CAPÍTULO 2

PROGRAMAS MONOARQUIVOS

2.1 Introdução

Este capítulo dedica-se a ensinar como construir programas simples usando a linguagem C. Estes programas são denominados monoarquivos devido ao fato de consistirem de um único arquivo-fonte. No **Capítulo 4**, programas multiarquivos serão explorados.

Aqui também faz-se uma introdução à biblioteca padrão de C, em geral, e ao módulo de entrada e saída stdio, em particular. A descrição é intencionalmente imprecisa, pois o objetivo aqui é transmitir o mínimo de conhecimento necessário para que o iniciante comece a construir programas interativos simples em C.

2.2 Introdução à Biblioteca Padrão de C

Parte da funcionalidade da linguagem C está contida em **funções de biblioteca** que residem fora da linguagem em si. A **biblioteca padrão** de C é dividida em grupos de funções que têm alguma afinidade entre si. Por exemplo, existem grupos de funções para entrada e saída, gerenciamento de memória, operações matemáticas etc. Cada grupo de funções de biblioteca, denominado **módulo**, possui dois arquivos associados:

(1) Um arquivo objeto que contém as implementações das funções do módulo previamente compiladas.

(2) Um arquivo-fonte, denominado **cabeçalho**, que contém *definições*

parciais (**alusões**) legíveis das funções implementadas no arquivo objeto. Um arquivo de cabeçalho pode conter ainda outras componentes, tais como definições de tipos e constantes. Arquivos de cabeçalho normalmente têm a extensão `.h`[1].

Para utilizar alguma função de um módulo de biblioteca num programa, você deve incluir o arquivo de cabeçalho do módulo usando uma diretiva #**include** (usualmente, no início do programa) com um dos seguintes formatos:

#**include** <*nome-do-arquivo*>

ou

#**include** "*nome-do-arquivo*"

A diferença entre os dois formatos é a forma como o compilador (ou, mais precisamente, o pré-processador) procura o arquivo a ser incluído. Os locais onde os arquivos são procurados dependem do compilador bem como do sistema operacional utilizados. O segundo formato é mais geral do que o primeiro no sentido de que a busca é mais extensa, mas, em compensação, ela pode ser mais demorada. No caso de inclusão de arquivos de biblioteca, se a instalação do compilador tiver sido feita de acordo com as recomendações do fabricante, não haverá problema de localização do arquivo com o primeiro formato acima. Se não conseguir incluir um arquivo com um dos formatos acima, consulte a documentação do compilador para saber em que diretórios estes arquivos podem ser localizados.

2.3 Funções Elementares de Entrada e Saída

2.3.1 O Módulo stdio

As funções responsáveis pelas operações básicas de entrada e saída em C fazem parte de um módulo da biblioteca padrão de C denominado

[1] "h" vem de ***header***; i.e., cabeçalho em inglês.

stdio[2]. Para utilizar as funções de biblioteca de entrada/saída você deve, portanto, incluir o cabeçalho que contém as alusões destas funções, que é o arquivo `stdio.h`, por meio da diretiva:

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

Quando encontra uma diretiva #**include**, o pré-processador[3] C a substitui pelo conteúdo do arquivo que se deseja incluir. Por exemplo, quando a diretiva acima é processada, o conteúdo do arquivo `stdio.h` é inserido no local do programa onde ela se encontra. Desta maneira, quando encontrar uma chamada de uma função contida no arquivo `<stdio.h>`, o compilador saberá que se trata realmente de uma função. Se você não incluir o arquivo de cabeçalho para as funções que deseja utilizar em seu programa, qualquer chamada de tal função será tratada pelo compilador, como, por exemplo, *identificador não declarado*, visto que nem elas fazem parte da linguagem C em si nem você as declarou explicitamente.

Note que, como foi afirmado antes, um arquivo de cabeçalho contém apenas declarações parciais de funções; as definições completas estão contidas no arquivo objeto correspondente. Em resumo, as definições parciais de funções num arquivo de cabeçalho incluído servem apenas como **alusões** que indicam ao compilador que aquelas funções são definidas em outro arquivo. Para que o programa seja executado, é necessário que ele seja ligado com o código binário das funções de biblioteca utilizadas. Esta ligação é feita por um **editor de ligações** (*link editor* ou *linker*, em inglês).

A seguir serão apresentadas funções básicas de entrada e saída que possibilitarão, ao final deste capítulo, que você possa escrever programas interativos elementares em C.

---

[2] Este nome vem de ***standard input/output***; i.e., entrada/saída padrão.

[3] Por enquanto, não se preocupe em saber exatamente o que é o pré-processador de C; considere-o como um programa que prepara previamente um arquivo-fonte a ser em seguida processado pelo compilador. As instruções processadas pelo pré-processador são denominadas diretivas (v. **Capítulo 5**).

2.3.2 Função getchar()

A função **getchar()** permite a entrada de um único caractere utilizando o meio de entrada padrão (usualmente, o teclado do computador). A forma ***normal*** de uso desta função é[4]:

variável-do-tipo-unsigned-char = **getchar()**;

Quando a função **getchar()** é chamada e não encontra um caractere no meio de entrada, a execução é interrompida até que o usuário digite um caractere seguido de `[ENTER]` (ou `[RETURN]`). Então, a função **getchar()** retorna o inteiro correspondente ao caractere digitado (no intervalo entre 0 e 255). Por exemplo, se um programa contém o fragmento a seguir:

```
unsigned char meuCaractere;

meuCaractere = getchar();
```

e o usuário digita o caractere `A` seguido de `[ENTER]`, à variável `meuCaractere` será atribuído o valor inteiro correspondente a este caractere no código de caracteres utilizado (por exemplo, se o código ASCII for utilizado, a variável receberá o valor `65`, que é o valor inteiro correspondente a `A` no código ASCII).

Note que a chamada de uma função sem parâmetros, como **getchar()**, requer que o nome da função seja seguido de abre e fecha-parênteses.

2.3.3 Função putchar()

A função **putchar()** recebe como entrada um parâmetro do tipo **int** e imprime o caractere correspondente a este valor no meio de saída padrão.

---

[4] Na realidade, o tipo de retorno de **getchar()** é **int**, mas você só precisa atribuir este valor retornado a uma variável do tipo **int** se estiver esperando que a função retorne, além de caracteres, a constante **EOF**, que indica o final de um arquivo, pois este valor não cabe numa variável do tipo **char**. No Unix, `[CTRL-D]` simula final de arquivo na entrada de dados padrão, enquanto no DOS `[CTRL-Z]` tem o mesmo efeito.

A função **putchar()** retorna um inteiro que corresponde ao caractere impresso se a impressão for efetuada com êxito. Entretanto, este valor de retorno raramente é utilizado, de modo que usualmente a função **putchar()** é utilizada simplesmente como:

putchar(*expressão-do-tipo-unsigned-char*);

Por exemplo:

```
putchar('A');
```

resultaria na impressão de `A` no meio de saída padrão. A função retornaria o valor inteiro correspondente a este caractere no código de caracteres utilizado, mas este valor não é utilizado. Outra forma equivalente de imprimir o caractere `A` no meio de saída padrão seria por meio da instrução:

```
putchar(65); // Não recomendável!
```

Entretanto, esse modo de chamada não é recomendável, pois assume o uso de um código de caracteres no qual o inteiro correspondente ao caractere `A` é `65`. Como o padrão da linguagem C não especifica o uso de nenhum código de caracteres em particular, esta última chamada de **putchar()** não é portável (i.e., seu funcionamento depende de implementação).

2.3.4 Função scanf()

A função **scanf()** é uma função de entrada mais geral do que **getchar()**, pois permite a entrada de valores de vários tipos simultaneamente. A função **scanf()** permite um número qualquer de argumentos. O primeiro argumento, que é obrigatório, é uma cadeia de caracteres denominada de **string de formatação**. Os argumentos seguintes são ***endereços*** de variáveis que irão conter os valores lidos no meio de entrada. Mais precisamente, estes últimos parâmetros representam endereços onde os dados lidos serão armazenados. Comumente, uma chamada da função **scanf()** tem o seguinte formato:

scanf(*string-de-formatação*, *endereço-de-variável1*, ..., *endereço-de variávelN*);

O ***string*** de formatação especifica justamente o formato dos dados a serem lidos e que serão atribuídos aos argumentos seguintes. Este ***string*** pode assumir formas bem variadas, mas, aqui, serão vistas apenas algumas formas mais comuns[5]. O ***string*** de formatação pode conter texto, mas, normalmente, contém apenas **especificadores de formato**, que em sua forma mais simples são constituídos do caractere **%** seguido de um ou dois caracteres que informam como o dado correspondente a ser lido será interpretado. A **Tabela 18**, apresentada a seguir, enumera os especificadores de formato mais comuns utilizados pela função **scanf()**.

ESPECIFICADOR DE FORMATO	INTERPRETAÇÃO DO VALOR LIDO
%c	Caractere
%s	Cadeia de caracteres (***string***)
%d ou **%i**	Inteiro em base decimal
%u	Inteiro sem sinal (**unsigned**)
%f, **%lf** e **%Lf**	Número de ponto-flutuante (**float**, **double** e **long double**, respectivamente)
%e, **%le** e **%Le**	Número de ponto-flutuante em notação científica (**float**, **double** e **long double**, respectivamente)

Tabela 18: Especificadores de formato comuns utilizados por scanf()

O uso correto da função **scanf()** requer que haja um endereço de variável para cada especificador de formato no ***string*** de formatação e que o tipo de cada variável seja compatível com a especificação de formato correspondente. Por exemplo, a seguinte chamada da função **scanf()** espera ler três valores no meio de entrada padrão:

[5] O **Apêndice B** apresenta uma descrição resumida desses especificadores de formato, enquanto o **Volume II** os explora em profundidade.

```
long            inteiroLongo;
float           numeroDePontoFlutuante;
unsigned  char  meuCaractere;

scanf("%ld %f %c", &inteiroLongo, &numeroDePontoFlutuante,
      &meuCaractere);
```

Observe que os endereços das variáveis requeridos para os parâmetros finais da função **scanf()** são obtidos precedendo-se o nome da variável com o caractere **&**. Este caractere, quando aplicado antes de uma variável, representa o operador unário de endereço e deve ser lido como ***o endereço de***. Cuidado para não esquecer de preceder cada variável na lista de entrada com o operador de endereço, pois a função **scanf()** não irá indicar diretamente o erro: simplesmente, ela não irá atribuir o valor esperado à variável.

A função **scanf()** retorna um valor inteiro que, muitas vezes, não é utilizado como deveria. Este valor retornado indica o número de variáveis que tiveram valores atribuídos e deve ser utilizado para checar a ocorrência de algum problema na entrada de dados (v. **Seção 2.6**).

Quando caracteres que não são parte de especificadores de formato são incluídos no ***string*** de formatação, a função **scanf()** espera que o usuário digite cada caractere exatamente como ele aparece no ***string*** de formatação. Você deve tomar cuidado com esta característica da função **scanf()**, pois pode ser que isto não seja aquilo que você deseja. Por exemplo, se você incluir em seu programa a instrução:

```
scanf("Digite um numero  inteiro: %d", &meuInteiro);
```

a função **scanf()** espera que o usuário digite exatamente a cadeia de caracteres: `Digite um numero inteiro:` antes de introduzir o número inteiro esperado. Portanto, não esqueça que a função **scanf()** é uma função ***apenas de entrada***. Se você deseja informar ao usuário que seu programa está esperando uma entrada de dados (o que, aliás, é recomendável) preceda a instrução de entrada com uma instrução de saída que solicite a entrada do usuário. A função **printf()**, a ser apresentada em seguida, pode ser utilizada para este propósito.

2.3.5 Função printf()

A função de saída **printf()** tem sintaxe similar àquela da função **scanf()**. A semelhança, entretanto, resume-se ao fato de ambas permitirem múltiplos argumentos, sendo que o primeiro, obrigatório, é um ***string*** de formatação. Os argumentos finais são variáveis, constantes ou expressões, cujos valores serão escritos no meio de saída padrão. Além de especificadores de formato, o ***string*** de formatação pode conter caracteres que são apresentados no meio de saída. Por exemplo, ao final da execução do trecho de programa a seguir:

```
int   n = 3;
printf("O quadrado de %d e' %d.", n, n*n);
```

seria impresso o seguinte no meio de saída:

```
O quadrad1o de 3 e' 9.
```

Os especificadores de formato utilizados pela função **printf()** nem sempre coincidem com os correspondentes utilizados pela função **scanf()**. Por exemplo, o especificador `%f` é utilizado na leitura de valores do tipo **float** por **scanf()**, mas utilizado na escrita de valores do tipo **double** por **printf()**. A **Tabela 19** enumera os especificadores de formato mais comuns utilizados pela função **printf()**.

ESPECIFICADOR DE FORMATO	O VALOR SERÁ IMPRESSO COMO...
%c	Caractere
%s	Cadeia de caracteres (string)
%d ou %i	Inteiro (int) em base decimal
%ld ou %li	Inteiro (long int) em base decimal
%u	Inteiro sem sinal (unsigned)
%f e %Lf	Número de ponto-flutuante (double e long double, respectivamente)
%e (ou %E) e %Le (ou %LE)	Número de ponto-flutuante em notação científica (double e long double, respectivamente)

Tabela 19: Especificadores de formato comuns utilizados por printf()

Especificadores de formato podem ser ainda mais específicos ao indicar como os argumentos finais de **printf()** devem ser impressos. Por exemplo, pode-se indicar que um número de ponto flutuante deve ser impresso com um total de cinco casas, sendo duas delas decimais, por meio do especificador `%5.2f` . Um estudo mais aprofundado de formatação de entrada/saída por meio de especificadores de formato encontra-se no **Volume II**.

2.4 Como Construir um Programa Simples em C

2.4.1 Construção de Programas Pequenos numa Linguagem Algorítmica

A construção de um programa *de pequeno porte* em qualquer linguagem algorítmica convencional segue, normalmente, a seqüência de tarefas descrita a seguir:

1. Escrita do algoritmo a ser seguido pelo programa numa linguagem algorítmica (pseudolinguagem). Este passo corresponde ao projeto de um programa pequeno.

2. Tradução do algoritmo na linguagem desejada. Se a linguagem algorítmica utilizada no passo 1 for próxima à linguagem de programação utilizada, este passo não oferece nenhuma dificuldade.

3. Edição do programa resultante do passo 2. A execução deste passo resulta num arquivo-fonte e pode ser considerada como parte do passo 2, isto é, pode-se usualmente traduzir e editar um programa num único passo, sem que se tenha que fazer esta tradução à mão, por exemplo.

4. Compilação e edição de ligações do programa. O processo de compilação resulta num arquivo objeto (i.e., em código de máquina), mas este arquivo não consiste necessariamente num programa executável. O arquivo objeto resultante da compilação precisa ser ligado a outros arquivos objetos que porventura contenham funções ou variáveis utilizadas pelo programa. Isto é realizado por meio de um **editor de ligações**

(*linker*). Muitos ambientes de programação possuem uma facilidade que processa compilação e edição de ligações num único comando (esta facilidade, tipicamente, é denominada *Make* ou *Build*). Alguns ambientes também oferecem uma facilidade com finalidade tripla: que compila, faz ligações e executa um programa num único comando (tipicamente, este comando é denominado *Run*). A **Figura 3** ilustra, de modo simplificado, o processo de transformação de um programa constituído de um único arquivo-fonte em programa executável.

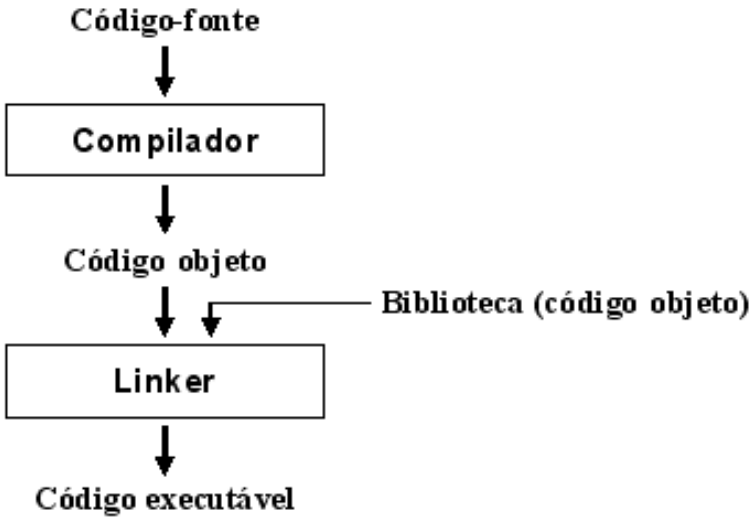


Figura 3: Processo simplificado de construção de um programa executável

5. Teste e depuração do programa. Neste passo, o programa deve ser executado para verificar-se se ele funciona da forma como seria esperado. Deve-se examinar o programa sob várias circunstâncias (**casos de entrada**) e verificar se, em cada uma delas, o programa funciona adequadamente. Se o programa apresenta um comportamento anormal, ele deve ser depurado (i.e., deve-se procurar as instruções responsáveis pelos erros e corrigi-las). Exceto para programas triviais, não se pode,

usualmente, provar que um programa é correto. Pode-se entretanto, provar que um programa é incorreto: é suficiente que ele não funcione numa dada circunstância.

Programas ***triviais***[6] não requerem os passos 1 e 2 mostrados anteriormente. Isto é, tais programas podem ser editados diretamente (passo 3) num editor de programas (v. **Seção 2.4.4**). Se você não consegue fazer isto, não se sinta frustrado: siga toda a seqüência de passos aqui citada, pois, com a prática adquirida, você conseguirá escrever cada vez mais programas sem ter que escrever seus algoritmos em linguagem algorítmica. Nunca esqueça, entretanto, que não se deve escrever programas mais complexos sem planejá-los previamente (passo 1 anterior). O tempo gasto no planejamento da solução é recuperado na depuração do programa. Freqüentemente, a depuração de um programa mal planejado leva muito mais tempo do que a escrita do programa em si.

2.4.2 Estrutura de um Programa Simples em C

Um programa simples, consistindo em um único arquivo-fonte, em C possui o seguinte formato geral:

```
#include <stdio.h>

/* Inclua aqui outros arquivos de cabeçalho usados no programa */

/*  Inclua aqui declarações de constantes e tipos */
/*  que serão necessários no seu programa         */

int main(void)
{

    /*  Inclua aqui definições de variáveis que */
    /*  serão necessárias no seu programa      */

    /*  Inclua aqui instruções que executem as ações    */
    /*  necessárias para funcionamento do seu programa. */
    /*  Isto é, traduza aqui, em C, seu algoritmo.     */
```

[6] Evidentemente, o conceito de ***trivial*** aqui é relativo; isto é, ele depende, dentre outros fatores, da experiência prática do programador.

```
    return 0;  /* Informa ao sistema operacional que */
               /* o programa terminou normalmente   */
}
```

A instrução:

```
return 0;
```

que aparece ao final da maioria dos programas escritos em C indica que o programa foi executado sem anormalidades. Quando o programador é capaz de prever a ocorrência de algum problema que impeça a execução normal do programa (talvez, por exemplo, porque o programa tenha recebido dados incorretos), ele deve usar um valor diferente de zero (tipicamente, 1). Alguns programas, quando executam outro programa, testam o valor retornado pelo programa chamado. Um exemplo de tal programa é o utilitário *make*, discutido na **Seção 4.11.4**.

Aliás, existe uma convenção tácita em programação em C que reza que zero denota uma operação bem-sucedida, enquanto um valor diferente de zero significa que uma operação não obteve êxito. Assim, a última instrução executada por um programa em C deve ser uma instrução **return** com o valor zero, quando o programa é bem-sucedido, ou diferente de zero quando ocorre algum problema durante sua execução.

2.4.3 Utilizando um Ambiente Integrado de Desenvolvimento

Um **ambiente integrado de desenvolvimento** (ou **IDE**) é um programa que coordena a execução de vários outros programas dedicados ao desenvolvimento de programas. Tipicamente, um IDE[7] está associado a pelo menos os seguintes programas de desenvolvimento:

- **Editor de programas**. Um editor de programas é semelhante a um editor de texto comum, mas possui a vantagem adicional de, pelo menos, conhecer a sintaxe de uma ou mais linguagens de programação. A vantagem de usar um editor de programas

[7] IDE é derivado de ***Integrated Development Environment***, em inglês.

em vez de um simples editor de texto em desenvolvimento é que um editor de programa não apenas é capaz de facilitar bastante a edição de programas como também de indicar possíveis erros de sintaxe antes mesmo de o programa ser compilado. Tipicamente, o editor de programas faz parte do próprio IDE. Qualquer IDE possui a capacidade de integrar este componente.

• **Compilador**. Evidentemente este programa é imprescindível, pois é o responsável pela tradução de programa-fonte em programa objeto. A tarefa do IDE é executar o compilador sem que seja necessário abandonar o ambiente de desenvolvimento. Qualquer IDE possui a capacidade de integrar este componente.

• **Editor de Ligações (*Linker*)**. Este programa também é imprescindível num IDE, pois é ele quem realmente cria os programas executáveis. A tarefa do IDE é permitir a execução do *linker* sem que seja necessário abandonar o ambiente de desenvolvimento. Qualquer IDE possui a capacidade de integrar este componente.

• **Carregador (*Loader*)**. Um IDE capaz de executar este programa permite que o programador execute seus próprios programas sem ter que abandonar o IDE. A maioria dos IDEs possui a capacidade de integrar este componente.

• **Depurador**. Este programa auxilia o programador na depuração de programas e não é imprescindível como os demais, mas é altamente desejável. Bons IDEs (por exemplo, IDEs Borland) são capazes de integrar um depurador, mas outros (por exemplo, DevC++) não o são.

O ambiente de desenvolvimento escolhido para demonstração aqui é DevC++ e as razões para esta escolha são as seguintes:

- Este IDE é facilmente encontrado na internet.
- Ele utiliza uma versão do compilador gcc, que apresenta razoável suporte para o padrão C99.
- Demanda poucos recursos computacionais.
- É gratuito.
- É muito fácil de usar.

Para quem utiliza Linux ou Unix, existem ambientes de desenvolvimento, como Anjuta e KDevelop, bem melhores do que o ambiente DevC++. Todavia, estes IDEs não são tão simples de ser utilizados quanto o ambiente DevC++ e são recomendados apenas quando o programador já apresenta um certo domínio na linguagem C. Assim, ele não tem que se ocupar em aprender duas coisas ao mesmo tempo: como programar e como usar devidamente o ambiente de desenvolvimento.

O primeiro passo para um bom acompanhamento das orientações apresentadas a seguir é obter o IDE DevC++. Esse passo você não terá nenhuma dificuldade em executar, pois esse programa é facilmente encontrado na internet e sua instalação não apresenta nenhuma dificuldade.

A breve introdução ao ambiente DevC++ apresentada a seguir assume o uso da versão 4.9.9.2 acompanhada do compilador MingGW versão 2.95.2-1, incluso neste ambiente. Após a instalação, você deverá ser capaz de encontrar e executar este programa no menu *Programas* do sistema operacional Windows.

▶ Configurando o Ambiente DevC++

Quando você abre o programa DevC++, obtém uma janela semelhante àquela mostrada na **Figura 4**.

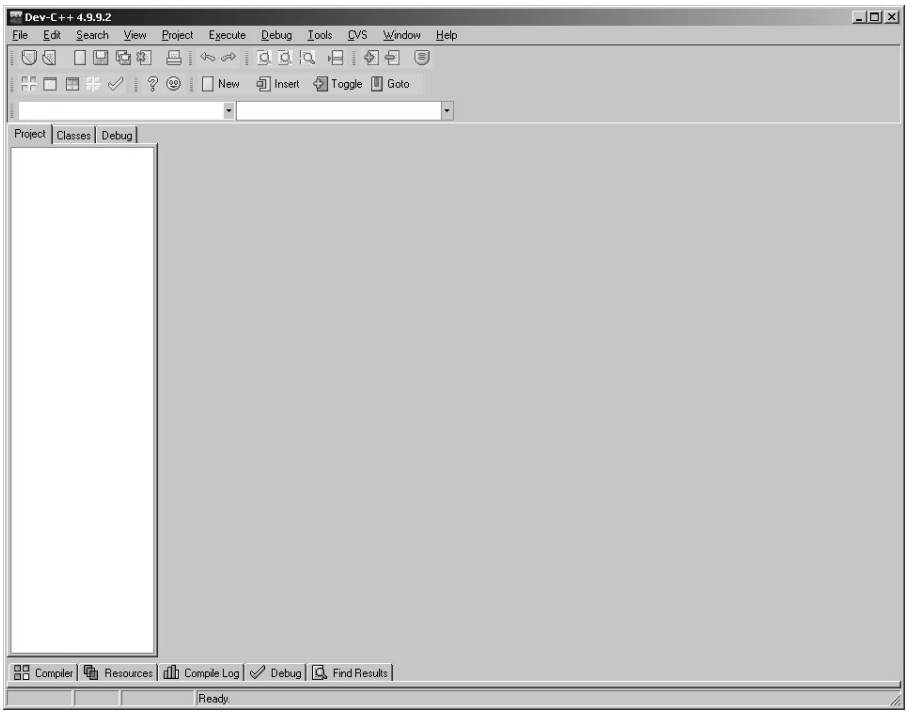


Figura 4: Janela principal do ambiente DevC++

Esta janela é dividida em três painéis:

a. Painel da esquerda. Este é o painel de projetos, utilizado em programas multiarquivos. Este painel pode ser ocultado se você preferir e estiver criando um programa monoarquivo. Se assim desejar, utilize o menu *View*.

b. Painel da direita. Este painel pertence ao editor de programas e é o principal foco de atenção quando você estiver editando um arquivo de seu programa.

c. Painel inferior. Neste painel são apresentadas mensagens geradas pelas ferramentas (e.g., compilador) usadas pelo ambiente DevC++. Este painel aparece quando uma dessas

ferramentas integradas pelo ambiente é executada ou quando você clica numa das abas na parte inferior da tela.

Antes de começar a editar qualquer programa, certifique-se de que seus programas serão compilados usando um padrão estritamente ISO. Para isso, siga estes passos:

1. Escolha a opção ***Compiler Options*** do menu ***Tools***.

2. Na caixa de diálogo que aparece, selecione a aba ***Compiler*** e digite as opções para o compilador e para o ***linker***, conforme mostrado na **Figura 5**.

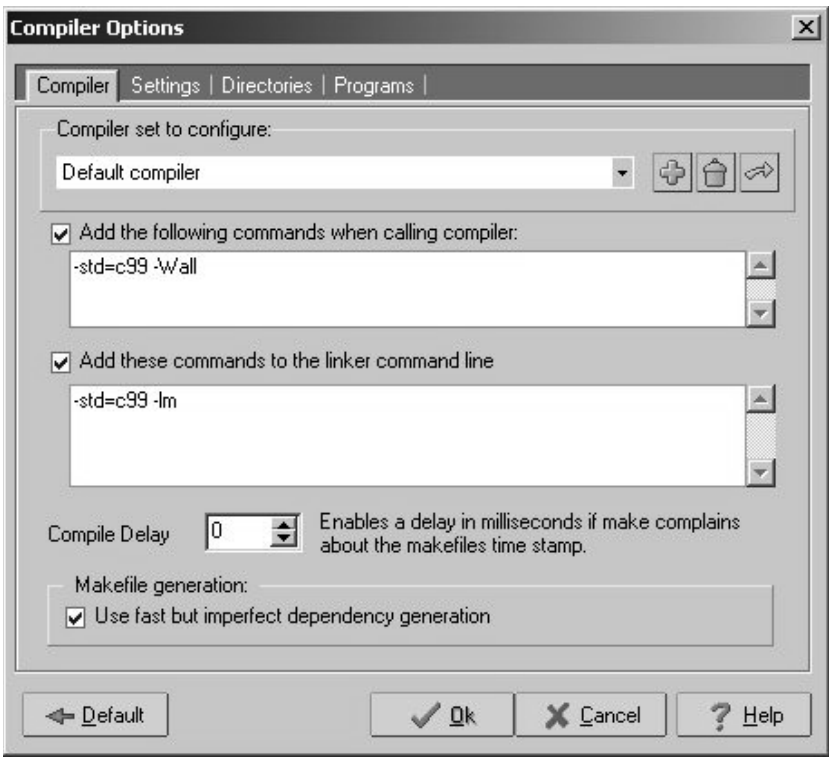


Figura 5: Caixa de configuração do ambiente DevC++

3. Clique no botão OK para fechar a caixa de diálogo.

Provavelmente, você não precisará alterar mais nenhuma outra opção de configuração e, se estiver utilizando DevC++ num computador pessoal, não precisará mais repetir essas recomendações de configuração. Entretanto, se o computador utilizado for multiusuário, é sempre bom checar se alguém não alterou as configurações antes de começar a programar.

As explicações referentes às opções de configurações escolhidas serão apresentadas na seção seguinte, dedicada ao compilador gcc, pois o ambiente DevC++ utiliza uma versão desse compilador.

▶ Editando um Programa no IDE DevC++

Para criar um programa simples usando DevC++, siga estes passos:

i. Escolha a opção *New* e, em seguida, ***Source File*** do menu *File* e você obterá, no painel da direita, um conteúdo com fundo branco[8], conforme mostrado na **Figura 6**.

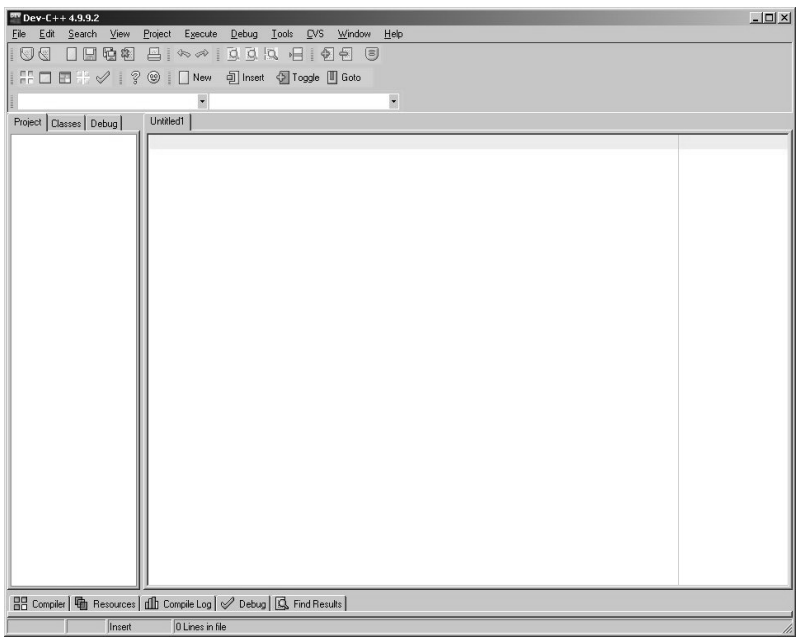


Figura 6: Painel de edição do ambiente DevC++

[8] Em algumas versões do DevC++, poderá aparecer um esqueleto de programa. Se for o caso, simplesmente selecione todo o conteúdo e apague-o.

ii. No painel da direita digite seu programa. Você pode começar com um programa tão simples quanto o seguinte:

```
#include <stdio.h>
int main()
{
   int i;

   for (i = 1; i <= 10; ++i)
      printf("Eu adoro programar em C\n");

   return 0;
}
```

iii. Salve o arquivo com o nome `AdoroC.c` utilizando a opção *Save* do menu *File*.

▶ Transformando um Programa-fonte em Executável no IDE DevC++

Para transformar seu programa em arquivo executável, escolha a opção *Compile* no menu *Execute*. Seu programa será então compilado e ligado[9]. Então, você poderá obter um dos seguintes resultados:

Situação 1

O compilador não detectou nenhuma irregularidade no seu programa. Neste caso, você obterá uma caixa de mensagem como a apresentada na **Figura 7**.

[9] No ambiente DevC++, *Compile* significa *construir um programa executável* e não, literalmente, *compilar um arquivo*.

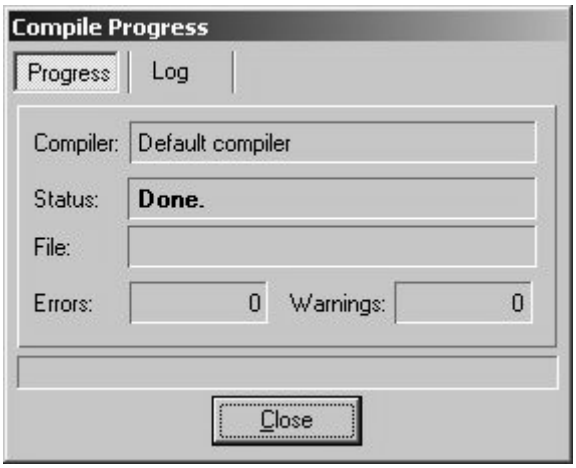


Figura 7: Caixa do DevC++ indicando compilação bem-sucedida

Esta caixa de mensagem informa que seu programa foi compilado com sucesso (***Errors: 0***). Se você editou o arquivo-fonte `AdoroC.c` sugerido, certamente atingirá esta situação. Simplesmente, clique no botão ***Close*** para fechar a caixa de diálogo e você estará apto a executar seu programa conforme descrito na **Seção 2.5**.

Situação 2

Nesta situação, o compilador não detectou nenhum erro no seu programa, mas emitiu uma ou mais mensagens de advertência. Para simular esta situação e ver o que acontece, altere a instrução **printf()** do arquivo-fonte `AdoroC.c` para o seguinte:

```
printf("Eu adoro programar em C\n", i);
```

Neste caso, quando o programa for compilado, será apresentada uma mensagem de advertência no painel inferior da janela principal, conforme a mostrada na **Figura 8**.

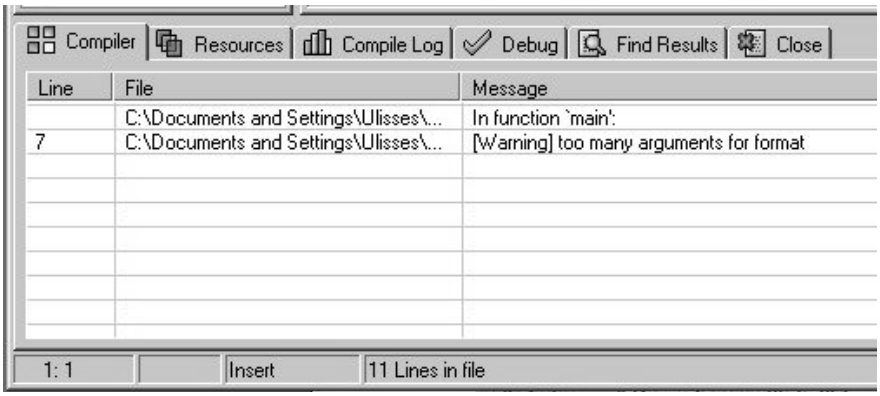


Figura 8: Mensagem de advertência apresentada pelo ambiente DevC++

Nesta última figura, a mensagem de advertência informa que existem argumentos excedentes na chamada da função **printf()**.

Mensagens de advertência não impedem que seu programa seja executado. No entanto, elas indicam situações que podem causar o mau funcionamento do seu programa. Portanto, nunca ignore completamente uma mensagem de advertência. Ao invés disso, leia e entenda por que cada mensagem de advertência foi gerada. Se, eventualmente, você decidir desprezar uma dada mensagem de advertência, convença-se de que isto não trará conseqüências danosas ao seu programa. A observância deste conselho pode lhe resguardar de muita dor de cabeça na depuração de seu programa.

Situação 3

O compilador detectou erros no seu programa. Para simular tal situação, remova o ponto-e-vírgula no final da instrução **return** do arquivo-fonte `AdoroC.c` original. Neste caso, será apresentada no painel inferior uma mensagem descrevendo aproximadamente o erro cometido, como a mostra a **Figura 9** a seguir.

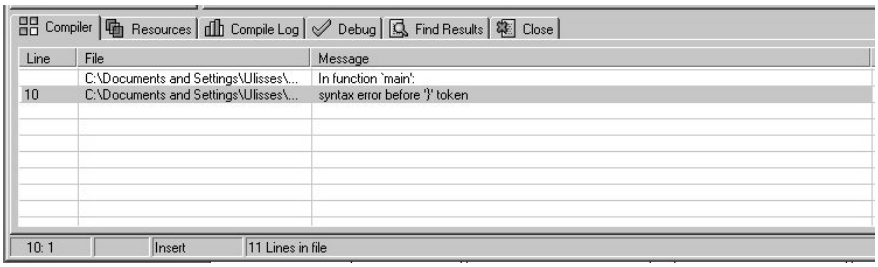


Figura 9: Mensagem de erro apresentada pelo ambiente DevC++

O que diferencia a mensagem que aparece na última figura daquela apresentada na situação anterior é que esta é uma mensagem de erro, e não de advertência como antes. Note ainda que no painel inferior aparece uma mensagem contendo uma descrição (grosseira) do erro encontrado. Este painel informa ainda em que linha (coluna 1) e em que arquivo (coluna 2) do programa o erro ocorreu. No exemplo aqui apresentado, o erro é descrito como:

```
syntax error before '}' token
```

Esta mensagem que, convenha-se, é bastante enigmática para um iniciante, informa apenas que existe um erro antes do fecha-chaves da função **main()**.

Mensagens de erro indicam que seu programa está sintaticamente incorreto e, portanto, não pode ser compilado (e muito menos executado). É importante observar ainda que um único erro pode dar surgimento a duas ou mais mensagens de erro. Portanto, enquanto você não adquire experiência suficiente para perceber isso, dê atenção a cada mensagem de erro individualmente (i.e., tente resolver uma mensagem de erro de cada vez e recompile o programa depois de cada tentativa de correção).

Apesar de o compilador muitas vezes apresentar mensagens de erro vagas e difíceis de compreender, com a experiência adquirida após encontrar várias situações semelhantes, em pouco tempo você será capaz de facilmente encontrar e corrigir erros apontados pelo compilador. Uma

opção do editor do ambiente DevC++ que facilita bastante a localização precisa de erros de sintaxe é o uso de numeração de linhas.

Para habilitar esta opção de numeração de linhas, escolha a opção *Editor Options* do menu *Tools*. Em seguida, na caixa de diálogo que aparece, clique na aba *Display* e marques as opções *Visible* e *Line Numbers* na seção denominada *Gutter*[10], conforme mostrado na **Figura 10**.

Figura 10: Habilitação de numeração de linhas no ambiente DevC++

Situação 4

Esta última situação é uma combinação das situações 2 e 3. Isto é, seu programa apresenta tanto mensagens de erro quanto mensagens de advertência. Neste caso, resolva primeiro os erros detectados pelo compilador e, em seguida, dê atenção às mensagens de advertência.

[10] *Gutter* é a calha que aparece à esquerda do painel de edição quando a opção *Visible* é selecionada e é o local onde a numeração de linhas é apresentada.

2.4.4 Utilizando Editor de Programas e gcc

▶ Editor de Texto versus Editor de Programas

Qualquer editor de texto[11] pode ser utilizado na edição de um programa, mas o mais indicado para esta tarefa é um **editor de programas**. Um editor de programas é um editor de texto que ***entende*** pelo menos uma linguagem de programação e oferece, pelo menos, as seguintes vantagens:

- **Coloração de sintaxe** – esta característica facilita a identificação visual dos diversos componentes do programa. Alguns editores de programas já vêm com uma configuração de coloração padrão que agrada a maioria dos programadores. Outros editores de programas, por outro lado, vêm com uma configuração de coloração exagerada e de gosto bastante duvidoso. Em qualquer caso, a maioria dos editores de programas permite que o usuário altere a configuração do padrão de cores. Utilize esta facilidade com coerência e parcimônia e não transforme a visualização de seu programa num maracatu.
- **Endentação automática** – as vantagens obtidas com o uso de endentação são enfatizadas em vários pontos deste livro. Esta facilidade permite que o programador não tenha que fazer endentação manualmente, economizando, assim, tempo e esforço físico despendidos em digitação. Como no caso de coloração de sintaxe, os espaços de endentação automática também podem ser configurados. Três ou quatro espaços em branco são considerados a configuração ideal de endentação.

A maioria dos editores de programas oferece outras características úteis, tais como completação de código e emparelhamento de parênteses, chaves e colchetes, mas aquelas descritas acima são suficientes para começar a programar.

[11] Não confunda processador de texto com editor de texto. Um processador de texto, tal como Microsoft Word, inclui formatação de texto que é obviamente útil em outros contextos mas não é conveniente para a escrita de programas.

Se você utiliza algum sistema operacional da família Windows da Microsoft, o editor de programas recomendado é TextPad, devido a sua disponibilidade na internet e facilidades de instalação, configuração e uso. Para o sistema operacional Linux recomenda-se, pelas mesmas razões expostas para TextPad, SciTE ou KWrite. Existem, evidentemente, outros editores que são muito mais poderosos (por exemplo, vi e gVim), mas estes são muito mais difíceis de instalar, configurar e usar. Se você possui familiaridade com algum outro editor de programas, é melhor utilizá-lo; não invista seu tempo aprendendo a utilizar editores mais complicados enquanto seu objetivo principal é aprender a programa em C.

▶ O Compilador gcc

O compilador gcc[12] pode ser encontrado em todas as principais distribuições de Linux e é provavelmente o melhor compilador de C que existe atualmente. Duas das principais vantagens oferecidas pelo gcc são produção de código muito eficiente e suporte quase completo ao padrão C99. O compilador gcc pode ser executado em sistemas operacionais da família Windows utilizando como intermediário o programa Cygwin, mas, neste caso, infelizmente, o gcc não é fácil de instalar nem funciona tão bem quanto em Linux.

Para instalar o gcc em conjunto com o programa Cygwin num sistema Windows, visite o site do Cygwin (www.cygwin.com) e instale o pacote básico do Cygwin. Em seguida, instale o pacote Devel[13].

▶ Obtendo Um Executável Padrão

Suponha que você tenha editado um programa simples constituído de um único arquivo-fonte. Então, o próximo passo é obter um arquivo executável por meio de compilação e edição de ligações.

---

[12] Quando foi criado *gcc* era um acrônimo derivado de *Gnu C Compiler*. Atualmente, o nome *gcc* representa *Gnu Compiler Collection*.

[13] Na realidade, os únicos programas que interessam aqui são gcc, gdb e ddd (estes dois últimos serão discutidos no **Capítulo 6**). Isto é, se você instalar todo o pacote Devel, muitos programas que, provavelmente, não serão utilizadas serão instalados, mas é mais fácil do que selecionar cada programa individualmente (a não ser que você tenha problema de espaço de armazenamento).

O modo mais simples de construir um programa executável usando o compilador gcc a partir um programa constituído de um único arquivo-fonte é por meio do comando:

```
gcc nome-do-arquivo-fonte
```

onde *nome-do-arquivo-fonte* deve incluir a extensão do arquivo (usualmente `.c`).

Usando este comando, o compilador gcc não apenas compila o arquivo-fonte como invoca o ***linker*** para fazer as devidas ligações no arquivo objeto resultante da compilação. O resultado final do processo é um programa executável denominado `a.out` (usando Linux) ou `a.exe` (usando Windows/DOS). Portanto, o problema com este comando é que o nome do arquivo resultante é sempre o mesmo[14].

Por exemplo, se você tem um programa contido no arquivo `AdoroC.c`, o comando:

```
gcc AdoroC.c
```

produzirá um arquivo executável denominado `a.out` (usando Linux) ou `a.exe` (usando DOS).

▶ Especificando o Nome do Arquivo Executável

Uma variante do comando do gcc apresentado anteriormente permite especificar o nome do arquivo executável resultante:

```
gcc nome-do-arquivo-fonte -o nome-do-arquivo-executável
```

onde *nome-do-arquivo-executável* representa o nome do arquivo resultante da compilação e ligação.

[14] Cuidado, pois se já houver um arquivo com o mesmo nome do arquivo gerado no diretório corrente, este será substituído sem advertência prévia.

Exemplo:

```
gcc AdoroC.c –o AdoroC.exe
```

produzirá um arquivo executável denominado `AdoroC.exe`.

▶ Compilação e Ligação Separadas

Para compilar (literalmente) um arquivo-fonte, utiliza-se a opção `–c` do compilador gcc:

```
gcc –c nome-do-arquivo-fonte -o nome-do-arquivo-objeto
```

Neste caso, recomenda-se o uso de uma extensão que caracterize o arquivo resultante como sendo um arquivo objeto, mas não-executável. Note que, se a opção `–o` *`nome-do-arquivo-objeto`* não for utilizada, o arquivo resultante terá o mesmo nome do arquivo-fonte e a extensão `.o`.

Como exemplo de uso do gcc com a opção `–c` considere a invocação:

```
gcc –c AdoroC.c –o AdoroC.obj
```

que resulta no arquivo objeto (não-executável) `AdoroC.obj`. Por outro lado, se o gcc for invocado como:

```
gcc –c AdoroC.c
```

o resultado será um arquivo objeto denominado `AdoroC.o`.

Uma vez obtido o arquivo objeto usando o gcc com a opção `–c`, o arquivo executável correspondente pode ser obtido invocando o ***linker*** por meio do seguinte comando:

```
gcc nome-do-arquivo-objeto -o nome-do-arquivo-executável
```

Por exemplo, supondo que tenha previamente obtido o arquivo `AdoroC.obj`, o comando:

```
gcc AdoroC.obj –o AdoroC.exe
```

cria um arquivo executável denominado `AdoroC.exe`.

Em termos práticos, não há nenhum ganho resultante do uso de compilação e ligação em dois passos separados no caso de programas monoarquivos. Ou seja, é muito mais prático compilar esses programas usando as opções descritas anteriormente. No entanto, como será visto na **Seção 4.11**, esta separação entre compilação e ligação é muitas vezes a opção mais prática no caso de programas multiarquivos.

▶ Obtendo Mensagens de Advertência

Normalmente, um compilador apresenta apenas mensagens de erro relacionadas ao uso ilegal da linguagem, mas pode-se instá-lo a apresentar mensagens que advertem o programador sobre construções do programa que podem eventualmente causar um comportamento anormal dele. Por exemplo, o compilador não tem a obrigação de apresentar nenhuma mensagem sobre o uso do valor de uma variável não-iniciada, mas pode fazê-lo se for solicitado. Utilizando o compilador com a opção `-Wall`, o programador obtém mensagens de advertência sobre todas as construções do programa que o compilador julgar suspeitas. Estas mensagens são apenas advertências e não impedem que o programa seja compilado ou ligado. Entretanto, é uma boa medida de precaução levar sempre em consideração estas mensagens de advertência e corrigir o programa para evitar que cada uma delas deixe de ser apresentada.

A opção `-Wall` pode ser acrescentada a qualquer outra opção de invocação do compilador gcc. Por exemplo:

```
gcc -Wall AdoroC.c -o AdoroC
```

▶ Habilitando o Padrão C99

Na ausência de especificação alternativa, o compilador gcc utiliza correntemente o padrão C90. Se você desejar utilizar as novidades introduzidas pelo padrão C99, utilize a opção `-std=c99`, como, por exemplo[15]:

```
gcc -Wall -std=c99 AdoroC.c -o AdoroC
```

[15] Eventualmente, o padrão ***default*** tornar-se-á C99 e essa opção será, então, redundante.

2.5 Executando um Programa

Executar um programa baseado em console, como aqueles que se ensina a desenvolver aqui, é fácil em sistemas da família Unix/Linux. Supondo que você esteja correntemente no diretório onde se encontra o arquivo executável, basta digitar[16]:

```
./nome-do-arquivo-executável
```

Por outro lado, executar um programa baseado em console em sistemas operacionais da família Microsoft Windows não é tão simples quanto parece. Localize um programa executável construído de acordo com as seções anteriores e tente executá-lo como se fosse um programa qualquer do Windows (i.e., usando um clique duplo sobre o ícone do programa). Se você fizer isso, talvez tenha uma desagradável surpresa, pois você não verá o resultado esperado, mas simplesmente obterá um breve piscado na tela. Isto não significa que seu programa está incorreto, quer dizer apenas que você ainda não sabe como executá-lo.

Os programas aqui desenvolvidos são dirigidos ao DOS, um antigo sistema operacional que tornou-se obsoleto após a introdução do sistema Windows95. Em sistemas operacionais Windows da Microsoft (com exceção da geração Windows 3.x), existe emulação do sistema operacional DOS[17]. Quando você executa um programa do DOS sob o Windows, este sistema age como agiria como se estivesse executando qualquer outro programa: uma janela é aberta (neste caso uma janela de emulação do DOS), o programa é executado e, finalmente, a janela é fechada quando o programa encerra. Portanto, para que você veja algum resultado da execução de um programa DOS nestas circunstâncias, é necessário que o programa seja executado por um longo período de tempo ou que sua execução seja interrompida.

---

[16] Se o diretório onde se encontra o arquivo fizer parte da variável de ambiente `PATH`, torna-se desnecessário usar `./`. Se você não entende nada disso relacionado à execução de um programa no sistema Linux, procure um bom livro sobre o assunto.

[17] Isto é, o sistema DOS não mais existe, mas o Windows permite que programas do DOS sejam executados num ambiente que simula o funcionamento do antigo DOS

Essa situação parece ter propagado a idéia de que qualquer programa escrito em C deve conter uma instrução tal como[18]:

```
getchar(); // Esta opção é ruim
```

ou

```
system("PAUSE"); // Esta opção é pior, pois PAUSE não é portável
```

sem a qual seria impossível ler no terminal aquilo que o programa escreve.

Todavia, o problema de não-visualização de resultados impressos pelo programa nessas circunstâncias é devido ao simples fato de o sistema Windows fechar a janela de console DOS quando a execução do programa é encerrada, se esta janela tiver sido aberta apenas para a execução do programa. Uma forma de contornar este problema é simplesmente executar o prompt de comando[19], navegar até o diretório onde se encontra seu programa e, na janela de console DOS, digitar o nome do programa executável. Assim, você verá o resultado do programa, como mostrado na **Figura 11** a seguir.

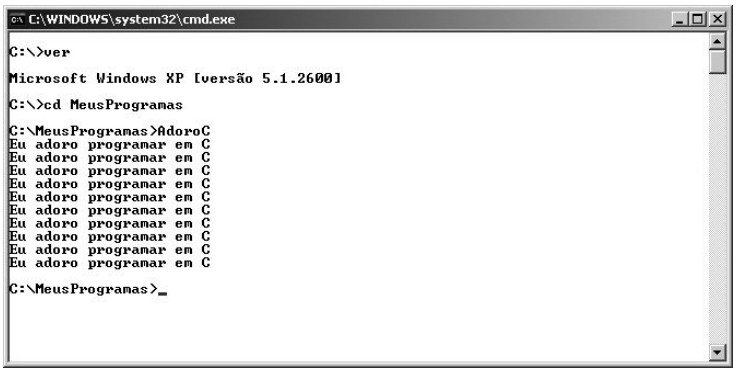


Figura 11: Execução de um programa no Sistema DOS/Windows

[18] **NB**: Não há mal nenhum em, circunstancialmente, utilizar, por exemplo, **getchar()** com o objetivo de evitar que a janela de console desapareça antes que se tenha oportunidade de examinar o que o programa escreveu. O mal é achar que incluir tal instrução é essencial num programa em C em todas as situações.

[19] Esta denominação é utilizada no sistema Windows XP. Em outras versões de Windows, este programa pode ter outras denominações (por exemplo, Prompt do DOS), mas as recomendações apresentadas aqui também se aplicam a essas outras versões.

2.6 Usando a Entrada de Dados Padrão

Na prática, usar **getchar()** ou qualquer outra função que faz leitura na entrada padrão não é tão trivial quanto parece. As dificuldades residem no fato de a leitura de dados depender do comportamento do usuário do programa. Isto é, diferentemente da saída de dados, que depende apenas do programador, a entrada de dados depende do modo como o usuário introduz os dados para o programa, o que nem sempre corresponde àquilo que é esperado. Um programa, portanto, deve ser capaz de responder adequadamente a qualquer dado, correto ou incorreto, introduzido pelo usuário.

Para ser capaz de construir programas robustos (i.e., resistentes a usuário), é necessário, em primeiro lugar, entender o funcionamento da entrada de dados padrão, normalmente efetuada via teclado. Em seguida, é necessário aprender a construir trechos de programa dedicados à leitura e análise de dados introduzidos pelo usuário.

2.6.1 Entendendo Entrada de Dados Padrão

A entrada de dados padrão para programas escritos em C é normalmente efetuada por meio do teclado. Entretanto, quando uma função de leitura [por exemplo, **getchar()**] é executada, ela não tenta ler dados diretamente no dispositivo de entrada padrão. Ao invés disso, a leitura é feita numa região de memória, denominada **buffer**, para onde os caracteres introduzidos pelo usuário são enviados. Entender o funcionamento desta área de buffer é fundamental para lidar com possíveis problemas com leitura de dados.

Em primeiro lugar, deve-se ressaltar que uma função de leitura só causa interrupção do programa à espera da introdução de dados se ela não encontra dados (caracteres) no buffer associado com a entrada de dados padrão. Considere, por exemplo, o seguinte programa:

```
/**** Programa 1 ****/

#include <stdio.h>

int main(void)
{
```

```
    unsigned char umChar;

    printf("Digite um caractere:");
    umChar = getchar();

    printf("Digite outro caractere:");
    umChar = getchar();

    return 0;
}
```

Este programa é extremamente simples, mas causa enorme frustração ao programador iniciante que não entende o funcionamento da entrada de dados padrão. Tudo o que este programa faz é solicitar que o usuário introduza dois caracteres e ler estes 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 o que ocorre.

Para entender melhor o que se está afirmando, execute o programa anterior e tente introduzir o caractere `A` quando for impresso na tela *`Digite um caractere:`* e o caractere `B` quando for impresso *`Digite outro caractere:`*. Seguindo essas instruções, você verá que, quando tentar introduzir o caractere `B`, o programa já terá encerrado. Por que o programa age desta maneira? Para entender o que realmente acontece com esse programa, acompanhe, a seguir, sua execução passo a passo.

A primeira instrução do programa é:

```
printf("Digite um caractere:");
```

que imprime na tela do computador:

```
Digite um caractere:
```

Em seguida, a função **getchar()** começa a ser executada. Como esta função não encontra nenhum caractere armazenado no buffer, ela causa a interrupção do programa à espera de que um caractere seja colocado no buffer. Quando você digita o caractere `A`, precisa também digitar `[ENTER]` (ou `[RETURN]`) para encerrar a entrada de dados. Acontece que `[ENTER]` também representa um caractere, que é o caractere de quebra de

linha, representado em C por `'\n'`. Portanto, o conteúdo do buffer nesse instante, após a digitação desses caracteres, é:

Observe que os caracteres são armazenados no buffer na ordem em que são digitados. Nesse instante, a função **getchar()**, que estava à espera de que algum caractere fosse armazenado no buffer, conclui sua execução lendo o primeiro caractere que encontra no buffer. Então, este caractere é removido do buffer, que fica com o seguinte conteúdo:

Em seguida, é executada a próxima instrução do programa:

```
printf("Digite outro caractere:");
```

causando a impressão de:

```
Digite um caractere:
```

Em seguida, a segunda chamada de **getchar()** é executada. Agora, diferentemente do que ocorre com a primeira chamada desta função, ela encontra um caractere (i.e., `'\n'`) no buffer de entrada (v. última figura) e, portanto não interrompe a execução do programa à espera de outro caractere. Neste caso, a função simplesmente lê o caractere encontrado no buffer. Assim, o usuário não tem chance de digitar outro caractere, pois, logo em seguida, o programa é encerrado.

Antes de apresentar uma solução para o problema apresentado pelo programa anterior, é importante ressaltar que outras funções de entrada podem deixar caracteres remanescentes que podem causar o mesmo problema em tentativas de leituras subseqüentes. Considere agora o seguinte programa:

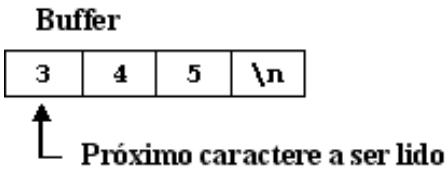
```
/**** Programa 2 ****/

#include <stdio.h>

int main(void)
{
   unsigned char umChar;
   int           umInt, nValores;

   printf("Digite um valor inteiro:");
   nValores = scanf("%d", &umInt);
   printf("Numero de valores lidos: %d\nValor lido:%d\n",
            nValores, umInt);

   printf("Digite um caractere:");
   umChar = getchar();

   return 0;
}
```

Um exemplo de interação com o último programa é aqui apresentado[20]:

```
E:>Entrada2
Digite um valor inteiro:345
Numero de valores lidos: 1
Valor lido: 345
Digite um caractere:
E:>
```

Novamente, o programa não permite que o usuário digite o último caractere solicitado. Neste caso, quanto o usuário digita `345` seguido de `[ENTER]`, o conteúdo do buffer torna-se:

[20] Assuma que este programa está sendo executado no comando DOS do sistema operacional Windows XP e que o nome do programa é `Entrada2`.



Nem sempre a função **scanf()** lê e remove um único caractere do buffer de entrada, como sempre faz a função **getchar()**. Neste caso específico, a função **scanf()** lê o número de caracteres suficientes para formar um número inteiro, devido ao uso do especificador de formato `%d`. Ou seja, esta função lê os caracteres "3", "4" e "5" e deixa no buffer de entrada o caractere `'\n'`. O resto da história é a mesma do penúltimo exemplo.

A conclusão que se tira dos exemplos apresentados nesta seção é que, antes de fazer uma nova leitura no meio de entrada padrão, é necessário garantir que o buffer associado a este meio de entrada esteja vazio[21]. Existem várias maneiras de se fazer isto e uma delas é chamar a função `LimpaBuffer()`, definida a seguir:

```
void LimpaBuffer(void)
{
    int valorLido; /* valorLido deve ser int para poder conter
EOF */

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

A função `LimpaBuffer()` simplesmente lê e descarta repetidamente todos os caracteres encontrados no buffer de entrada até encontrar o caractere `'\n'` ou **EOF**[22]. Apesar da simplicidade desta função, não existe

[21] A não ser que se deseje realmente ler caracteres remanescentes de uma entrada de dados anterior, o que é raro.

[22] Na realidade, EOF, como será visto no Capítulo 12, não representa um caractere lido, mas sim um valor inteiro negativo que indica que se tentou ler além do final do buffer.

nenhuma função na biblioteca padrão de C que execute esta operação[23].

Portanto, para resolver os problemas apresentados nos últimos dois programas, deve-se acrescentar a função `LimpaBuffer()` a estes programas e alterá-los da seguinte maneira:

```
/**** Programa 1 ****/

#include <stdio.h>

void LimpaBuffer(void)
{
   int valorLido;

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

int main(void)
{
   unsigned char umChar;

   printf("Digite um caractere:");
   umChar = getchar();
   LimpaBuffer();

   printf("Digite outro caractere:");
   umChar = getchar();

   return 0;
}

/**** Programa 2 ****/

#include <stdio.h>

void LimpaBuffer(void)
{
   int valorLido;

   do {
```

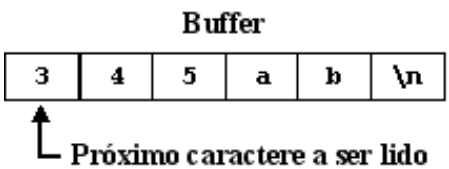
[23] Alguns livros de programação utilizam a função fflush() com esta finalidade, mas esta função deve ser utilizada apenas para buffers de saída e não de entrada, como é o caso aqui.

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

int main(void)
{
    unsigned char umChar;
    int           umInt, nValores;

    printf("Digite um valor inteiro:");
    nValores = scanf("%d", &umInt);
    printf("Numero de valores lidos: %d\nValor lido: %d\n",
            nValores, umInt);

    LimpaBuffer();

    printf("Digite um caractere:");
    umChar = getchar();

    return 0;
}
```

Observe que, agora, os programas corrigidos funcionam mesmo quando o usuário digita caracteres além do esperado. Por exemplo, suponha a seguinte execução do programa 2 anterior:

```
E:>Entrada2
Digite um valor inteiro:345ab
Numero de valores lidos: 1
Valor lido: 345
Digite outro caractere: u
E:>
```

Note que, nesta última execução do programa 2, o usuário digitou `345ab` em vez do número inteiro solicitado. Portanto, logo após a introdução destes caracteres, o conteúdo do *buffer* de entrada é o seguinte:



Como antes, a função **scanf()** lê caracteres que possam compor um número inteiro e pára no primeiro caractere que não pode fazer parte do número inteiro. No último exemplo de interação, esta função deixava no buffer apenas o caractere `'\n'`, mas agora ela deixa os caracteres `'a'`, `'b'` e `'\n'`. Entretanto, isto não mais constitui problema, pois a função `LimpaBuffer()` remove todos os caracteres do buffer de entrada, não importando quanto ou quais são esses caracteres.

Agora, suponha que, em vez de digitar `345ab`, como foi o caso, o usuário digite `ab` quando solicitado a digitar um número inteiro. Neste caso, se o programa não precisar do valor inteiro para prosseguir (o que, provavelmente, não faria sentido na prática), o programa 2 anterior é suficiente (i.e, ele ainda funciona). Mas, e se o programa realmente necessita desse valor para prosseguir? A seção a seguir lida com esta categoria de problemas.

2.6.2 Uso de Laços de Repetição em Leitura de Dados

Laços de repetição são utilizados em leitura de dados para oferecer uma nova chance ao usuário após ele ter introduzido uma entrada de dados incorreta para o programa. Para entender como isso funciona, considere o programa a seguir:

```
#include <stdio.h>

int main(void)
{
   int  umInt, outroInt;

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

   printf("Digite outro valor inteiro: ");
   scanf("%d", &outroInt);

   printf("\n%d + %d = %d", umInt, outroInt, umInt + outroInt);

   return 0;
}
```

Este programa funciona perfeitamente bem desde que o usuário ***seja bem comportado***, isto é, se ele introduzir corretamente dois números inteiros. Mas, este é justamente o problema deste programa: ele depende do usuário para funcionar bem. Um bom programador não deve jamais fazer suposições de bom comportamento por parte do usuário. O que aconteceria, então, se o usuário não se comportasse conforme o esperado? Algumas possibilidades serão analisadas a seguir.

Suponha, inicialmente, que, quando solicitado a introduzir o primeiro inteiro, o usuário digite `345ab`. Conforme foi visto na seção anterior, neste caso a função **scanf()** lê e transforma em número inteiro os caracteres `'3'`, `'4'` e `'5'` e deixa no buffer os caracteres `'a'`, `'b'` e `'\n'`. A solução para este problema também foi apresentada na seção anterior[24]. Ou seja, basta colocar uma chamada da função `LimpaBuffer()` descrita na seção anterior e este problema estará resolvido.

Agora suponha que o usuário digite `ab25` quando solicitado a introduzir o primeiro inteiro. Nesta situação, a função **scanf()** não será capaz de transformar a entrada num número inteiro, pois o primeiro caracteres não-branco encontrado é `'a'` e este caractere não pode fazer parte de um número inteiro na base decimal. Conseqüentemente, nenhum valor será atribuído à variável `umInt`.

Como primeira tentativa para tentar solucionar este último problema, deve-se testar se a função **scanf()** lê realmente um número inteiro como deveria. Isto é feito testando-se o valor retornado pela função **scanf()**, como mostrado no fragmento de programa a seguir:

```
int  umInt, outroInt, nValoresLidos;

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

[24] Note que se o usuário digitar apenas um número inteiro, a função **scanf()** deixa o caractere `'\n'` no buffer, mas isto não constitui problema para a próxima chamada de **scanf()** porque, neste caso, a função **scanf()** salta caracteres em branco encontrados no início do buffer. A função **scanf()** só não age deste modo quando se utiliza o especificador `%c` para leitura de um único caractere.

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

O trecho de programa acima é fácil de ser entendido. Como foi dito na **Seção 2.3.4**, a função **scanf()** retorna o número de variáveis, cujos endereços são recebidos como argumentos, que tiveram seus valores alterados pela função. No caso em questão, a função **scanf()** recebe apenas um endereço de variável (i.e., `&umInt`) como argumento e, portanto, o valor retornado só poderá ser 0 quando esta variável não for alterada, devido a um erro ocorrido na entrada de dados, ou 1, quando tudo ocorrer bem e esta função atribuir o valor lido à variável. Portanto, a instrução **if** do trecho de programa testa se ocorreu algum erro na entrada de dados do usuário e, se for este o caso, solicita ao usuário que introduza um novo valor inteiro. Antes da nova leitura, porém, chama-se a função `LimpaBuffer()` para promover a remoção dos caracteres que ficaram armazenados no buffer de entrada em conseqüência dos dados incorretamente digitados pelo usuário.

A correção apresentada no trecho de programa acima quando introduzida no programa anterior melhora a qualidade deste programa porque permite que o usuário tenha uma nova chance para corrigir um eventual erro. Mas esta ainda não é uma solução ideal, pois não leva em consideração que o usuário pode errar mais de uma vez. Portanto, a melhor solução deve permitir que o usuário tente digitar corretamente tantas vezes quanto forem necessárias. Logo, o ideal seria construir um laço de repetição do qual só se sairá quando o usuário digitar corretamente o dado desejado. Uma pequena alteração no trecho de programa anterior produz o resultado desejado:

```
int  umInt, outroInt, nValoresLidos;

printf("Digite um valor inteiro: ");

leitura1:
   nValoresLidos = scanf("%d", &umInt);
```

```
if (nValoresLidos == 0) { // Nenhum inteiro foi lido
   LimpaBuffer();
   printf("Entrada incorreta. Digite um valor inteiro:");
   goto leitura1;
}
```

O uso de **goto** nessa situação é perfeitamente aceitável, pois não causa nenhuma perda de legibilidade do programa, além de resultar numa codificação eficiente. Mas, se você tem alguma cisma com o uso de **goto**, pode substituí-lo pelo laço de repetição **while** equivalente apresentado a seguir:

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

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

Fazendo as devidas alterações, o programa apresentado no início desta seção ficaria assim:

```
#include <stdio.h>

void LimpaBuffer(void)
{
   int valorLido; /* valorLido deve ser int! */

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

int main(void)
{
   int  umInt, outroInt, nValoresLidos;

printf("Digite um valor inteiro: ");
```

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

   printf("Digite outro valor inteiro: ");
   nValoresLidos = scanf("%d", &outroInt);

   while (nValoresLidos == 0) {
      LimpaBuffer();
       printf("Entrada incorreta. Digite um valor inteiro: ");
      nValoresLidos = scanf("%d", &outroInt);
   }

   printf("\n%d + %d = %d", umInt, outroInt, umInt + outroInt);

   return 0;
}
```

Observe que o problema resolvido pelo programa apresentado nesta seção é bastante trivial: ler dois números inteiros e apresentar a soma dos mesmos. Você esperava que um problema tão simples requeresse um programa com tantas instruções e relativamente complexo? Espera-se que você esteja convencido de quão relativamente complexa é a tarefa de ler dados na entrada de dados padrão, conforme foi afirmado anteriormente. Esta seção mostra ainda como é complicado interagir com o usuário, pois um bom programa deve prever todas as ações que o usuário possa realizar durante a interação com o programa.

2.7 Exercícios de Revisão

1. O programa a seguir é legal em C?

```
int main()
{
    (1 + 2) * 4;
    return  0;
}
```

2. Ambos os trechos de programa a seguir contêm erros:

Trecho de programa 1	Trecho de programa 2

`int x = 0;`
`if (x = 10)`
`  y += 1;`
	`int x = 0;`
	`if (10 = x)`
	`  y += 1;`

Por que o compilador indica uma ocorrência de erro no trecho de programa 2, mas o mesmo não ocorre com o trecho de programa 1?

3. O que é um ambiente integrado de desenvolvimento (IDE)?

4. Que opção do compilador gcc é utilizada para a geração do maior número de mensagens de erro?

5. (a) O que é a biblioteca padrão de C? (b) Por que funções desta biblioteca não são estritamente consideradas partes integrantes da linguagem C?

6. O que é um arquivo de cabeçalho da biblioteca padrão de C?

7. Para que serve a diretiva **#include**?

8. Descreva o funcionamento das seguintes funções:
(a) getchar()
(b) putchar()
(c) scanf()
(d) printf()

9. (a) Qual é o significado do valor retornado pela função **scanf()**? (b) Como este valor deve ser utilizado?

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

11. Por que nem sempre a função **getchar()** interrompe a execução do programa e espera que o usuário digite um caractere?

12. Para que serve a função `LimpaBuffer()` apresentada na **Seção 2.6.1**?

13. Por que o programa a seguir não imprime o resultado esperado?

```
#include <stdio.h>

int main()
{
   int resultado = 2 + 2;

   printf("O Resultado e': %d\n");

   return 0;
}
```

14. Por que o programa a seguir imprime: *O valor de 2/3 e' 0.000000*?

```
#include <stdio.h>

int main()
{
   float produto = 1/3;

   printf("O valor de 2/3 e' %f\n", produto);

   return 0;
}
```

15. Por que o programa a seguir imprime sempre a mesma mensagem: *Voce tem credito de 0*?

```
#include <stdio.h>

int main()
{
   int dividaInicial, pago, debito;

   printf("Divida inicial: ");
   scanf("%d", &dividaInicial);

   getchar();
```

```
   printf("Quanto voce pagou? ");
   scanf("%d", &pago);

   debito = pago - dividaInicial;

   if (debito = 0)
      printf("Voce nao deve nada\n");
   else if (debito < 0)
      printf("Voce deve %d\n", -debito);
   else
      printf("Voce tem credito de %d\n", debito);

   return 0;
}
```

16. O programa do exercício anterior depende muito do bom comportamento do usuário. (a) Critique este programa à luz daquilo que foi discutido na **Seção 2.6**. (b) Re-implemente este programa de modo que ele seja compatível com os ensinamentos encontrados na **Seção 2.6**.

17. Por que o programa a seguir *acredita* que zero não é zero?

```
#include <stdio.h>

int main()
{
   int  numero;

   printf("Introduza um numero inteiro: ");

   scanf("%d", &numero);

   if (numero =! 0)
      printf("O numero NAO e' zero\n");
   else
      printf("O numero e' zero\n");

   return 0;
}
```

18. O programa a seguir foi criado com o objetivo de construir uma tabela de conversão entre graus centígrados e Fahrenheit para os 100 primeiros valores de graus centígrados.

```
#include <stdio.h>

int main()
{
   int cent;

   printf("Centigrados\t\tFahrenheit\n");

   for (cent = 0; cent <= 100; ++cent);
     printf("%d\t\t\t%d\n", cent, (9*cent)/5 + 32);

    return 0;
}
```

No entanto, o programa consegue imprimir apenas:

```
Centigrados             Fahrenheit
101                     213
```

Explique por que o programa não funciona conforme deveria e apresente uma maneira de corrigi-lo.

19. Por que o programa a seguir não imprime o resultado desejado?

```
#include <stdio.h>

int main()
{
    float resultado = 7.0 / 22.0;

    printf("Resultado da divisao: %d\n", resultado);
    return 0;
}
```

2.8 Exercícios de Programação

EP2.1) Escreva um programa em C que imprima o seguinte na tela:

```
Meu nome e' [apresente aqui seu nome]
Sou [aluno ou aluna] do curso [nome do curso] e este e'
meu primeiro programa em C.
Digite uma tecla seguida de ENTER para encerrar o
programa:
```

Este programa deverá apresentar os conteúdos entre colchetes substituídos adequadamente e só deverá ter sua execução encerrada quando o usuário digitar alguma tecla seguida de `[ENTER]` (ou `[RETURN]`).

SUGESTÃO

Utilize as recomendações contidas na **Seção 2.4**.

EP2.2) (a) Escreva um programa em C que utilize o operador **sizeof** em conjunto com a função **printf()** para determinar e imprimir o número de bytes ocupados por valores dos seguintes tipos primitivos de C:

(i) **int**
(ii) **short**
(iii) **long**
(iv) **long long**
(v) **float**
(vi) **double**
(vii) **long double**

A saída deste programa deve ser algo como: *`O tipo short ocupa 4 bytes`*.

(b) Sabendo o número de bytes ocupado pelos tipos **short**, **int**, **long** e **long long**, determine analiticamente o intervalo de possíveis valores dos tipos:

(i) **short**
(ii) **unsigned short**
(iii) **int**
(iv) **unsigned int**
(v) **long int**
(vi) **unsigned long**
(vii) **long long int**
(viii) **unsigned long long**

(c) Você poderia repetir a tarefa solicitada no item anterior para os tipos de ponto-flutuante? Explique que conhecimentos adicionais você precisaria ter, além do conhecimento básico sobre aritmética binária, para efetuar esta tarefa.

EP2.3) Escreva um programa em C que imprima as letras de `A` a `Z` (maiúsculas) e seus respectivos valores decimais.

EP2.4) Escreva um programa em C que receba um número inteiro como entrada e determine se o mesmo é par ou ímpar. O programa deve terminar quando um número inteiro negativo for introduzido. A saída do programa deve ser algo como: *`O numero introduzido e' par`*.

EP2.5) O programa a seguir contém erros sintáticos e resulta em mensagens de advertência quando se tenta compilá-lo com o compilador gcc ou no ambiente DevC++ usando a opção `-Wall`:

```
#include <stdio.h>  /* Permite o uso de scanf() e printf() */

int main(void)
{
       int x, y, z;  /* Declaração das variáveis inteiras x, y e z */

       x = 10; /* Atribui 10 à variável x */
       Y = 0; /* Atribui 0 à variável y  */

       SCANF("%d", &z); /* Lê um valor para a variável z */
       printf("%d", y); /* Imprime o valor da variável y */

       return 0
}
```

Sua tarefa consistirá no seguinte:

i) Usando o ambiente de programação DevC++, edite o programa anterior ***exatamente*** como ele é apresentado aqui e salve o arquivo.

ii) Compile o programa conforme descrito na **Seção 2.4**. Você deverá obter como resultado uma janela contendo quatro mensagens de erro e duas mensagens de advertência.

iii) Descreva o significado e a causa de cada mensagem de erro e de advertência.

iv) Para cada mensagem de erro, descreva uma forma de contornar o problema.

v) Para cada mensagem de advertência, descreva uma forma de evitar que o compilador continue a emitir a referida advertência.

vi) Implemente as mudanças sugeridas em iv) e v) e apresente o conteúdo do programa modificado. Certifique-se de que este programa realmente seja compilado sem nenhuma mensagem de erro ou advertência.

EP2.6) O arquivo de cabeçalho `<limits.h>`, que faz parte da biblioteca padrão de C, define constantes simbólicas que representam valores limites (máximos e mínimos) dos tipos inteiros primitivos da linguagem C, bem como outros valores dependentes de implementação. Encontre o arquivo `limits.h` no diretório de instalação do ambiente DevC++, caso você esteja usando este ambiente, ou no diretório `/usr/include`, caso esteja usando Linux e gcc. Abra este arquivo com um editor de texto qualquer e encontre respostas para as questões a seguir. (**NB**: Não tente entender todo o conteúdo do arquivo `limits.h` ou qualquer outro arquivo de cabeçalho da biblioteca padrão de C, pois existem várias construções de C que serão vistas mais adiante no desenrolar do livro.)

(a) Que constante simbólica representa:

(i) O número de bits em um byte? Será que esta constante é realmente necessária?

(ii) O valor mínimo do tipo **short**?

(iii) O valor máximo do tipo **short**?

(iv) O valor mínimo do tipo **unsigned short**?
(v) O valor máximo do tipo **unsigned short**?
(vi) O valor mínimo do tipo **long**?
(vii) O valor máximo do tipo **long**?
(viii) O valor mínimo do tipo **unsigned long**?
(ix) O valor máximo do tipo **unsigned long**?
(x) O valor mínimo do tipo **unsigned long long**?
(xi) O valor máximo do tipo **unsigned long long**?

(b) Escreva um programa em C que imprima os valores das constantes encontradas no item (a). A saída do programa deve ser algo como:

```
...
O valor mínimo do tipo short e' ...
O valor máximo do tipo short e' ...
...
```

(c) Confronte os resultados apresentados por este programa com aqueles do exercício **EP2.2**.

Sugestões

1. Há algumas pegadinhas nesta questão, pois algumas constantes solicitadas no item (a) não existem; espera-se que, nesses casos, você diga por que a existência dessas constantes simplesmente não faz sentido.

2. Lembre-se de usar os especificadores de formato convenientes nas chamadas da função **printf()**; use `%lld` para **long long** e `%ull` para **unsigned long long**.

3. Não se esqueça de incluir o arquivo `<limits.h>` no seu programa usando **#include**.

EP2.7) O arquivo de cabeçalho `<float.h>`, que faz parte da biblioteca padrão de C, define constantes simbólicas que representam valores limites (máximos e mínimos) dos tipos de ponto-flutuante

elementares da linguagem C, bem como outros valores relativos a números de ponto-flutuante dependentes de implementação. Encontre este arquivo, abra-o e tente responder às questões a seguir.

(a) Que constante simbólica representa:

(i) O valor mínimo do tipo **float**?
(ii) O valor máximo do tipo **float**?
(iii) O valor mínimo do tipo **double**?
(iv) O valor máximo do tipo **double**?
(v) O valor mínimo do tipo **long double**?
(vi) O valor máximo do tipo **long double**?

(b) Escreva um programa em C que imprima, ***em notação científica***, os valores das constantes encontradas no item (a). A saída do programa deve ser algo como:

```
...
O valor mínimo do tipo float e' ...
O valor máximo do tipo short e' ...
...
```

SUGESTÕES

1. Não é fácil encontrar este arquivo nem usando gcc e Linux nem o ambiente DevC++; caso esteja usando gcc e Linux, procure-o num diretório semelhante a `/usr/lib/gcc/i486-linux-gnu/4.1.2/include`; caso esteja usando DevC++, procure-o numa pasta como `C:\Dev-Cpp\lib\gcc\mingw32\3.4.2\include`,.

2. Lembre-se de usar os especificadores de formato convenientes nas chamadas da função **printf()**.

3. Não se esqueça de incluir o arquivo `<float.h>` no seu programa usando **#include**.

EP2.8) Considere o seguinte programa em C:

```
#include <stdio.h>
#define TAXA_DE_DEDUCAO 0.2

int main(void)
{
  char letra1, letra2, letra3; /* Iniciais do usuário */

  double horas, /* entrada - horas trabalhadas  */
         taxa, /* entrada - pagamento por hora */
       íquido, /* saída - pagamento líquido */
        bruto; /* pagamento antes da dedução */

    /* Entrada das iniciais do usuário */
   printf("Introduza suas iniciais > ");
   scanf("%c%c%c", letra1, letra2, letra3);

        /* Entrada do número de horas */
        /* trabalhadas e da taxa de pagamento */
   printf("\nIntroduza o numero de horas trabalhadas > ");
   scanf("%lf", @horas);
   printf("\nIntroduza a taxa de pagamento por hora > ");
   scanf("%lf", @taxa);

    /* Calcula o pagamento bruto */
   bruto = horas * taxaDePagamento;

    /* Calcula o pagamento líquido */
   deducao = TAXA_DE_DEDUCAO * bruto;
   líquido = bruto - TAXA_DE_DEDUCAO;

    /* Apresenta o resultado */
  printf("\n\n\tSeu pagamento líquido e' R$%f", líquido);

   return 0;
}
```

(a) Com que propósito o programa anterior foi escrito (i.e., qual era a ***mais provável*** intenção do programador)?

(b) Quando compilado, este programa dá origem a uma série de mensagens de erro e advertência. Compile o programa

anterior e, para cada mensagem de erro ou advertência, descreva o que origina tal mensagem.

(c) Modifique o programa de tal modo que o mesmo seja compilado sem apresentar nenhuma mensagem de erro ou advertência.

(d) Crie um programa executável a partir do programa modificado no item (c) e teste-o com alguns casos de entrada. Você verá que este programa não produz os resultados esperados pelo programador original. Isto é, o programa apresenta ***erros lógicos***. Identifique as causas destes erros lógicos e apresente soluções para corrigi-los.

EP2.11) Escreva um programa em C que solicite ao usuário que introduza uma nota, cujo valor pode variar entre 0.0 e 10.0, e imprima o conceito referente a esta nota de acordo com a tabela a seguir:

NOTA	CONCEITO
Entre 9.0 (inclusive) e 10.0 (inclusive)	A
Entre 8.0 (inclusive) e 9.0	B
Entre 7.0 (inclusive) e 8.0	C
Entre 6.0 (inclusive) e 7.0	D
Entre 5.0 (inclusive) e 6.0	E
Menor do que 5.0	F

O programa deve ainda apresentar mensagens de erro correspondentes a entradas fora do intervalo de valores permitido. Em qualquer outro aspecto, você pode assumir que as entradas são corretas.

EP2.12) Escreva um programa em C que solicite ao usuário que introduza uma série de valores inteiros positivos e leia estes números até que o usuário introduza o valor 0. Então, o programa deve apresentar o menor, o maior e a média dos valores introduzidos (sem levar em consideração 0).

Caso o usuário introduza um número negativo, o programa deve informá-lo de que o valor não é válido e não deve levá-lo em consideração. Veja um exemplo de interação com o programa (o **negrito** corresponde à entrada do usuário):

```
[Apresentação do programa]
Introduza uma série de números inteiros positivos terminando a
série com zero.
Introduza o próximo número: 5
Introduza o próximo número: -2
-2 não é um valor válido.
Introduza o próximo número: 1
Introduza o próximo número: 6
Introduza o próximo número: 0

Menor valor introduzido: 1
Maior valor introduzido: 6
Média dos valores introduzidos: 4.0
```

EP2.13) Escreva um programa em C que solicite ao usuário que introduza `n` valores inteiros, leia estes números e apresenta o menor, o maior e a média dos valores introduzidos. O valor `n` deve ser o primeiro dado introduzido pelo usuário. Veja um exemplo de interação com o programa (o **negrito** corresponde à entrada do usuário):

```
[Apresentação do programa]
Quantos números você irá introduzir? 3
Introduza o próximo número: 5
Introduza o próximo número: -2
Introduza o próximo número: 0

Menor valor introduzido: -2
Maior valor introduzido: 5
Média dos valores introduzidos: 1.0
```

EP2.14) Escreva um programa em C que brinque de adivinhar números com o usuário. O jogo consiste em gerar um número aleatório entre 1 e 100 e solicitar ao usuário que tente adivinhar o número gerado. Cada partida termina quando o usuário acerta o número, faz o máximo de 5 tentativas ou introduz um número negativo. Números não negativos fora

do intervalo de 1 a 100 não devem ser levados em consideração. Ao final de cada partida, informe ao usuário se ele acertou o número gerado ou qual era este número. Permita que o usuário jogue quantas partidas ele desejar e, antes de encerrar o programa, informe ao usuário quantas partidas foram jogadas e quantas vezes ele acertou o número. Veja um exemplo de interação do programa (o **negrito** corresponde à entrada do usuário):

```
Vou pensar num numero entre 1 e 100.
Tente adivinhar este numero em no máximo 5 tentativas.
Se você digitar um numero negativo, estará desistindo da
partida.

*** Nova partida ***

Apresente seu chute entre 1 e 100: 5
Infelizmente você errou.
Apresente seu chute entre 1 e 100: 16
Infelizmente você errou.
Apresente seu chute entre 1 e 100: 41
Infelizmente você errou.
Apresente seu chute entre 1 e 100: 6
Infelizmente você errou.
Apresente seu chute entre 1 e 100: 3
Partida encerrada. Parabéns!!! Você acertou!!!
Deseja jogar uma nova partida? s

*** Nova partida ***

Apresente seu chute entre 1 e 100: 12
Infelizmente você errou.
Apresente seu chute entre 1 e 100: 0
Numero invalido!!! O numero deve estar entre 1 e 100.
Apresente seu chute entre 1 e 100: 21
...
Partida encerrada. Infelizmente, você não acertou. Meu número
era 36.
Deseja jogar uma nova partida ('n' ou 'N' encerra o jogo)? n

*** Resumo do jogo ***

*** Número de partidas jogadas: 2
*** Número de vezes que você acertou: 1

Obrigado por ter jogado comigo. Digite qualquer tecla para sair
do programa.
```

Sugestões

1. A função **rand()** (`#include <stdlib.h>`) gera números inteiros aleatórios entre 0 e o valor dado pela constante **RAND_MAX**. Para gerar números aleatórios entre `N` e `M (N < M)`, utilize a fórmula: `rand()%(M-N+1) + N`.

2. Para não gerar os mesmos números aleatórios a cada execução do programa, a função **rand()** precisa ser nutrida com um valor arbitrário que não seja o mesmo em duas execuções do programa. Uma técnica comum utilizada para fazer isso é usar o valor retornado pela função **time()** (`#include <time.h>`), que corresponde ao número de segundos decorridos desde 00:00:00 de 1° de janeiro de 1970. Para usar esta função, declare uma variável `tempo` do tipo **time_t** (definido em `<time.h>`) e utilize a função **srand()** (`#include <stdlib.h>`) para alimentar o gerador de números aleatórios, como na instrução: `srand((unsigned) time(&tempo));`. [O *casting* `(unsigned)` é necessário aqui porque a função **time()** retorna um valor do tipo **time_t**.]

3. É muito trabalhoso testar esse programa com números no intervalo entre 1 e 100, porque a probabilidade de acerto é muito pequena. Portanto, teste seu programa com números entre 1 e 5 e, após estar satisfeito com os resultados, retorne aos valores originais.