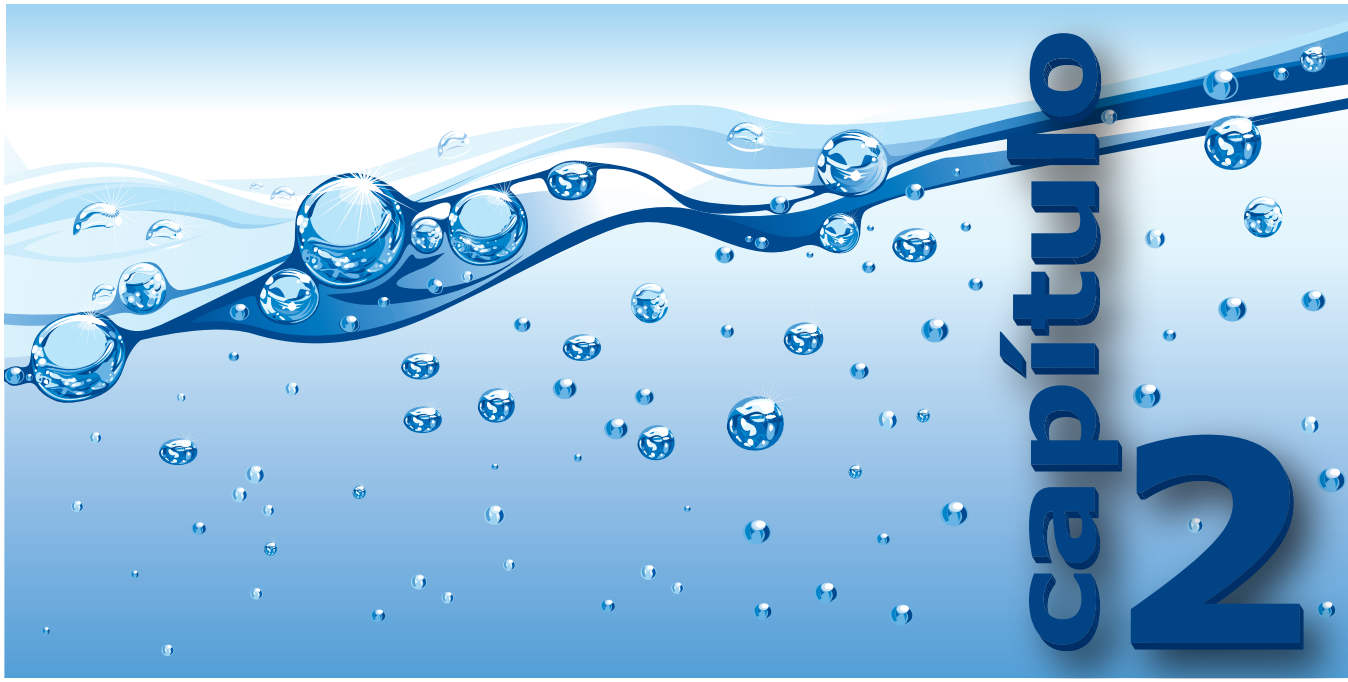


# Funções e Programas Multiarquivo

**Após estudar este capítulo, você deverá ser capaz de:**

- Definir e usar a seguinte terminologia:
  - ❒ Função
  - ❒ Chamada de função
  - ❒ Alusão de função
  - ❒ Protótipo de função
  - ❒ Retorno de função
  - ❒ Duração de variável
  - ❒ Duração fixa
  - ❒ Duração automática
  - ❒ Escopo de bloco
  - ❒ Módulo
  - ❒ Ponteiro
  - ❒ Modo de parâmetro
  - ❒ Buffer de entrada
  - ❒ Compilação condicional
  - ❒ Make
  - ❒ Makefile
  - ❒ Escopo de função
  - ❒ Escopo de arquivo
  - ❒ Parâmetros real e formal
  - ❒ Escopo de identificador
  - ❒ Escopo de programa
  - ❒ Parâmetro de entrada
  - ❒ Parâmetro de saída
  - ❒ Parâmetro de entrada e saída
  - ❒ Passagem de parâmetro
  - ❒ Programa multiarquivo
- Especificar e implementar uma função em C
- Explicar o funcionamento de leitura de dados via teclado
- Discutir como a duração de uma variável afeta seu valor
- Implementar uma macro com parâmetro
- Incluir compilação condicional num programa
- Implementar um programa multiarquivo
- Implementar uma função para entrada de dados robusta

objetivos

ESTE CAPÍTULO PROSSEGUE com a revisão da linguagem C iniciada no capítulo anterior. Desta vez, os seguintes tópicos serão revisados:

- Funções
- Leitura de dados robusta via teclado
- Duração e escopo de variáveis
- Diretivas de pré-processamento
- Macros
- Programas multiarquivo

# 2.1 Funções

Em C, **função** é um subprograma que consiste em um 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.

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).

## 2.1.1 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.

## 2.1.2 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)***

O valor retornado por uma função corresponde ao que ela produz como resultado de seu processamento e que pode ser usado numa expressão. 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**, ela só pode ser chamada isoladamente numa instrução.

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, nunca omita o tipo de retorno 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. Quando uma 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. adiante). Nomes de parâmetros devem seguir as mesmas recomendações de estilo para nomes de variáveis.

Um nome de uma função é um identificador como outro qualquer, 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 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.
- **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.

### 2.1.3 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. Em termos de estilo de programação, é recomendado usar endentação para ressaltar que uma instrução ou declaração faz parte de uma 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.

Em C, funções não podem ser aninhadas. Isto é, uma função não pode ser definida dentro do corpo de outra função. Por outro lado, 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.

### 2.1.4 Retorno de Função

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.

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. Ou seja, uma função com tipo de retorno **void** só pode usar uma instrução **return** como:

```
return;
```

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. Para retornar um valor, utiliza-se uma

instrução **return** seguida do valor que se deseja como 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**.

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.

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. É importante observar que, assim como ocorre com outras formas de conversão implícita, não há arredondamento (v. **Seção 1.6**).

### 2.1.5 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, mas esse valor normalmente é desprezado.

### 2.1.6 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. Também 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 2.4**) 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.
- **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 visto, 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 3**).

[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 3**).

### 2.1.7 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. 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**.

[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. Para precaver-se contra surpresas desagradáveis, 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. Esse cuidado na passagem de parâmetros deve ser redobrado quando se lidam com parâmetros representados por ponteiros.

### 2.1.8 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.
- **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.

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.

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. Mas, mediante o uso de ponteiros e endereços de variáveis pode-se simular passagem por referência em C e modificar valores de variáveis, como mostra a função `Troca()` a seguir.

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

Suponha que a seguinte função **main**() seja utilizada para chamar a função `Troca()`:

```
int main(void)
{
   int i = 2, j = 5;
   printf( "\n\t>>> ANTES da troca <<<\n \n\t    i = %d, j = %d\n", i, j );
   Troca(&i, &j);
   printf( "\n\t>>> DEPOIS da troca <<<\n \n\t    i = %d, j = %d\n", i, j );
   return 0;
}
```

No instante em que a função `Troca()` é chamada pela função **main**() acima, seus parâmetros formais são alocados em memória e cada um deles recebe uma cópia do parâmetro real correspondente. Nesse instante, a situação das variáveis e parâmetros do programa pode ser ilustrada como na **Figura 2–1 (a)**. Observe nessa ilustração que os parâmetros `ptrInt1` e `ptrInt2` recebem cópias dos endereços das variáveis `i` e `j`, conforme indicam as setas que emanam desses parâmetros. Em seguida, a variável `aux` é alocada e recebe o valor do conteúdo apontado pelo parâmetro `ptrInt1` [v. **Figura 2–1 (b)**].

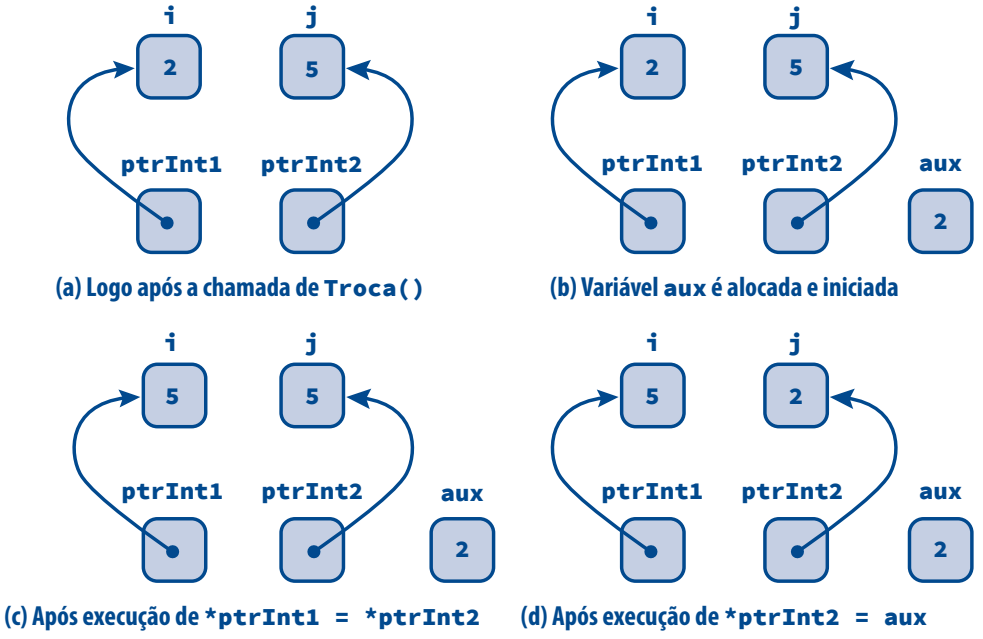


**FIGURA 2–1: SIMULANDO PASSAGEM POR REFERÊNCIA EM C**

A próxima instrução da função `Troca()` a ser executada é: `*ptrInt1 = *ptrInt2` e de acordo com essa instrução, o conteúdo apontado por `ptrInt1` recebe o conteúdo apontado por `ptrInt2`, o que equivale a substituir o valor de `i` pelo valor de `j` e a situação em memória passa a ser aquela mostrada na **Figura 2–1(c)**. Finalmente, a última instrução da função `Troca()` 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 2–1(d)**.

Ao encerramento da execução da função `Troca()`, os espaços alocados para os seus parâmetros e para a variável local `aux` são liberados (v. **Seção 2.3**), restando em memória apenas as variáveis `i` e `j`, que foram alocadas na função **main**().

### 2.1.9 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 e pode-se, ainda, iniciar a alusão com a palavra-chave **extern**. Portanto uma alusão deve ter o seguinte formato:

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

Por exemplo, a função `Troca()` apresentada acima poderia ter a seguinte alusão:

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

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

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

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. No caso de alusão, o uso de **void** entre parênteses 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.

## 2.2 Leitura de Dados via Teclado 2: Robusta

### 2.2.1 Entendendo Leitura de Dados via Teclado

Na prática, usar corretamente uma função da biblioteca padrão que faz leitura via teclado não é tão trivial quanto aparenta. As dificuldades, nesse caso, residem no fato de a leitura de dados depender do comportamento do usuário do programa, o que nem sempre corresponde àquilo que o programa espera.

Leitura de dados para programas interativos baseados em console é efetuada por meio do teclado. Além disso, quaisquer que sejam as teclas pressionadas pelo usuário durante uma leitura de dados, apenas caracteres são enviados para um programa. Por exemplo, mesmo que o usuário tecle apenas dígitos, o programa não recebe números diretamente do teclado. Nesse caso, o programa só receberá um número se houver uma função que leia os caracteres digitados e converta-os num número. A função **scanf()**, por exemplo, é capaz de realizar essa tarefa.

Outro ponto importante que deve ser ressaltado é que, quando uma função que efetua leitura via teclado [p. ex., **getchar()**] é executada, ela não tenta ler caracteres diretamente nesse dispositivo de entrada. Ao invés disso, a leitura é feita num buffer, que é uma área de armazenamento temporário para a qual os caracteres introduzidos pelo usuário são enviados e enfileirados após o usuário pressionar a tecla `[ENTER]`. Além disso, uma função que efetua leitura no meio de entrada padrão só causa interrupção do programa à espera da introdução de caracteres se ela não encontrar nenhum deles no buffer associado a esse meio de entrada (mas há exceções, conforme será visto adiante).

Compreender bem o funcionamento do buffer associado ao teclado é fundamental para lidar com possíveis problemas com leitura de dados. Para começar a entender o funcionamento da leitura de dados via teclado, considere, por exemplo, o seguinte programa:

```
#include <stdio.h>

int main(void)
{
   int umChar;

   printf("\nDigite um caractere: ");
   umChar = getchar();

   printf("\nDigite outro caractere: ");
   umChar = getchar();

   return 0;
}
```

Apesar de simples, esse programa causa enorme frustração ao programador iniciante que não entende o funcionamento da leitura de dados via teclado. Tudo que esse programa faz é solicitar que o usuário introduza dois caracteres e ler esses caracteres usando a função **getchar()**. O que o programador certamente deseja que aconteça é que o usuário digite cada caractere quando solicitado pelo programa, mas não é isso que ocorre.

Para sentir melhor o que se está afirmando, execute o último programa e tente introduzir o caractere `A` quando for escrito na tela `Digite um caractere:` e o caractere `B` quando for escrito `Digite outro caractere:`. Se você seguir essas recomendações, verá que, quando tentar introduzir o caractere `B`, o programa já terá encerrado. Por que o programa age dessa maneira? A resposta está no fato de a primeira leitura ter deixado caracteres remanescentes no buffer associado ao teclado e um desses caracteres ser lido na segunda leitura. Se você não consegue entender por que isso acontece, procure um bom texto sobre programação em C. Aqui apenas se demonstrará como resolver esse tipo de problema.

É importante ressaltar que, além de **getchar()**, outras funções de leitura podem deixar caracteres remanescentes no buffer que podem causar o mesmo problema em tentativas subsequentes de leitura.

### 2.2.2 Esvaziamento do Buffer de Entrada

A conclusão que se obtém da discussão apresentada acima é que, antes de fazer uma leitura via teclado, é importante garantir que buffer associado a esse meio de entrada esteja vazio. Uma maneira de realizar isso é por meio de uma chamada da função `LimpaBuffer()` definida como:

```
int LimpaBuffer(void)
{
   int carLido, /* Armazena cada caractere lido */
       nCarLidos = 0; /* Conta o número de caracteres lidos */

      /* Lê e descarta cada caractere lido até    */
      /* encontrar '\n' ou getchar() retornar EOF */
   do {
      carLido = getchar(); /* Lê um caractere */
      ++nCarLidos; /* Mais um caractere foi lido */
   } while ((carLido != '\n') && (carLido != EOF));

      /* O último caractere lido foi '\n' ou  */
      /* EOF e não deve ser considerado sobra */
   return nCarLidos - 1;
}
```

A função `LimpaBuffer()` lê e descarta repetidamente todos os caracteres encontrados no buffer de entrada até encontrar o caractere `'\n'` ou **getchar()** retornar **EOF** (v. **Seção 1.11**). A função `LimpaBuffer()` retorna o número de caracteres descartados, com exceção do caractere `'\n'` ou **EOF**.

Para resolver problemas como aquele apresentado no programa acima, deve-se acrescentar a função `LimpaBuffer()` a tal programa e inserir uma chamada dessa função entre duas operações consecutivas de leitura, como mostra a seguinte função **main**().

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

   printf("\nDigite um caractere: ");
   umChar = getchar();

      /* Verifica se foi lido algum caractere */
      /* e, se for o caso, limpa o buffer     */
   if (umChar !=  EOF)
      (void) LimpaBuffer();

   printf("\nDigite outro caractere: ");
   umChar = getchar();

   return 0;
}
```

### 2.2.3 Uso de Laços de Repetição em Leitura de Dados

Laços de repetição são utilizados em leitura de dados para oferecer novas chances ao usuário após ele ter introduzido dados considerados inusitados pelo programa. Para entender como isso funciona, considere a função `LeInteiro()` para leitura de inteiros apresentada a seguir.

```
int LeInteiro(void)
{
   int num, /* O número lido */
       teste, /* Valor retornado por scanf() */
       nResto = 0; /* Número de caracteres excedentes */

   /* Desvia para cá se o valor for inválido */
inicio:
   teste = scanf("%d", &num); /* Tenta ler um valor válido */

      /* Se não ocorreu erro de leitura ou de final de arquivo, */
      /* há caracteres remanescentes que precisam ser removidos */
   if (teste != EOF)
      nResto = LimpaBuffer();

      /* Enquanto o valor retornado por scanf()  */
      /* não indicar que um valor válido foi     */
      /* lido continua tentando obter esse valor */
   while(teste != 1) {
      printf( "\a\n\t>>> O valor digitado e' invalido. Tente novamente\n\t> " );
      goto inicio; /* Não causa dano algum */
   }

      /* Repreende o usuário se ele digitou demais */
   if (nResto == 1)
      printf("\t>>> Um caractere foi descartado\n");
   else if (nResto > 1)
      printf("\t>>> %d caracteres foram descartados\n", nResto);

      /* O valor retornado certamente é válido */
   return num;
}
```

## 2.3 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.

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. adiante), 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. 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.

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**.

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, como, por exemplo, na seguinte função:

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

    ++contador;

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

Uma variável de duração fixa é iniciada apenas uma vez, ao passo que uma variável de duração automática é iniciada sempre que o bloco que contém sua definição é executado. 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:

A **Tabela 2–1** 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 2–1: Iniciação de Variáveis de Durações Automática e Fixa**

## 2.4 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.

- **Escopo de Programa**. Um identificador com esse tipo de 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 por **static**. Qualquer função que não seja precedida por **static** também tem esse tipo de escopo.
- **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 por **static** têm esse tipo de escopo. Funções cujos cabeçalhos sejam qualificados com **static** também têm escopo de arquivo. Identificadores associados a tipos de dados definidos pelo programador (v. **Seção 3.9**) e identificadores associados a constantes simbólicas (v. **Seção 1.12**) possuem esse tipo de escopo.
- **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 1.15.3**), 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.
- **Escopo de bloco**. Um identificador com esse tipo de escopo 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. Mas 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. Também é 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 utilizado.

## 2.5 Diretivas de Pré-processamento

O **pré-processador** de C é um programa conceitualmente independente do compilador que prepara arquivos-fonte para compilação. Esse programa utiliza uma linguagem própria, cujas instruções, denominadas **diretivas**, começam sempre com o símbolo `#`.

Uma diretiva pode aparecer em qualquer linha de um programa-fonte e termina quando se passa para a próxima linha. Quando se deseja que uma diretiva ocupe mais de uma linha, utiliza-se uma barra invertida para indicar esse fato. Por exemplo:

```
#define UMA_CONSTANTE_SIMBOLICA_MUITO_LONGA "Este e' um exemplo de \
diretiva que ocupa duas linhas"
```

Compiladores antigos requerem que qualquer diretiva comece sempre com o símbolo # escrito na primeira coluna e que não exista espaço entre esse símbolo e a primeira palavra da diretiva. Apesar de essas restrições não mais existirem em compiladores mais modernos, essa forma de escrever diretivas ainda é comum.

As seguintes facilidades providas pelo pré-processador de C serão discutidas nesta seção:

- Processamento de macros e expressões contendo o operador # (v. **Seção 2.5.1**)
- Compilação condicional (v. **Seção 2.5.2**)
- Inclusão de arquivos (v. **Seção 2.5.3**)

### 2.5.1 Macros

Uma **macro** é um identificador que possui usualmente uma sequência de símbolos associada denominada **corpo da macro**. Basicamente, existem dois tipos de macros: **sem parâmetros** e **com parâmetros**.

Convencionalmente, nomes de macros são constituídos apenas de letras maiúsculas e subtraço, pois esta convenção facilita a identificação visual de macros espalhadas pelo programa.

Macros são usualmente definidas no início de um arquivo de programa ou em qualquer local de um arquivo de cabeçalho e têm escopo de arquivo. Se você deseja que uma macro seja utilizada em vários arquivos, coloque-a num arquivo de cabeçalho e inclua-o nos arquivos em que desejar tê-la disponível.

Uma macro sem parâmetros corresponde exatamente a uma constante simbólica (v. **Seção 1.12**). Quando o nome de uma macro aparece num local diferente daquele de sua definição, o nome é substituído *literalmente* pelo corpo da macro. Essa substituição é denominada **expansão de macro** e já foi discutido na **Seção 1.12**.

Uma definição de macro com parâmetros utiliza a seguinte sintaxe

**#define** ***nome-da-macro(parâmetros-da-macro) corpo-da-macro***

Os parâmetros de uma macro devem ser separados por vírgulas e usualmente são representados por letras minúsculas. Como exemplo de macro com parâmetro, considere a seguinte definição de macro:

```
#define QUADRADO(a) ((a)*(a))
```

A diretiva apresentada acima define a macro `QUADRADO` com um parâmetro e tem por objetivo definir o quadrado de um número.

Uma macro com parâmetros é utilizada da mesma forma que uma chamada de função. Por exemplo, na instrução:

```
x = QUADRADO(5);
```

a macro `QUADRADO` seria expandida pelo pré-processador, resultando em:

```
x = ((5)*(5));
```

O uso de parênteses envolvendo cada uso de parâmetro no corpo da macro, assim como envolvendo todo o corpo da macro, como no exemplo acima, é altamente recomendável para evitar problemas no uso da macro.

O corpo de uma macro pode conter uma ou mais instruções ou declarações de variáveis. Quando o corpo de uma macro possui mais de uma instrução ou definição de variável, é comum utilizar-se o seguinte artifício:

```
do {instruções-e-declarações} while (0)
```

Esse artifício permite que as instruções e declarações que constituem o corpo da macro sejam encapsuladas numa única instrução. Assim a macro pode ser invocada terminando com ponto e vírgula, sem que esse símbolo seja interpretado como instrução vazia. Por exemplo, considere a definição de macro a seguir:

```
#define ATRIBUI(a, b) {a = 5; b = -5;}
```

Quando esta macro for invocada como:

```
ATRIBUI(x, y);
```

o ponto e vírgula utilizado nesta chamada representa a instrução vazia, pois, quando a macro for expandida, ele sucederá o fecha-chaves. Por outro lado, se a macro for definida como:

```
#define ATRIBUI(a, b) do {a = 5; b = -5;} while (0)
```

a expansão da mesma chamada resulta em:

```
do {x = 5; y = -5;} while (0);
```

e o ponto e vírgula será considerado normalmente como um terminal de instrução.

Normalmente, macros são executadas mais rapidamente do que funções equivalentes, mas nem sempre o uso de macros é mais indicado do que o uso de funções. De fato, algumas vezes, é difícil decidir se uma dada operação deve ser implementada como função ou como macro. A seguir serão enumeradas vantagens e desvantagens decorrentes do uso de macros em comparação com funções.

As principais vantagens decorrentes do uso de macros são:

- **Macros são mais eficientes**. Quando invocadas, elas não demandam operações de empilhamento e desempilhamento de variáveis locais e parâmetros como ocorre em chamadas de funções.
- **Uma macro pode servir para vários tipos de dados**, pois os parâmetros de uma macro não têm tipos definidos.

Por outro lado, macros apresentam as seguintes desvantagens:

- **Um parâmetro é avaliado cada vez que é utilizado no corpo de uma macro**. Isso pode levar a resultados inesperados quando a macro é invocada com parâmetros contendo operadores com efeitos colaterais.
- **O corpo de uma função é compilado apenas uma vez, enquanto o corpo de uma macro é compilado em cada instrução na qual ela é invocada**. Assim o código executável de um programa contendo muitas macros pode ocupar muito mais espaço em memória do que aquele de um programa que utiliza funções equivalentes.
- **Macros são processadas pelo pré-processador e pelo compilador, enquanto funções são processadas apenas pelo compilador**. Isso torna mais difícil depurar programas contendo macros com relação àqueles que contêm funções equivalentes.
- **Macros não checam os tipos de seus parâmetros**, o que torna o uso de macros mais sensível a erros do que o uso de funções.

- **Macros não podem ser chamadas recursivamente**. Assim elas não servem para implementar algoritmos recursivos (v. **Capítulo 4**).
- **Macros não possuem endereços**. Assim não podem ser representadas por ponteiros (v. **Capítulo 9**).

Você deve fazer um balanço entre vantagens e desvantagens antes de decidir utilizar uma macro em substituição a uma função numa situação particular. O impacto decorrente da substituição de funções por macros em termos de rapidez de execução só será percebido se a função substituída for chamada com muita frequência.

Quando colocado antes de um parâmetro no corpo de uma macro, o símbolo #, que representa um operador de pré-processamento, faz com que o pré-processador envolva o parâmetro com aspas que resultam num string constante em C. O uso do operador # para produção de strings é indicado quando se deseja que um dado parâmetro de uma macro seja tratado tanto como uma expressão quanto como um string, como na seguinte macro:

```
#define ESCREVE_VALOR(x) printf("Valor de %s = %f\n", #x, x)
```

Quando essa macro é invocada, como no trecho de programa a seguir:

```
double x = 1.0, y = 2.0, z = 3.0;
...
ESCREVE_VALOR(x + y + z);
```

o pré-processador expande as chamadas da macro `ESCREVE_VALOR` como:

```
printf("Valor de %s = %f\n", "x + y + z", x + y + z);
```

É permitido concatenar strings constantes com strings produzidos por meio do operador #, colocando-os justapostos. Utilizando esse conhecimento a macro apresentada acima poderia ter sido definida como:

```
#define ESCREVE_VALOR2(x) printf("Valor de " #x " = %f\n", x)
```

O corpo de uma macro pode ser vazio dando origem a uma **macro vazia**. A expansão de uma macro vazia não resulta em nada e essas macros são úteis apenas quando utilizadas em conjunto com compilação condicional (v. mais adiante). Pode-se, por exemplo, utilizar uma macro vazia para indicar que um programa está em fase de depuração. Por exemplo:

```
#define EM_DEPURACAO
```

Neste exemplo, o que interessa apenas é o fato de a macro ser definida; o valor da macro em si não importa.

### 2.5.2 Compilação Condicional

Compilação condicional permite que certos trechos de um programa sejam compilados ou não de acordo com o valor de uma ou mais condições. Isto pode ser obtido por meio de dois conjuntos de diretivas similares à instrução condicional **if-else** de C. Um uso comum de compilação condicional é em prevenção de inclusão múltipla de arquivos de cabeçalho (v. adiante). Compilação condicional também é particularmente útil durante o estágio de depuração de um programa (consulte a **Bibliografia**).

As diretivas **#if**, **#else**, **#elif** e **#endif**, possuem a sintaxe apresentadas abaixo. A expressão condicional que segue um **#if** ou **#elif** deve ser uma constante (usualmente representada por uma macro sem parâmetros) e sua interpretação é semelhante àquela vista anteriormente para expressões condicionais em C. Mas a diferença mais importante entre blocos de diretivas **#if** e instruções **if-else** é que essas diretivas não determinam se uma instrução será executada ou não: *elas apenas especificam aquilo que será efetivamente compilado posteriormente pelo compilador*.

```
#if condição1
    trecho de programa a ser compilado se condição1 for satisfeita
#elif condição2
    trecho de programa a ser compilado se condição2 for satisfeita
  ...
#elif condiçãoN
    trecho de programa a ser compilado se condição N for satisfeita
#else
    trecho de programa a ser compilado se nenhuma condição for satisfeita
#endif
```

O pré-processador expande macros antes da avaliação de qualquer bloco de diretivas **#if**. Além disso, se uma expressão condicional contém um nome que não tenha sido definido ainda, ele será expandido em zero.

Pode-se especificar compilação condicional com base na existência ou não de uma macro (independentemente do valor da macro) utilizando, respectivamente, as diretivas **#ifdef** ou **#ifndef**. Como exemplo de compilação condicional com o uso da diretiva **#ifdef**, considere o seguinte trecho de programa:

```
#ifdef TESTE
   printf("Isto é um teste.\n");
#else
   printf("Isto não é um teste.\n");
#endif
```

No último exemplo, se a macro `TESTE` for definida com qualquer valor ou mesmo for uma macro vazia, a primeira instrução **printf()** será compilada; caso contrário, a segunda chamada de **printf()** será compilada.

### 2.5.3 Inclusão de Arquivos

A diretiva **#include** para inclusão de arquivos já foi suficientemente discutida (v. **Seção 1.9**). Aqui será apresentado um esquema que previne a inclusão múltipla de um arquivo de cabeçalho e que constitui um exemplo prático de uso de diretivas de compilação condicional e de macros vazias.

Prevenir a inclusão múltipla de um arquivo de cabeçalho é útil, pois economiza tempo de processamento do arquivo e, às vezes, é realmente necessário. Por exemplo, se o arquivo multiplamente incluído contiver uma definição de tipo (v. **Seção 3.9**), o compilador indicará erro quando encontrar mais de uma definição do mesmo tipo.

Suponha que um arquivo denominado `c1.h` inclua um arquivo denominado `c2.h` e que um arquivo denominado `arq.c` inclua ambos `c1.h` e `c2.h`. Como `c2.h` já é incluído em `c1.h`, `c2.h` seria incluído duas vezes em `arq.c` na ausência de uma estratégia que previna inclusão múltipla de arquivos. Essa situação é representada esquematicamente na **Figura 2–2**.

Supondo que um arquivo de cabeçalho é denominado `Arq1.h`, a estratégia utilizada para prevenir sua inclusão múltipla é delineada a seguir:

```
#ifndef _Arq1_H_ /* Arq1.h é o nome do arquivo de cabeçalho */
#define _Arq1_H_
/* O conteúdo do arquivo aparece aqui */
#endif
```

Utilizando-se essa estratégia, se um arquivo tenta incluir o cabeçalho `Arq1.h` sem tê-lo incluído antes, a macro `_Arq1_H_` não foi ainda definida. Logo esta macro é definida na segunda linha do arquivo `Arq1.h` como uma macro vazia e a inclusão do arquivo ocorre normalmente. Agora, suponha que um arquivo tente incluir

o arquivo `Arq1.h` mais de uma vez. Neste caso, como a macro `_Arq1_H_` foi definida na primeira inclusão do arquivo `Arq1.h`, a diretiva de compilação condicional na primeira linha deste arquivo faz com que o restante do conteúdo do arquivo seja saltado, evitando assim a inclusão múltipla. Note que o sucesso dessa estratégia depende de uma boa escolha para o nome da macro de controle (aqui denominada `_Arq1_H_`) de modo que ele não coincida com o nome de nenhuma outra macro do programa. Usualmente, essa macro é escolhida como uma combinação do nome do arquivo com subtraços.

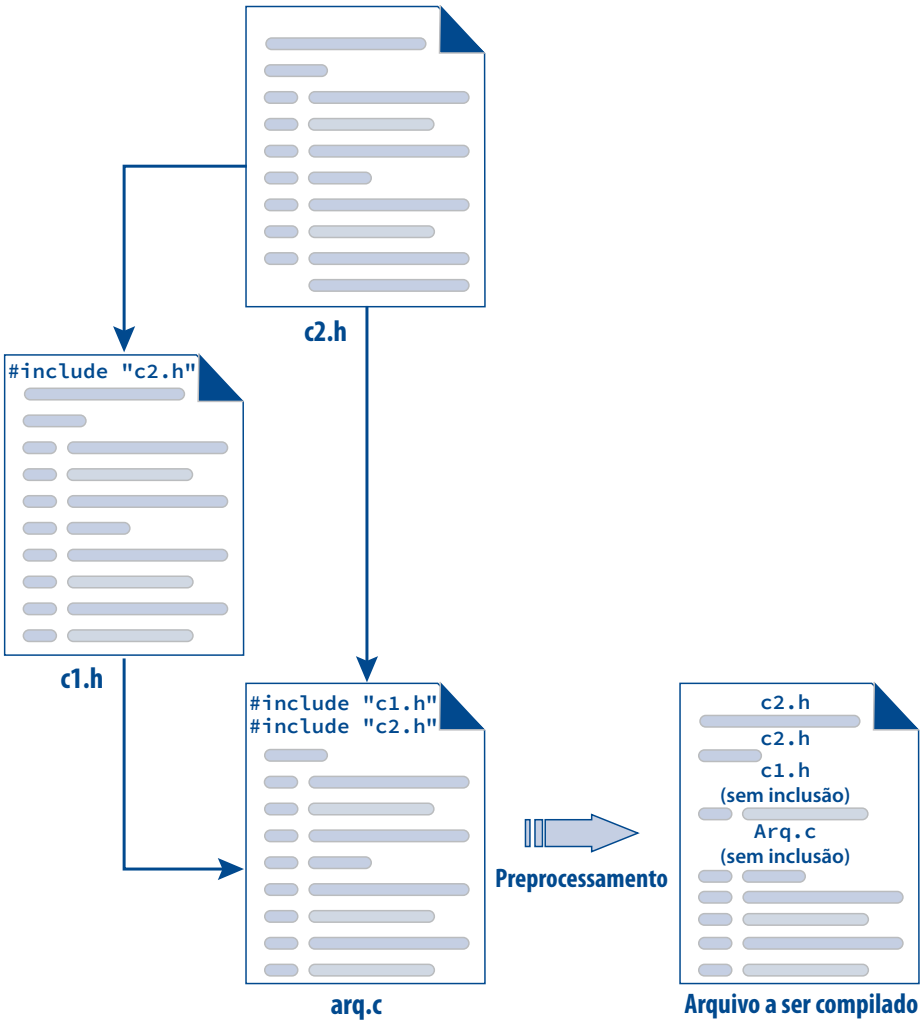


**Figura 2–2: Inclusão Múltipla de um Arquivo de Cabeçalho**

# 2.6 Programas Multiarquivo

Muitos programas práticos são constituídos de vários arquivos-fonte. Esta seção apresenta técnicas para a construção de programas multiarquivo.

## 2.6.1 Variáveis Globais

Em geral, deve-se evitar o uso de variáveis globais tanto quanto possível, pois isto resulta em programas mais complexos e difíceis de ser mantidos. Mas, quando seu uso é justificável, o programador deve adotar certas precauções, como será visto a seguir.

O uso de variáveis globais deve seguir alguma convenção de nomes que as tornem distintas das demais. Uma convenção frequentemente utilizada é iniciar o nome de cada variável global com a letra **g** (por exemplo, `gMinhaGlobal`).

Em C, variáveis globais aparecem sob formas de definições e alusões. A definição de uma variável global, que deve ser única em todo o programa, é responsável por sua alocação em memória. Por outro lado, uma alusão a uma variável global pode aparecer em vários arquivos que fazem parte de um programa e é similar a uma definição de variável, mas não causa alocação de memória para a variável. Em vez disto, uma alusão serve para informar o compilador que a variável aludida é uma variável global definida em outro ponto do programa (talvez, em outro arquivo). Uma alusão à variável utiliza a palavra-chave **extern** seguida do tipo e do nome da variável. Por exemplo:

```
extern int gMinhaGlobal;
```

Do mesmo modo que ocorre com alusões a funções, o uso de **extern** precedendo uma alusão de variável é opcional. No entanto, no caso de variáveis globais, a ausência de **extern** numa alusão de variável conduz a ambiguidade. Por exemplo, a declaração:

```
int gMinhaGlobal;
```

pode tanto representar uma alusão quanto uma definição de variável. Portanto o programador deve adotar consistentemente em seus programas a estratégia delineada na **Tabela 2–2**.

| | Definição | Alusão |
|---|---|---|
| **extern** | Não | Sim |
| **Iniciação** | Sim | Não |

**Tabela 2–2: Definição versus Alusão de Variável Global**

### 2.6.2 Funções de Arquivo e Globais

Em programas distribuídos em múltiplos arquivos-fonte, alguns arquivos constituintes do programa contêm várias funções com afinidades entre si. Algumas dessas funções podem ser necessárias apenas no arquivo em que elas são definidas, enquanto outras precisam ser chamadas em outros arquivos do programa. Uma função que é necessária apenas no arquivo que contém sua definição é denominada **função local**, ao passo que uma função que é chamada em outro arquivo diferente daquele que contém sua definição é chamado de **função global**.

Pode-se especificar se uma função será local ou global precedendo-se seu cabeçalho respectivamente pela palavra-chave **static** ou **extern**. Na ausência de uma dessas palavras-chave na definição da função, o especificador **extern** é assumido. Por isso, **extern** é muito raramente utilizado nesse contexto.

### 2.6.3 Módulos

A partir de um certo tamanho, um programa deve ser dividido em unidades denominadas **módulos**. Em C, um módulo é dividido em duas partes:

[1] **Arquivo de cabeçalho** — comumente esses arquivos têm a extensão `.h` (p. ex., `Interface.h`)

[2] **Arquivo de programa** — comumente esses arquivos têm a extensão `.c` (p. ex., `Interface.c`)

Existem algumas exceções a essa regra de divisão. Uma delas diz respeito ao módulo que contém a função **main**(), que possui apenas arquivo de programa (tipicamente denominado `main.c`) contendo essa função.

O conteúdo típico de um arquivo de cabeçalho deve ser o seguinte:

- Alusões de funções (v. **Seção 2.1**)
- Alusões de variáveis globais (v. **Seção 2.6.1**)
- Definições de tipos (v. **Seção 3.9**)
- Definições de macros (v. **Seção 2.5.1**)

Esses componentes devem ter alguma afinidade entre si e devem gerar instruções em linguagem de máquina. O objetivo de um arquivo de cabeçalho é tornar seus componentes disponíveis para outros arquivos que fazem parte do programa. Para tal, basta que o arquivo que deseje utilizar esses componentes inclua o arquivo de cabeçalho por meio de uma diretiva **#include**.

Um programa de grande porte pode conter arquivos de cabeçalho contendo apenas definições de tipos ou macros utilizadas por vários arquivos que constituem o programa. Neste caso, o módulo é constituído apenas pelo arquivo de cabeçalho (i.e., não há arquivo de programa correspondente).

Um arquivo de programa recebe esta denominação porque seus componentes geram código; i.e., contribuem para o programa executável. Portanto um arquivo de programa deve conter definições de variáveis e funções. Mas, um arquivo de programa pode também conter qualquer componente típico de um arquivo de cabeçalho. Nesse caso, o componente só poderá ser utilizado nesse arquivo de programa. É importante ressaltar que um arquivo de programa (i.e., com extensão `.c`) não deve jamais ser incluído em outro arquivo que constitui o programa.

A **Tabela 2–3** resume as diferenças entre arquivos de programa e arquivos de cabeçalho.

| Arquivo de cabeçalho | Arquivo de programa |
|---|---|
| Não gera código | Gera código |
| Extensão usual: `.h` | Extensão usual: `.c` |
| Deve ser incluído por outros arquivos | Nunca deve ser incluído por outros arquivos |

**Tabela 2–3: Diferenças entre Arquivo de Cabeçalho e Arquivo de Programa**

### 2.6.4 Como Construir Programas Multiarquivo

Qualquer que seja a metodologia empregada na construção de programas multiarquivo, cada arquivo que constitui o programa é compilado separadamente produzindo seu próprio código objeto (não executável). Então, estes arquivos são combinados pelo editor de ligações (linker) para resultar num programa executável, conforme ilustrado na **Figura 2–3**.

#### *Utilizando um Ambiente Integrado de Desenvolvimento*

A maioria dos ambientes de desenvolvimento (IDE) modernos (p. ex., CodeBlocks, Eclipse, Netbeans), utiliza o conceito de **projeto** para organizar os arquivos que compõem um programa multiarquivo. A **Figura 2–4** mostra um exemplo de projeto no ambiente CodeBlocks. Se você ainda não sabe como criar um programa multiarquivo no ambiente de desenvolvimento de sua preferência, recomenda-se que você dedique um pouco de seu tempo para dominar essa técnica, pois os resultados serão compensadores.

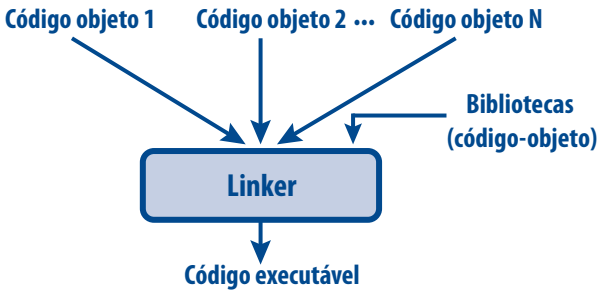


**Figura 2–3: Edição de Ligações de um Programa Multiarquivo**

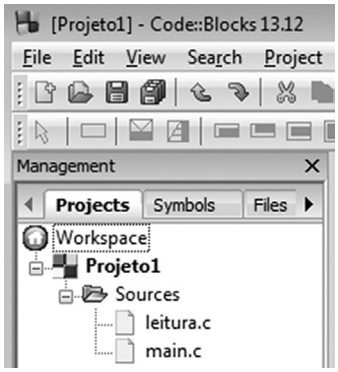


**Figura 2–4: Árvore de Projeto Multiarquivo no IDE CodeBlocks**

Uma vez que você tenha aprendido a montar seu projeto multiarquivo em seu IDE, transformar seu projeto em programa executável pode ser tão simples quanto um mero clique de mouse.

### *Utilizando Editor de Programas e Compilador*

Na construção de arquivos-fonte de um programa multiarquivo, pode-se utilizar, em vez de IDE, um editor de programas (p. ex., Notepad++, TextWrangler). Já o uso do compilador GCC para criação de um programa executável resultante de um programa multiarquivo requer o entendimento de algumas opções de compilação adicionais.

Para compilar e fazer as devidas ligações de um programa composto dos arquivos `arq1.c`, `arq2.c`, ..., `arqN.c` invoque o compilador GCC como:

```
gcc arq1.c arq2.c ... arqN.c
```

Ou como:

```
gcc arq1.c arq2.c ... arqN.c -o arqExec
```

A diferença entre esses dois comandos é que, no segundo comando, o nome do arquivo executável é especificado. Se esse nome não for explicitamente especificado, o nome do executável será `a.out` ou `a.exe`.

No caso de compilação e ligação separadas, é necessário compilar (literalmente) cada arquivo que compõe o programa separadamente, conforme foi visto anteriormente:

```
gcc -c arq1.c
gcc -c arq2.c
...
gcc -c arqN.c
```

Em seguida, invoca-se o *linker* para fazer as ligações e produzir um arquivo executável do seguinte modo:

```
gcc arq1.o arq2.o ... arqN.o
```

Ou:

```
gcc arq1.o arq2.o ... arqN.o -o arqExec
```

A vantagem desse último método em comparação ao método anterior é que, se apenas um arquivo precisar ser modificado após todos terem sido compilados, você só precisará recompilar esse arquivo e invocar o linker para refazer as ligações.

### Make e Arquivos Makefiles

**Make** é um programa utilitário encontrado em sistemas operacionais da família Unix e distribuído junto com alguns ambientes de desenvolvimento. Na ausência de um ambiente IDE, esse utilitário pode facilitar bastante a construção (i.e., compilação e ligação) de programas multiarquivo. O utilitário *make* é particularmente útil quando utilizado na construção de programas que consistem de muitos arquivos, pois ele recompila apenas os arquivos que realmente precisam ser recompilados.

**Makefile** é um arquivo escrito numa linguagem própria que o programa *make* entende. Quando o utilitário *make* é executado sem informação sobre qual arquivo processar, ele procura um arquivo denominado `Makefile` ou `makefile` no diretório corrente. Se o arquivo a ser processado tiver um outro nome, ele precisa ser especificado na invocação de *make*.

Os principais componentes de um arquivo *makefile* são regras que assumem o seguinte formato.

***alvo:dependências***
***[TAB]comando$_1$***
***[TAB]comando$_2$***
***...***
***[TAB]comando$_N$***

em que:

- ***alvo*** é o alvo que a regra representa. Usualmente, um alvo consiste num nome de arquivo resultante do processamento de um programa (por exemplo, um arquivo gerado por um compilador).
- ***dependências*** representam nomes de arquivos ou alvos utilizados em outras regras. Quando há mais de uma dependência, elas devem ser separadas por espaços em branco e, quando não há nenhuma dependência, o espaço reservado para dependências deve ser deixado em branco.
- ***comando$_1$***, ..., ***comando$_N$*** são comandos do sistema operacional que serão executados quando cada uma das dependências (se existir alguma) for satisfeita. É importante notar que precedendo cada comando deve existir um caractere de tabulação (representado por ***[TAB]*** no esquema acima). Portanto, se seu editor de texto transforma tabulações em espaços em branco, desabilite esta opção.

Tipicamente, um comando faz parte de uma regra com dependências e serve para criar o arquivo que representa o alvo da regra quando alguma dependência é alterada. Os comandos são executados apenas quando todas as respectivas dependências são satisfeitas. Quando uma dependência consiste num arquivo, ela será satisfeita se a data da última modificação do arquivo for mais recente do que a data da última modificação do alvo. Tipicamente, arquivos-objeto são considerados dependentes de arquivos-fonte e estes são considerados dependentes dos arquivos de cabeçalho que eles incluem. Por exemplo, suponha que você tenha um programa

multiarquivo contendo os arquivos: `arq1.c`, `arq1.h`, `arq2.c`, `arq2.h` e `main.c`. Suponha ainda que o nome desejado para o executável seja `MeuProg` e que os arquivos de cabeçalho sejam incluídos pelos arquivos de programa de acordo com a tabela a seguir:

| Arquivo de programa | Inclui arquivo de cabeçalho... |
|---|---|
| `arq1.c` | `arq1.h` |
| `arq2.c` | `arq2.h` |
| `main.c` | `arq1.h` e `arq2.h` |

Então, poder-se-ia escrever o seguinte arquivo *makefile* para automatizar a criação do programa executável desejado:

```
MeuProg: main.o arq1.o arq2.o
	gcc main.o arq1.o arq2.o -o MeuProg
main.o: main.c arq1.h arq2.h
	gcc -c main.c -o main.o
arq1.o: arq1.c arq1.h
	gcc -c arq1.c -o arq1.o
arq2.o: arq2.c arq2.h
	gcc -c arq2.c -o arq2.o
```

A primeira regra do arquivo *makefile*:

```
MeuProg: main.o arq1.o arq2.o
	gcc main.o arq1.o arq2.o -o MeuProg
```

informa o utilitário *make* que o alvo principal do arquivo é `MeuProg`. Esse alvo tem três dependências: `main.o`, `arq1.o` e `arq2.o`, cada uma das quais é tanto um nome de arquivo resultante de compilação quanto um alvo de regras subsequentes. O comando associado ao alvo principal será executado se cada um destes arquivos existe e pelo menos um deles é mais recente do que o alvo `MeuProg`.

Considere agora o pré-requisito `main.o` do alvo principal. Se esse arquivo não existir, o utilitário *make* tentará obtê-lo utilizando a regra que tem esse pré-requisito como alvo:

```
main.o: main.c arq1.h arq2.h
	gcc -c main.c -o main.o
```

Esse alvo tem três dependências: `main.c`, `arq1.h` e `arq2.h`, cada uma das quais é um nome de arquivo-fonte. Se algum deles não for encontrado, o alvo `main.c` não poderá ser criado; consequentemente, o comando associado ao alvo principal também não será executado. Por outro lado, se os três arquivos que compõem as dependências do alvo `main.c` existem, o comando:

```
gcc -c main.c -o main.o
```

será executado resultando no arquivo `main.o`. Assim, se as demais dependências da regra associadas ao alvo principal (i.e., `MeuProg`) forem satisfeitas, o comando associado a esta regra será executado.

Agora, suponha que, enquanto avalia a primeira regra, o utilitário *make* descobre que o arquivo `main.o` existe. Como também existe uma regra que especifica como esse arquivo pode ser obtido, o utilitário examinará esta regra para verificar se o arquivo precisa ser reconstruído. Assim, se todos os arquivos que constituem as dependências do alvo `main.o` existirem e algum deles for mais recente do que o arquivo `main.o`, o comando que reconstrói esse arquivo será executado.

O mesmo raciocínio empregado acima para a dependência `main.o` do alvo principal aplica-se às demais dependências (i.e., `arq1.o` e `arq2.o`) deste alvo.

O que foi exposto até aqui sobre *make* e *makefile* é fundamental. Se você já entendeu como o utilitário *make* funciona, o que será apresentado a seguir apenas incrementa seu conhecimento com o objetivo de facilitar ainda mais a construção de programas multiarquivo utilizando *make* e *makefile*.

Um **comentário** num arquivo *makefile* é qualquer sequência de caracteres que segue o símbolo `#` e termina quando encerra a linha que o contém. Por exemplo:

```
# Isto é um comentário de um arquivo makefile
```

**Macros** (ou **variáveis**) facilitam a alteração de um arquivo *makefile* do mesmo modo que constantes simbólicas facilitam a alteração de programas escritos em C. Uma definição de macro num arquivo *makefile* tem o seguinte formato:

***nome-da-macro=valor***

Por exemplo:

```
COMPILADOR=gcc
```

Uma macro pode ser expandida no interior de uma regra ou na definição de outra macro utilizando a seguinte sintaxe:

***$(nome-da-macro)***

Por exemplo, considerando a definição da macro `COMPILADOR` acima, a regra a seguir:

```
arq1.o: arq1.c
	$(COMPILADOR) -c arq1.c -o arq1.o
```

após a expansão da macro `COMPILADOR`, tornar-se-ia:

```
arq1.o: arq1.c
	gcc -c arq1.c -o arq1.o
```

A seguir será apresentado um modelo simples de arquivo *makefile* que é suficiente para facilitar a construção de muitos arquivos executáveis baseados em programas multiarquivo. Se você sente necessidade de construir um arquivo *makefile* mais poderoso, consulte a documentação do programa *make* utilizado.

```
# Compilador utilizado
COMP = gcc

# Linker utilizado
LINKER = gcc

# Opções de compilação
OPCOES_COMP = -c -std=c99 -Wall -g

# Opções de ligação
OPCOES_LINK = -lm

# Arquivos-objeto
OBJETOS = arq1.o arq2.o main.o

# Nome do arquivo executável
EXEC = MeuProg.exe

$(EXEC): $(OBJETOS)
	$(LINKER) $(OPCOES_LINK) $(OBJETOS) -o $(EXEC)

arq1.o: arq1.c arq1.h
	$(COMP) $(OPCOES_COMP) arq1.c -o arq1.o
```

```
arq2.o: arq2.c arq2.h
	$(COMP) $(OPCOES_COMP) arq2.c -o arq2.o

main.o: main.c
	$(COMP) $(OPCOES_COMP) main.c -o main.o
```

# 2.7 Exemplos de Programação

## 2.7.1 Um Módulo para Leitura de Dados Resiliente

**Preâmbulo:** Programas interativos necessitam ler dados introduzidos pelo usuário e, por outro lado precisam ser robustos; i.e., capazes de responder adequadamente a qualquer tipo de entrada de dados. Ocorre que, apesar de a biblioteca padrão de C oferecer várias funções para leitura de dados, muitas dessas funções [p. ex., **scanf()**] não são equipadas para serem robustas.

**Problema:** Escreva um módulo, denominado `Leitura1`, que implementa as funções com protótipos e especificações apresentados na Tabela 2–4.

| Protótipo | Especificação |
|---|---|
| `int LeCaractere(void)` | *Lê um caractere* |
| `int LeInteiro(void)` | *Lê um número inteiro do tipo* **int** |
| `int LeNatural(void)` | *Lê um número inteiro do tipo* **int** *não negativo* |
| `int LeNaturalPositivo()` | *Lê um número inteiro do tipo* **int** *positivo* |
| `double LeReal(void)` | *Lê um número real do tipo* **double** |

**Tabela 2–4: Módulo para Leitura de Dados Robusta**

Além de realizarem as tarefas especificadas na Tabela 2–4, essas funções devem ser resilientes no sentido de permitirem que os usuários tenham um número ilimitado de oportunidades para introduzirem os valores esperados. Como requisito final, essas funções devem deixar limpo o buffer associado ao teclado após concluírem suas tarefas.

**Solução:**

*Arquivo de Cabeçalho*

O arquivo de cabeçalho do módulo em questão, denominado `leitura1.h`, contém alusões das funções globais disponibilizadas pelo módulo para uso em qualquer outro módulo. O conteúdo desse arquivo é o seguinte.

```
#ifndef _leitura1_H_
#define _leitura1_H_

extern int LeCaractere(void);
extern int LeInteiro(void);
extern int LeNatural(void);
extern int LeNaturalPositivo(void);
extern double LeReal(void);

#endif
```

*Arquivo de Implementação: Funções Locais*

As funções locais auxiliam a implementação das funções globais que efetivamente fazem parte do módulo. Isso significa que as funções descritas aqui são locais ao arquivo de programa da biblioteca e não podem ser usadas

fora dele. Por exemplo, a função `LimpaBuffer()` é essencial na implementação de todas as funções da biblioteca em discussão, mas ela não é necessária em programas que utilizam as funções disponibilizadas por essa biblioteca. Por isso, decidiu-se não tornar global a função `LimpaBuffer()`. Todas as funções locais são qualificadas com **static** (v. **Seção 2.6.2**) em suas definições e é exatamente esse qualificador que as tornam funções locais.

```
static int LimpaBuffer(void)
{
   int carLido, /* Armazena cada caractere lido */
       nCarLidos = 0; /* Conta o número de caracteres lidos */

      /* Lê e descarta cada caractere lido até    */
      /* encontrar '\n' ou getchar() retornar EOF */
   do {
      carLido = getchar(); /* Lê um caractere */
      ++nCarLidos; /* Mais um caractere foi lido */
   } while ((carLido != '\n') && (carLido != EOF));

      /* O último caractere lido foi '\n' ou  */
      /* EOF e não deve ser considerado sobra */
   return nCarLidos - 1;
}
```

A função `LimpaBuffer()` foi discutida na **Seção 2.2** e não requer discussão adicional.

*Arquivo de Implementação: Funções Globais*

A função `LeCaractere()` lê um caractere via teclado e remove os caracteres remanescentes no buffer. Ela retorna `O caractere lido e` deixa o buffer limpo.

```
int LeCaractere(void)
{
   int c, /* Armazenará cada caractere lido */
       nResto = 0; /* Número de caracteres excedentes */

   /* Volta para cá se ocorrer erro de leitura */
inicio:
   c = getchar(); /* Lê o caractere digitado */

      /* Verifica se getchar() retornou EOF */
   if (c == EOF) {
      printf( "\a\n\t>>>Erro de leitura. Tente novamente\n\t> " );
      goto inicio; /* Não faz mal algum */
   }

   nResto = LimpaBuffer(); /* Pelo menos '\n' se encontra no buffer */

      /* Repreende o usuário se ele digitou demais */
   if (nResto == 1)
      printf("\t>>> Um caractere foi descartado\n");
   else if (nResto > 1)
      printf( "\t>>> %d caracteres foram descartados\n", nResto );

   return c; /* Retorna o caractere lido */
}
```

A função `LeInteiro()` lê um número inteiro via teclado e deixa o buffer intacto. Essa função já foi apresentada na **Seção 2.2**.

A função `LeNatural()` lê um número natural via teclado e deixa o buffer intacto. Ela retorna o número natural lido.

```
int LeNatural(void)
{
   int num, /* O número lido */
       teste, /* Valor retornado por scanf() */
       nResto = 0; /* Número de caracteres excedentes */

   /* Desvia para cá se o valor for inválido */
inicio:
   teste = scanf("%d", &num); /* Tenta ler um valor válido */

      /* Se não ocorreu erro de leitura ou de final de arquivo, há */
      /* caracteres remanescentes que precisam ser removidos       */
   if (teste != EOF)
      nResto = LimpaBuffer();

      /* Enquanto o valor retornado por scanf() não indicar que um */
      /* valor válido foi lido continua tentando obter esse valor  */
   while(teste != 1 || num < 0) {
      printf("\a\n\t>>> O valor digitado e' invalido. Tente novamente\n\t> " );
      goto inicio; /* Não causa dano algum */
   }

      /* Repreende o usuário se ele digitou demais */
   if (nResto == 1)
      printf("\t>>> Um caractere foi descartado\n");
   else if (nResto > 1)
      printf("\t>>> %d caracteres foram descartados\n", nResto);

      /* O valor retornado certamente é válido */
   return num;
}
```

A função `LeNaturalPositivo()` lê um número natural positivo via teclado e deixa o buffer intacto. Ela retorna o número natural lido.

```
int LeNaturalPositivo(void)
{
   int num, /* O número lido */
       teste, /* Valor retornado por scanf() */
       nResto = 0; /* Número de caracteres excedentes */

   /* Desvia para cá se o valor for inválido */
inicio:
      /* Tenta ler um valor válido */
   teste = scanf("%d", &num);

      /* Se não ocorreu erro de leitura ou de final de arquivo, */
      /* há caracteres remanescentes que precisam ser removidos */
   if (teste != EOF)
      nResto = LimpaBuffer();

      /* Enquanto o valor retornado por scanf() não indicar que um */
      /* valor válido foi lido continua tentando obter esse valor  */
   while(teste != 1 || num < 1) {
      printf("\a\n\t>>> O valor digitado e' invalido. Tente novamente\n\t> " );
      goto inicio; /* Não causa dano algum */
   }

      /* Repreende o usuário se ele digitou demais */
   if (nResto == 1)
      printf("\t>>> Um caractere foi descartado\n");
```

```
   else if (nResto > 1)
      printf("\t>>> %d caracteres foram descartados\n", nResto);

      /* O valor retornado certamente é válido */
   return num;
}
```

A função `LeReal()` lê um valor do tipo **double** via teclado deixando o buffer associado ao teclado sem caracteres remanescentes. Ela retorna o valor lido e validado.

```
double LeReal(void)
{
   int    teste, /* Armazena o retorno de scanf() */
          nResto = 0; /* Número de caracteres  excedentes */
   double valorLido; /* O valor digitado pelo usuário */

      /* O laço encerra quando o usuário se comporta bem */
   while (1) {
         /* Tenta ler um valor do tipo double */
      teste = scanf("%lf", &valorLido);

         /* Se não ocorreu erro de leitura ou de final de arquivo, */
         /* há caracteres remanescentes que precisam ser removidos */
      if (teste != EOF)
         nResto = LimpaBuffer();

         /* Se o valor retornado por scanf() foi 1, a leitura foi OK */
      if (teste == 1)
         break; /* Leitura foi nota 10 */
      else /* Usuário foi mal comportado */
         printf("\a\n\t>>> O valor digitado e' invalido. Tente novamente\n\t> ");
   }

      /* Repreende o usuário se ele digitou demais */
   if (nResto == 1)
      printf("\t>>> Um caractere foi descartado\n");
   else if (nResto > 1)
      printf("\t>>> %d caracteres foram descartados\n", nResto);

      /* Retorna um valor válido do tipo double */
   return valorLido;
}
```

### Observações:

- A função `LeCaractere()` lê um caractere via teclado e, após a leitura, remove os caracteres remanescentes no buffer. Essa função também alerta o usuário caso ele tenha digitado caracteres além do esperado. A função `LeCaractere()` lê um caractere em **stdin** usando **getchar()**, limpa o buffer associado a esse meio de entrada por meio de uma chamada de `LimpaBuffer()` e, finalmente, informa se algum caractere foi descartado (porque o usuário digitou caracteres além do esperado).
- A função `LeInteiro()` lê um valor do tipo **int** em base decimal via teclado e, após a leitura, deixa o buffer associado a esse meio de entrada limpo. Se você entendeu bem o material apresentado na **Seção 2.2**, não terá dificuldade para entender a implementação da função `LeInteiro()`.
- A função `LeReal()` lê um valor do tipo **double** via teclado deixando o buffer associado ao teclado zerado. Apesar das aparências, a função `LeReal()` é semelhante a `LeInteiro()`, mas a implementação de `LeReal()` não usa **goto** para agradar os rigoristas.
- As funções `LeNatural()` e `LeNaturalPositivo()` são semelhantes à função `LeInteiro()` e não merecem maiores comentários.

### 2.7.2 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.

**Problema:** 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.

**Solução:** 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 (representada abaixo pela constante simbólica `DELTA`) que se deseja obter para um programa. A função `ComparaDoubles()` apresentada a seguir implementa o que foi exposto. Essa função recebe `d1` e `d2` como parâmetro e retorna: `0`, se os números forem considerados iguais, `-1`, se `d1` for considerado menor do que `d2` ou `1`, se `d1` for considerado maior do que `d2`.

```
int ComparaDoubles(double d1, double d2)
{
   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 */
}
```

**Observações:**

- 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). 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.

### 2.7.3 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:**

```
#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;

      /* 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 */

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

### 2.7.4 Raiz Quadrada Usando o Método de Newton e Raphson

**Problema:** A raiz quadrada de um número real pode ser calculada pelo método iterativo de Newton e Raphson:

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

em que $x_0$ é uma estimativa inicial que pode ser calculada como `x/2`, sendo `x` o número cuja raiz se deseja calcular.

**Problema:** Escreva uma função que calcula a raiz quadrada de um número real usando o método de Newton e Raphson.

**Solução:** A função `SqrtNewton()` a seguir calcula raiz quadrada usando o método de Newton e Raphson. Ela retorna a raiz quadrada do seu parâmetro, se ele não for negativo; caso contrário, essa função retorna `-1.0`.

```
double SqrtNewton(double x)
{
   double raiz;
      /* A raiz quadrada de zero é zero */
   if (x == 0.0)
      return 0.0;

      /* Se o parâmetro for negativo apresenta uma mensagem correspondente */
      /* e retorna um valor indicando o erro na chamada da função          */
   if (x < 0) {
      printf("\nChamada incorreta da funcao\n");
      return -1.0;
   }
      /* A estimativa inicial da raiz é o próprio */
      /* valor cuja raiz se deseja calcular       */
   raiz = x;
      /* Enquanto o quadrado da estimativa não for suficientemente */
      /* próximo de x, continua calculando estimativas             */
   while (ComparaDoubles(x, raiz*raiz))
      raiz = (raiz + x/raiz)/2; /* Faz nova estimativa */

   return raiz;
}
```

A função `SqrtNewton()` chama a função `ComparaDoubles()` definida na **Seção 2.7.2** para verificar se a raiz quadrada calculada é suficientemente próxima do verdadeiro valor. Entretanto, o valor da constante `DELTA` usada por essa última função não pode ser tão pequeno quanto aquele usado no programa da **Seção 2.7.2**; caso contrário, corre-se o risco de não haver convergência do método. No programa usado para testar a função `SqrtNewton()` o valor dessa constante foi `1.0E-10`.

# 2.8 Exercícios de Revisão

### Funções (Seção 2.1)

1. (a) O que é uma função? (b) Em que situações o uso de funções, além de **main()**, é recomendável num programa? (c) Como o uso de funções facilita a escrita de programas? (d) Cite algumas vantagens decorrentes do uso de funções num programa em C. (e) Descreva as partes que constituem uma definição de função?
2. (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?
3. Na linguagem C, uma função pode ser definida no corpo de outra função?
4. Explique o uso de **void** nos seguintes cabeçalhos de funções:
   (a) `void F(int x)`
   (b) `int G(void)`
5. O que há de errado com o seguinte cabeçalho de função?

   ```
   void F(int x, y)
   ```
6. Qual é o propósito de uma instrução **return**?
7. (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?
8. 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.
9. Uma função pode ter mais de uma instrução **return**? Explique.

**10.** 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;
}
```

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

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

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

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

**13.** 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.

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

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

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

**17.** O que é modo de um parâmetro?

**18.** (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?

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

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

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

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

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

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

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

**26.** Qual é o significado de `(void)` precedendo uma chamada de função, como no exemplo abaixo?

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

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

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

**29.** Explique por que, mesmo quando não são estritamente necessárias, alusões facilitam o trabalho do programador.

**30.** (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?

**31.** 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.

**32.** 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?

### Leitura de Dados via Teclado 2: Robusta (Seção 2.2)

**33.** (a) Considerando o trecho de programa a seguir, o que a função **scanf()** lê se o usuário digitar `123z`? (b) Que consequência danosa ao programa essa entrada de dados pode infligir?

```
int x;
...
printf("\nDigite um valor inteiro: ");
scanf("%d", &x);
```

**34.** (a) Considerando o mesmo trecho de programa da questão anterior, o que a função **scanf()** lerá se o usuário digitar `z123`? (b) Qual será o conteúdo do buffer associado ao teclado logo após o usuário ter concluído sua digitação?

**35.** Considerando o mesmo trecho de programa das duas últimas questões, qual será o conteúdo do buffer associado ao teclado após a atuação de **scanf()** quando o usuário digitar `123`?

**36.** (a) Em que situação a função **scanf()** lê o caractere `'\n'` encontrado no buffer associado ao teclado? (b) Quando ela o remove do buffer? (c) Quando ela não o remove do buffer?

**37.** (a) Que caracteres remanescentes no buffer associado ao teclado não afetam uma leitura de valor numérico efetuada por **scanf()**? (b) Em que situação qualquer caractere remanescente no buffer afeta uma leitura efetuada por **scanf()**?

**38.** Descreva o funcionamento do buffer associado à entrada de dados padrão.

**39.** Para que serve a função `LimpaBuffer()` apresentada na **Seção 2.2**?

**40.** Como laços de repetição são usados em leitura de dados via teclado?

### Duração de Variáveis (Seção 2.3)

**41.** 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?

**42.** (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?

**43.** (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?

**44.** (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?

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

**46.** (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?

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

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

**48.** (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?

### Escopo (Seção 2.4)

**49.** (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?

**50.** 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`?

**51.** (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?

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

**53.** (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?

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

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

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

### Diretivas de Pré-processamento (Seção 2.5)

**57.** (a) O que é uma macro? (b) Descreva o uso de parâmetros em macros. (c) Quais são as principais diferenças entre macros e funções em C?

**58.** Quais são os cuidados que devem ser tomados quando se utilizam macros?

**59.** Descreva (a) vantagens e (b) desvantagens do uso de macros em relação ao uso de funções.

**60.** (a) Por que é recomendável envolver parâmetros de macros entre parênteses? (b) Por que é recomendável envolver expressões contendo parâmetros de macros entre parênteses?

**61.** O que é uma macro predefinida?

**62.** Para que serve uma macro vazia?

**63.** Descreva o operador de pré-processamento **#**.

**64.** (a) O que é compilação condicional? (b) Quais são as diretivas de pré-processador utilizadas em compilação condicional?

**65.** Qual é a diferença entre as diretivas **#if** e **#ifdef**?

**66.** Qual é a diferença entre compilação condicional e execução condicional?

**67.** Por que deve-se evitar inclusão múltipla de arquivos de cabeçalho?

**68.** Descreva a estratégia utilizada para prevenir a inclusão múltipla de arquivos de cabeçalho.

**69.** Em Matemática, define-se o valor absoluto de um número real como:

$$|x| = \sqrt{x^2}$$

Assim parece natural concluir que a seguinte macro é adequada para calcular o valor absoluto de um número de ponto flutuante:

```
ABS(x) (sqrt((x)*(x)))
```

em que **sqrt()** é a função que calcula raiz quadrada declarada no cabeçalho `<math.h>` da biblioteca padrão de C. Entretanto, essa macro apresenta um sério problema. Qual é esse problema?

### Programas Multiarquivo (Seção 2.6)

**70.** Por que o uso de **extern** numa alusão de variável global, apesar de ser opcional, é recomendável?

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

**72.** Quais são as vantagens obtidas com a divisão de um programa em módulos?

**73.** (a) O que é um arquivo de cabeçalho? (b) Qual é o conteúdo provável de um arquivo de cabeçalho? (c) O que é um arquivo de programa? (d) Qual é o conteúdo provável de um arquivo de programa?

**74.** (a) Um arquivo de cabeçalho pode incluir outro arquivo de cabeçalho? (b) Um arquivo de cabeçalho pode incluir um arquivo de programa? (c) Um arquivo de programa pode incluir outro arquivo de programa? Justifique suas respostas.

**75.** Qual é o significado de *projeto* num ambiente IDE?

**76.** Comente a seguinte afirmação: *compilar um programa não significa necessariamente construir um programa executável*.

**77.** Qual é o significado comumente atribuído aos comandos *Make* e *Build* num ambiente IDE?

**78.** (a) O que é make? (b) O que é um arquivo makefile?

**79.** Suponha que você tenha um programa em C constituído pelos seguintes módulos e respectivos arquivos:

| Módulo | Arquivos |
|---|---|
| Principal | `main.c` |
| Entrada | `Entrada.h, Entrada.c` |
| Saída | `Saida.h, Saida.c` |
| Processamento | `Process.h, Process.c` |

(a) Suponha ainda que o nome desejado para o programa executável seja `prog1.exe`. Como você criaria este programa executável utilizando o compilador GCC?

(b) Agora suponha que este programa precisa ser ligado a uma biblioteca denominada `Lib1.a` armazenada no diretório `/home/bibs`, caso você esteja utilizando Unix/Linux, ou `C:\Bibs`, caso você esteja utilizando Windows. Como você instruiria o *linker* a fazer a ligação entre seu programa e esta biblioteca?

(c) Escreva um arquivo *makefile* para automatizar o processo de criação deste programa.

**80.** Suponha que você tenha um programa multiarquivo escrito em C cujos arquivos incluem outros arquivos conforme mostrado na tabela a seguir:

| Arquivo | Arquivos incluídos |
|---|---|
| `main.c` | `stdio.h`, `Process1.h`, `Process2.h`, `Entrada.h`, `Saida.h` |
| `Entrada.c` | `stdio.h`, `Entrada.h` |
| `Saida.c` | `stdio.h`, `Saida.h` |
| `Process1.c` | `stdlib.h`, `Process1.h` |
| `Process2.c` | `math.h`, `Process1.h`, `Process2.h` |

(a) Que arquivos teriam que ser recompilados se fosse efetuada uma alteração apenas no arquivo:
(i) `Entrada.h`
(ii) `Saida.c`
(iii) `Process2.c`

(b) Escreva um arquivo *makefile* que facilite a construção de um executável contendo os arquivos do programa descrito acima, supondo que o nome deste executável seja `processos`.

### Exemplos de Programação (Seção 2.7)

**81.** O que é um programa interativo robusto do ponto de vista de entrada de dados?

**82.** Por que operadores relacionais não são adequados para comparar números reais?

**83.** Descreva o funcionamento da função `ComparaDoubles()`.

## 2.9 Exercícios de Programação

**EP2.1** 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:

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

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

**EP2.2** **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`.

**EP2.3** (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)!)*.]

**EP2.4** **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.

**EP2.5** **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.

**EP2.6** (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).

**EP2.7** **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.

**EP2.8** **Preâmbulo:** A função `ComparaDoubles()`, definida na **Seção 2.7.2**, 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 que teste a função especificada no item (a).

**EP2.9** (a) Escreva uma função que determina o número de dígitos de um número inteiro não negativo recebido como parâmetro.

**EP2.10** (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.

**EP2.11**

(a) Escreva um módulo, denominado **`Geometria`** (arquivos **`Geometria.h`** e **`Geometria.c`**), contendo funções que calculem o seguinte:

(i) A área de um círculo

**Protótipo:** `double AreaDeCirculo(double raio)`

(ii) O volume de um cilindro

**Protótipo:** `double VolumeDeCilindro(double raio, double altura)`

(iii) O volume de um conE

**Protótipo:** `double VolumeDeCone(double raio, double altura)`

(iv) A área de um retângulo

**Protótipo:** `double AreaDeRetangulo(double lado1, double lado2)`

(b) Escreva um módulo, denominado Interface (arquivos **`Interface.h`** e **`Interface.c`**), responsável por uma interação com o usuário dirigida por menu contendo funções que realizem o seguinte:

(i) Apresente o programa para o usuário (i.e., apresente o propósito do programa, como utilizá-lo, etc.)

**Protótipo:** `void Apresentacao(void)`

(ii) Apresente o menu principal do programa para o usuário. Este menu deve ter o seguinte formato:

```
Escolha uma das opções a seguir:
Área de um círculo.............1
Volume de um cilindro..........2
Volume de um cone..............3
Área de um retângulo...........4
Sair do programa...............5
Opção:
```

**Protótipo:** `void ApresentaMenu(int nItens, int menorOpcao, ...)`

(iii) Leia, valide e retorne a opção escolhida pelo usuário.

**Protótipo:** `int LeOpcao(int menorValor, int maiorValor)`

(iv) Solicite ao usuário os dados correspondentes à opção 1.

**Protótipo:** `void DadosDeCirculo(double* raio)`

(v) Solicite ao usuário os dados correspondentes à opção 2.

**Protótipo:** `void DadosDeCilindro(double *raio, double *altura)`

(vi) Solicite ao usuário os dados correspondentes à opção 3.

**Protótipo:** `void DadosDeCone(double * raio, double * altura)`

(vii) Solicite ao usuário os dados correspondentes à opção 4.

**Protótipo:** `void DadosDeRetangulo(double *lado1, double *lado2)`

[**NB:** As funções dos itens de (iv) a (viii) devem validar as entradas do usuário antes de retornarem.)

(c) Escreva um arquivo denominado **`main.c`**, contendo (apenas) uma função **main()** que realize o seguinte:

(i) Faça uma apresentação do programa para o usuário

(ii) Faça repetidamente (i.e., até que o usuário escolha sair do programa) o seguinte:

[1] Apresente o menu de opções para o usuário
[2] Leia a opção escolhida pelo usuário
[3] Solicite ao usuário informações complementares para a realização do cálculo desejado
[4] Efetue o respectivo cálculo
[5] Apresente o resultado numa forma inteligível para o usuário
[6] Despeça-se graciosamente do usuário

**Sugestão:** Utilize a função `LeReal()` apresentada na **Seção 2.7.1** para ler e validar os valores introduzidos pelo usuário.

**Observações Complementares:**

(1) As funções de seu programa devem respeitar os protótipos sugeridos aqui.
(2) Você pode utilizar outras funções auxiliares na implementação de cada módulo, mas, neste caso, estas funções não devem ser exportadas para outros módulos. Isto é, estas funções auxiliares devem ser declaradas com **static**.
(3) Seu programa não deve conter nenhuma variável global.

**EP2.12** Desenvolva um projeto semelhante àquele do exercício **EP2.11**, mas desta vez, substitua o módulo `Geometria` por um arquivo de cabeçalho (denominado `GeometriaM.h`) que substitui as funções providas por aquele módulo por macros correspondentes. Por exemplo, a função `AreaDeCirculo()` seria substituída pela macro:

```
#define AREA_DE_CIRCULO(raio) (PI*(raio)*(raio))
```

onde `PI` é uma constante simbólica definida no início do próprio arquivo `GeometriaM.h`.

**Sugestões:**

(1) Utilize a notação sugerida para macros na **Seção 2.5.1**.
(2) O módulo `Interface` do exercício **EP2.12** não precisa ser modificado.
(3) O arquivo que contém a função **main()**, que você poderá denominar `main2.c`, precisa ser modificado apenas substituindo as chamadas de funções por chamadas das macros correspondentes.
(4) Seu projeto terá agora apenas dois arquivos de programa: o arquivo `Interface.c` e o arquivo `main2.c`.

**EP2.13** (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.

**EP2.14** 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.