seção 5.10)

84. (a) Qual é a diferença entre escopo de bloco e escopo de função? (b) Que categorias de identificadores podem ter escopo de bloco? (c) Que categorias de identificadores podem ter escopo de função?

85. (a) Uma variável com escopo de arquivo pode ser ocultada por uma variável com escopo de bloco? (b) Um identificador com escopo de função pode ser ocultado?

86. Por que não é correto afirmar que parâmetros formais têm escopo de função?

87. (a) É possível haver duas variáveis com o mesmo nome num mesmo programa? (b) É possível haver duas variáveis com o mesmo nome numa mesma função? (c) É possível haver duas variáveis com o mesmo nome num mesmo bloco?

88. (a) O nome de um parâmetro formal pode coincidir com o nome de uma variável definida no corpo da função? (b) Se a resposta for afirmativa, como isso é possível?

89. Sabendo-se que uma variável tem duração automática, é possível inferir qual é seu tipo de escopo?

90. Sabendo-se que uma variável tem duração fixa, o que se pode concluir a respeito de seu escopo?

91. *Como parâmetros formais são definidos antes mesmo do início do corpo de uma função, ele vale em todo o corpo da função e, portanto, parâmetros formais têm escopo de função*. Explique por que essa argumentação é equivocada.

92. Considere o seguinte trecho de programa:

```
#include <stdio.h>

int         i;
static  int j;

void F( int  k )
{
   int        m;
   static int n;

   ...
}

static int G( int  l )
{
   ...
}
```

(a) Quais são os escopos das variáveis `i`, `j`, `m` e `n` e do parâmetro `k`?

(b) Quais são os escopos das funções `F()` e `G()`?

(c) Quais são as durações das variáveis `i`, `j`, `m` e `n`?

93. Por que o uso de variáveis globais num programa deve ser comedido?

94. O que o seguinte programa escreve na tela?

```
#include <stdio.h>

static int  x, y;

void  F(int *u, int *v)
{
   *u = 2*(*u);
   x = *u + *v;
   *u = *u - 1;
}

int main(void)
{
   x = 4;
   y = 2;

   F(&x, &y);

   printf("x = %d  y = %d\n", x, y);

   return 0;
}
```

95. O que o seguinte programa escreve na tela?

```
#include <stdio.h>

int main(void)
{
   int i = 0;

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

   { /* Bloco 1 */
      int i = 1;

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

      { /* Bloco 2 */
         int i = 2;

         printf("i = %d  ", i);

      } /* Bloco 2 */

   } /* Bloco 1 */

   return 0;
}
```

96. Qual das funções a seguir não pode ser compilada? Explique.

```
int F1(int x)
{
   double x = 2.5;

   return 0;
}

int F2(int x)
{
   {
      double x = 2.5;
   }
   return 0;
}
```

97. Por que a função `F()` abaixo não consegue ser compilada?

```
int F(int x)
{
   int y = 2, z = 1;

rotulo:
   z = x + y;

   if (z < 10) {
      goto rotulo;
   }

   {
   rotulo:
      z += x;
   }
   return x + y + z;
}
```

98. Por que o seguinte programa nunca termina?

```
#include <stdio.h>

int i;

void F()
{
   printf("\nFuncao F() chamada\n", i);

   for (i = 5; i > 0; --i)
      printf("\ti = %d\n", i);
}

int main(void)
{
   for (i = 0; i <= 5; ++i) {
      F();
   }

   return 0;
}
```

5.13 Exercícios de Programação

5.13.1 Fácil

EP5.1 Modifique a função `Incrementa()` apresentada a seguir de modo que as variáveis não sejam mais iniciadas, escreva um programa que chame várias vezes essa função modificada e verifique qual é a saída resultante. Compare os resultados com aqueles apresentados na **Seção 5.9.3**.

```
void  Incrementa( void )
{
    int          i = 1;
    static  int  j = 1;

    i++;
    j++;

    printf("Valor de i = %d\t\t Valor de j = %d", i, j);
}
```

EP5.2 Escreva um programa que recebe um número inteiro positivo como entrada. Então, se o número for primo, o programa deve apresentar na tela essa informação. Caso contrário, o programa apresentará na tela todos os divisores desse número. [**Sugestão**: Estude o exemplo apresentado na **Seção 5.11.2**.]

EP5.3 Escreva um programa que calcula a soma de todos números primos entre `1` e `100`. [**Sugestão**: Use a função `EhPrimo()` implementada no exemplo apresentado na **Seção 5.11.2**. O resultado apresentado pelo programa deve ser `1060`.]

EP5.4 (a) Escreva uma função que calcula raízes de equações do segundo grau. (b) Escreva um programa que lê repetidamente três valores reais (supostamente coeficientes de equações do segundo grau), calcula as raízes da equação se esses valores constituírem uma equação do segundo grau e apresenta o resultado. O programa deve encerrar quando o primeiro coeficiente introduzido pelo usuário for igual a zero. [**Sugestão**: Consulte o exemplo apresentado na **Seção 2.9.4** e use um laço de repetição que encerra quando o usuário digitar zero.]

EP5.5 A função cosseno pode ser expressa pela seguinte série (infinita) de Taylor:

$$cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

ou, equivalentemente:

$$sen(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Escreva um programa que calcula o cosseno de um arco em radianos usando uma aproximação da fórmula acima até o enésimo termo. [**Sugestão**: Utilize como referência o exemplo apresentado na **Seção 5.11.5**.]

EP5.6 **Preâmbulo**: Um número inteiro positivo 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`. **Problema**: (a) Escreva uma função denominada `EhPerfeito()`, que retorna `1` se o número inteiro positivo recebido como parâmetro é perfeito e `0` em caso contrário. (b) Escreva um programa que recebe um número inteiro maior do que ou igual a zero como entrada e informa 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 valor válido

Introduza um número inteiro positivo: 0
```

EP5.7 A técnica de busca binária, descrita no exemplo apresentado na **Seção 5.11.7**, pode ser estendida para calcular a raiz cúbica de um número real. (a) Escreva uma função denominada `RaizCubica()` com o seguinte protótipo:

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

que calcula a raiz cúbica de `x`, usando os parâmetros `inferior` e `superior` como limites iniciais do algoritmo de busca binária. Lembre-se que existe raiz cúbica de número negativo. [**Sugestões**: (1) Use como referência o exemplo apresentado na **Seção 5.11.7**. (2) Se o número for negativo, inverta seu sinal, calcule sua raiz cúbica e, após o cálculo, inverta o sinal do resultado.]

EP5.8 (a) Escreva uma função que calcula o número de maneiras que um subconjunto de `k` elementos pode ser escolhido de um conjunto de `n` elementos. (b) Escreva uma função **main()** que lê via teclado os valores de `k` e `n` e apresenta o resultado. [**Sugestão**: Esse é um problema elementar de análise combinatória que você deve ter estudado no ensino médio. A função solicitada no item (a) apenas calcula `n!/(k!*(n-k)!)`.]

EP5.9 **Preâmbulo**: Dois números inteiros positivos são **primos entre si** se o único divisor comum aos dois números for `1`. **Problema**: (a) Escreva uma função que recebe dois números inteiros positivos como parâmetros e retorna `1` se eles forem primos entre si ou `0`, em caso contrário. (b) Escreva uma função **main()** que lê dois valores inteiros positivos como entrada, verifica se eles são primos entre si e apresenta a conclusão na tela. [**Sugestões**: (1) Use a função `LeNatural()`, apresentada na **Seção 5.11.1**, para ler os dois números. (2) A função que verifica se os números são primos entre si deve retornar `1` quando um dos números for `1` ou quando o MDC deles for `1`. Nesse último caso, use a função `MDC()`, definida na **Seção 5.11.3**.]

EP5.10 **Preâmbulo**: Dois números inteiros positivos possuem **paridades distintas** quando um deles é par e o outro é ímpar. **Problema**: Escreva uma função que recebe dois números inteiros positivos como parâmetros e retorna `1` se eles tiverem paridades distintas ou zero, em caso contrário. (b) Escreva uma função **main()** que lê dois valores inteiros positivos como entrada, verifica se eles têm paridades distintas e apresenta a conclusão na tela. [**Sugestões**: (1) Use a função `LeNatural()`, apresentada na **Seção 5.11.1**, para ler os dois números. (2) Para facilitar a escrita da função solicitada no item (a), defina uma função que verifica se um número inteiro positivo é par.]

EP5.11 (a) Escreva uma função que recebe um número inteiro como parâmetro e retorna o valor desse número recebido com seus dígitos invertidos. (b) Escreva um programa que testa a função solicitada no item (a). A seguir, um exemplo de execução desse programa:

```
Digite um valor inteiro: 321
Inteiro invertido: 123

Digite um valor inteiro: -321
Inteiro invertido: -123
```

[**Sugestões**: (1) Use uma variável local à função para indicar se o número é negativo ou não. (2) Use outra variável local para armazenar o valor a ser retornado pela função. Essa variável deve ser iniciada com zero. (3) Se o número for negativo, considere seu valor absoluto. (4) Use um laço **do-while** que encerra quando o parâmetro assumir zero como valor. (5) No corpo desse laço, atualize o valor a ser retornado com o valor corrente desse número vezes `10` mais o resto da divisão do parâmetro por `10`. (6) Atualize o valor do parâmetro com seu valor corrente dividido por `10`. (7) Ao final do laço, retorne o valor obtido, não se esquecendo de corrigi-lo se o valor original do parâmetro for negativo.]

EP5.12 **Preâmbulo**: Um **palíndromo numérico** é um número natural que apresenta o mesmo valor quando lido da esquerda para a direita ou da direita para a esquerda. Por exemplo, `1221` é um palíndromo numérico. **Problema**: Escreva um programa que verifica se um número inteiro positivo introduzido pelo usuário constitui um palíndromo numérico. [**Sugestões**: (1) Use a função `LeNatural()`, definida na **Seção 5.11.1**, para ler o número. (2) Use a função solicitada no exercício **EP5.11** para obter o número invertido. (3) Compare o número introduzido pelo usuário com o número invertido e informe se esse número é um palíndromo ou não.]

EP5.13 Suponha que a taxa média de crescimento populacional de um país seja 1,2% e que ela se mantenha constante entre 2013 e 2023. Suponha ainda que a população atual desse país seja 100.000.000 de habitantes. Escreva um programa que lê um ano entre aqueles mencionados e informa qual é a estimativa populacional para esse ano. [**Sugestões**: (1) Defina constantes simbólicas para representar a taxa de crescimento populacional, a população inicial e os dois anos em questão. (2) Escreva uma função que lê um número inteiro cujo valor esteja entre dois valores recebidos como parâmetros. (3) Para calcular a população esperada no ano especificado pelo usuário, leve em consideração que a população cresce em progressão geométrica tendo como razão a taxa média de crescimento populacional.]

EP5.14 Escreva um programa que desenha setas para direita, esquerda, cima e baixo usando asteriscos, de acordo com a escolha do usuário. A seguir, um exemplo de execução desse programa:

```
1. Seta para esquerda
2. Seta para direita
3. Seta para cima
4. Seta para baixo
5. Encerra o programa

Escolha a opcao: 4
  *
  *
  *
  *
*****
 ***
  *
```

[**Sugestão**: Crie uma função para cada tipo de seta.]

EP5.15 (a) Escreva uma função que escreve um determinado caractere na tela um número específico de vezes. O protótipo dessa função deve ser:

```
void EscreveCaractere(int c, int nVezes)
```

Por exemplo, a chamada `EscreveCaractere('*', 15)` escreveria na tela `15` asteriscos. (b) Escreva um programa que lê um caractere e um valor inteiro positivo introduzidos via teclado, chama a função solicitada no item (a) para apresentar o resultado na tela.

EP5.16 **Preâmbulo**. A função `ComparaDoubles()`, definida na **Seção 5.11.6**, usa a expressão:

```
fabs(d1 - d2) <= DELTA
```

que é a diferença absoluta entre os valores ora comparados. No entanto, para testar se dois valores reais são tão próximos que se pode considerar que eles são iguais, uma comparação mais precisa deveria levar em consideração a diferença relativa entre os valores comparados dada por:

```
fabs(d1 - d2) <= DELTA*Max(fabs(d1), fabs(d2)
```

Nessa expressão, `Max()` é uma função que retorna o maior dentre dois valores do tipo **double** recebidos como parâmetros.

Problema. (a) Implemente uma função, denominada `ComparaDoubles2()`, que leve em consideração a diferença relativa entre os valores comparados. (b) Escreva um programa semelhante àquele da **Seção 5.11.6** que teste a função especificada no item (a). [**Sugestões**: (1) Use a função `ComparaDoubles()` como ponto de partida. (2) Você precisará também implementar a função `Max()` descrita no preâmbulo.]

EP5.17 Escreva um programa que encontra o primeiro número primo que é maior do que ou igual a um número inteiro positivo introduzido pelo usuário. [**Sugestões**: (1) Use a função `LeNatural()` definida na **Seção 5.11.1** para ler o valor introduzido pelo usuário. (2) Use um laço de contagem que encerra quando for encontrado um número primo maior do que ou igual a esse valor. (3) Use a função `EhPrimo()` definida na **Seção 5.11.2** para determinar se um número é primo.]

EP5.18 Escreva um programa que lê um número positivo menor do que `5000` e escreve na tela o número correspondente usando algarismos romanos. [**Sugestões**: (1) Escreva funções para escrita de milhar, centena, dezena e unidade. Em cada uma dessas funções use uma instrução **switch-case** para escolher a instrução de escrita adequada. Por exemplo, na função que exibe a centena do número, se o valor do parâmetro for `1`, será escrito `"C"`; se esse valor for `2`, será escrito `"CC"` e assim por diante. (2) Decomponha o número introduzido pelo usuário conforme visto no exemplo da **Seção 4.11.9** e chame as funções descritas na sugestão (1) para escrita de milhar, centena, dezena e unidade.]

EP5.19 (a) Escreva uma função que determina o número de dígitos de um número inteiro não negativo recebido como parâmetro. [**Sugestões**: (1) Use uma variável local à função para contar o referido número de dígitos e inicie-a com zero. (2) Use um laço **do-while** no corpo do qual o parâmetro é reduzido ao seu valor corrente dividido por `10` e a variável que conta o número de dígitos é incrementada. Esse laço encerra quando esse parâmetro torna-se nulo.] (b) Escreva um programa que lê um valor inteiro não negativo via teclado e informa o número de dígitos do número lido.

EP5.20 (a) Escreva uma função que simula um determinado número de lançamentos de um dado e exibe na tela o percentual de ocorrências de cada face do dado. [**Sugestões**: (1) Use uma variável local para armazenar o número de ocorrências de cada face (no total, serão seis variáveis). (2) Use um laço de contagem cujo número de execuções seja igual ao valor do parâmetro da função. (3) No corpo desse laço use uma instrução **switch-case** cuja expressão seja uma chamada da função `NumeroAleatorio()` definida na **Seção 5.11.8** e cujos valores associados aos casos sejam os valores das faces do dado. Além de **break**, a única instrução associada a cada caso é aquela que incrementa o contador de ocorrências da respectiva face. (4) Após o encerramento do laço, a função apresenta na tela o número de ocorrências de cada face.] (b) Escreva um programa que, repetidamente, lê um valor inteiro introduzido pelo usuário e, ele for positivo, a função solicitada no item (a) é chamada para simular o lançamento do dado o número de vezes especificado. Se o usuário introduzir um valor negativo ou zero, o programa deverá ser encerrado.

EP5.21 **Preâmbulo**: A raiz quadrada de um número real pode ser calculada pelo método iterativo de Newton e Raphson:

$$\sqrt{x_{n+1}} = \frac{x_n^2 + x_0}{2x_n}$$

Nessa fórmula, x_0 é uma estimativa inicial, que pode ser o próprio número cuja raiz se deseja calcular. **Problema**: (a) Escreva uma função que calcula a raiz quadrada de um número real usando o método de Newton e Raphson. [**Sugestões**: (1) Use uma variável local à função para armazenar o resultado e inicie-a com o parâmetro da função. (2) Use um laço **while** no corpo do qual a estimativa da raiz armazenada na variável local é atualizada usando a fórmula de Newton e Raphson. O laço deve encerrar quando a estimativa da raiz é suficientemente próxima do valor real. Use função `ComparaDoubles()` apresentada na **Seção 5.11.6** para comparar o quadrado da estimativa da raiz com o valor recebido como parâmetro.] (b) Escreva uma função **main()** que calcula raízes quadradas usando a função descrita no item (a) e compara os resultados com aqueles obtidos usando a função **sqrt()** declarada em `<math.h>`.

5.13.2 Moderado

EP5.22 (a) Escreva uma função em C com dois parâmetros: (1) um parâmetro do tipo **double** e (2) um parâmetro do tipo **int**. Essa função deverá retornar o valor do primeiro parâmetro elevado ao segundo. Em outras palavras, se o primeiro parâmetro é denominado `x` e o segundo é denominado `n`, essa função deverá retornar o resultado de x^n. (b) Escreva um programa em C que receba como entradas um valor real `x` e um valor inteiro `n`, utilize a função descrita em (a) para calcular x^n e exiba esse resultado na tela. [**Dica**: Esse exercício não tão trivial quanto parece, pois existem várias condições excepcionais como, por exemplo, quando `x = 0` e `y = 0`.]

EP5.23 A **Conjectura de Goldbach** constitui uma das afirmações mais antigas de Matemática que ainda não foram provadas. De acordo com essa conjectura, qualquer número inteiro par maior do que `2` pode ser expresso como a soma de dois números primos. Por exemplo, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$ e assim por diante. Escreva uma função que receba um número inteiro par como parâmetro e produza como saída dois números primos que, quando somados, resultam no número par. O protótipo dessa função deverá ser:

```
int DecomposicaoGoldbach(int n, int *primo1, int *primo2)
```

A função deverá retornar zero se o primeiro parâmetro não for par ou se ele for par e os números primos da decomposição não forem encontrados. Se os números primos da decomposição forem encontrados, a função deverá retornar `1`. (b) Escreva um programa que leia um número inteiro positivo via teclado e apresente os números primos que constituem sua decomposição. [**Sugestão**: Use a função `EhPrimo()` definida no exemplo apresentado na **Seção 5.11.2**.]

EP5.24 Escreva um programa em C que exibe na tela um calendário para qualquer mês a partir do ano de 1899, sabendo que o dia 1° de janeiro de 1899 foi um dia de domingo.

Sugestões:

[1] Esse programa não é tão difícil nem tão trivial quanto pode parecer, mas é bastante trabalhoso. O problema central aqui é determinar quantos dias decorrem de 1° de janeiro de 1899 (data de referência) até o início do mês desejado. Esse problema é complicado pela existência de anos bissextos, mas o programa apresentado na **Seção 4.11.6** mostra como determinar se um ano é bissexto ou não.

[2] Como um ano bissexto contém 366 dias, para calcular o número de dias decorridos desde a data de referência até logo antes do início do mês desejado, deve-se usar a seguinte fórmula:

dias decorridos = *nº de anos comuns* × 365 + *nº de anos bissextos* + *nº de dias até o mês desejado*

Nessa fórmula, tem-se que:

- *dias decorridos* representa o número de dias decorridos desde a data de referência até o início do mês cujo calendário deseja-se exibir.
- *nº de anos comuns* é o número de anos que não são bissextos entre a data de referência e o final do ano anterior àquele do calendário desejado.
- [número de anos bissextos] é o número de anos bissextos entre a data de referência e o final do ano anterior àquele do calendário desejado.
- *nº de dias até o mês desejado* é o número de dias decorridos desde o início do ano do calendário desejado até o último dia do mês anterior ao mês do mesmo calendário.

[3] Conhecendo-se o número de dias decorridos desde a data de referência até logo antes do início do mês desejado e sabendo-se que a data de referência caiu num domingo, o dia da semana em que o referido mês inicia é determinado pelo resto da divisão deste número por 7 mais 1, considerando que domingo é o dia número 1 da semana. Isto é:

dia inicial do mês = *1* + *dias decorridos* % *7*

[4] A última informação necessária para a apresentação do calendário desejado é o número de dias do mês em questão. Esse passo do algoritmo é relativamente trivial e não envolve nenhuma fórmula. (Mas, não esqueça que o número de dias de fevereiro depende do fato de o ano ser bissexto ou não).

EP5.25 Escreva um programa que determina dia da semana de nascimento de uma pessoa nascida entre 1900 e 2014. A data de nascimento da pessoa deve ser lida via teclado. [**Sugestão**: Use as sugestões do exercício **EP5.24**.]