Capítulo 2

ELEMENTOS DE PROGRAMAÇÃO EM C

2.1 Endereços e Ponteiros

2.1.1 Endereços

Qualquer variável declarada num programa em C possui um endereço em memória que indica o local onde a mesma é armazenada. Freqüentemente, é necessário utilizar num programa o endereço de uma variável, ao invés de seu próprio valor (por exemplo, na função **scanf()**, vista anteriormente, utilizam-se endereços de variáveis como argumentos). O endereço de uma variável pode ser determinado através do uso do operador de endereço **&**. Por exemplo, suponha a existência da seguinte declaração de variável:

```
int  x;
```

então, a expressão:

```
&x
```

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

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

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

2.1.2 Ponteiros

Ponteiros são tipos especiais de variáveis capazes de conter endereços em memória. Um ponteiro que contém um endereço válido em memória é dito ***apontar*** para aquele endereço (daí o nome ***ponteiro***). Um ponteiro pode apontar para um objeto de qualquer tipo armazenado em memória (por exemplo, **int**, **float**, e, mais freqüentemente, outros tipos mais complexos). Assim, podem-se ter ponteiros para inteiros, ponteiros para **float**s, etc. Uma declaração de ponteiros em C tem o seguinte formato:

<tipo apontado> *<variável do tipo ponteiro>;

Por exemplo:

```
int    *ponteiroParaInteiro;
```

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

Variáveis do tipo ponteiro podem ser inicializadas da mesma forma que outros tipos de variáveis[1]. Por exemplo, a segunda declaração a seguir:

[1] Observe que, em algumas linguagens de programação (como Pascal, por exemplo), ponteiros podem apontar apenas para variáveis anônimas alocadas dinamicamente, enquanto que em C, um ponteiro também é permitido assumir o endereço de uma variável normal, conforme visto aqui. Alocação dinâmica de memória em C utilizando ponteiros será vista mais adiante.

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

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

Note também que, apesar de qualquer endereço num dado computador ser representado da mesma maneira (por exemplo, como um inteiro de 32 bits), não existe compatibilidade entre ponteiros para tipos diferentes. Por exemplo, um ponteiro para uma variável **int** é considerado incompatível com um ponteiro para uma variável **float**. Deste modo, a inicialização da variável `ponteiroParaInteiro` no exemplo a seguir seria considerada inválida:

```
float  x;
int    *ponteiroParaInteiro = &x;
```

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

Referenciação de Ponteiros

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

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

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

Além da possibilidade de atribuição de um endereço a um ponteiro, é permitido ainda atribuir um valor ao conteúdo da posição de memória apontada pelo mesmo. Em outras palavras, é permitido (e muito freqüente) atribuir um valor a um ponteiro referenciado. Por exemplo, considerando as declarações do exemplo anterior, a instrução:

```
*ponteiroParaFloat = 1.6;
```

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

```
meuFloat = 1.6;
```

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

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

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

```
*ponteiroParaFloat = 1.6;
```

`*ponteiroParaFloat` recebe o valor de um **float**. Esta interpretação é errônea, entretanto, pois o asterisco na declaração de `ponteiroParaFloat` **não** representa o operador de referenciação. Portanto, na inicialização, o endereço é atribuído ao ponteiro (`ponteiroParaFloat`) e não ao ponteiro referenciado (`*ponteiroParaFloat`). Apenas na instrução de atribuição acima, o operador de referenciação é utilizado.

Os operadores de referenciação (*) e de endereço (&) fazem parte da mesma classe de precedência em que estão todos os outros operadores unários de C. A prioridade destes operadores é a mais alta dentre todos os operadores vistos até aqui e a associatividade deles é da direita para a esquerda.

Para um bom acompanhamento do material a seguir, é importante que esta seção sobre endereços e ponteiros seja bem compreendida. Portanto, releia esta seção e convença-se de que realmente entendeu todos os conceitos e exemplos contidos aqui antes de prosseguir.

2.2 Funções

Talvez a abordagem geral mais utilizada na construção de programas a partir da descrição de um problema é o método de **refinamentos sucessivos** (também conhecido como método ***top-down***). Utilizando este método, a descrição geral do problema é dividida em passos (i.e., subproblemas menos complexos do que o problema original). Então, cada passo é subdividido sucessivamente em subpassos até que cada um dos mesmos contenha operações elementares da linguagem a ser utilizada na construção do programa, ou existam procedimentos ou funções já construídos que realizem a tarefa de forma imediata.

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

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

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

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

Um conselho importante que ajuda a economizar tempo na escrita de programas é que, antes de escrever uma função, verifique se a mesma já existe na biblioteca de C utilizada. Se ela já existir, simplesmente importe-a para seu programa. Pode acontecer ainda que não exista

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

exatamente a função desejada, mas exista uma função que executa uma operação semelhante. Neste último caso, você pode copiar o código-fonte da função para seu programa e adaptá-lo para suas necessidades. (Infelizmente, nem todo compilador vem acompanhado de arquivos-fonte de suas funções de biblioteca.) Outra fonte de *inspiração* para construção de funções são os programas-exemplo que freqüentemente acompanham cada compilador[3].

Funções podem aparecer de três maneiras diferentes num programa:

• Em forma de **definição**, que especifica aquilo que a função realiza, bem como o número e tipos dos dados (argumentos) utilizados pela função.

• Em forma de **chamadas**, que causam a execução da função.

• Em forma de **alusões**, que contêm parte da definição da função e servem para informar ao compilador que a função aludida é definida em outro local (usualmente, num outro arquivo).

Estas formas de referências a funções serão examinadas a seguir.

2.2.1 Definições de Funções

Uma **definição** de função pode ser conceitualmente dividida em duas partes: (1) **cabeçalho** e (2) **corpo da função.** Estas partes serão examinadas a seguir.

2.2.1.1 Cabeçalho de Função

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

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

Formato 1:

<tipo de retorno> <nome da função> (<argumentos>) <declaração de argumentos>

Formato 2:

<tipo de retorno> <nome da função> (<declaração de argumentos>)

Note que a diferença entre os dois formatos é apenas na forma como os argumentos são declarados. O primeiro formato é obsoleto, mas deve ser aceito por qualquer compilador que utilize o padrão ANSI/ISO. O segundo formato é bem mais útil e elegante mas não é aceito por compiladores C muito antigos. Aqui, apenas o formato 2 será utilizado, mas se o compilador que você estiver utilizando não o aceitar, você não terá dificuldades para passar para o formato 1.

Não é obrigatório incluir o tipo de retorno da função, mas se o mesmo não for incluído, o tipo assumido (***default***) será **int**. É sempre bom incluir explicitamente o tipo de retorno de uma função, mesmo que este seja desnecessário (i.e., quando o tipo de retorno é **int**). Quando não se deseja que a função retorne nenhum valor, utiliza-se o tipo especial **void** como tipo de retorno. Quando o tipo de retorno de uma função é **void**, a mesma não deve

[3] Não é instrutivo seguir estes conselhos para os exercícios de programação sugeridos ao longo deste livro.

ser utilizada (chamada) numa expressão ou atribuição (i.e., uma função com tipo de retorno **void** comporta-se como um procedimento em Pascal). O tipo de retorno de uma função pode ser qualquer tipo não-estruturado[4].

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

Exemplos de cabeçalhos de função:

Formato 1:

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

Formato 2:

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

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

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

2.2.1.2 Corpo de Função

O corpo de uma função contém declarações e instruções necessárias para implementar aquilo que a função deve realizar quando a mesma for executada através de uma chamada. O corpo de uma função deve ser envolvido por chaves ({ e }). É importante chamar a atenção para o fato de que ***variáveis declaradas dentro do corpo de uma função não são reconhecidas fora do mesmo.*** Isto é, qualquer objeto declarado dentro de uma função tem validade apenas dentro da função[5].

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

---

[4] Alguns tipos estruturados, a serem vistos posteriormente, podem ser utilizados como tipos de retorno, mas isto não é recomendado, pois compromete o desempenho do programa

[5] Mais geralmente, o corpo de uma função é um **bloco** e objetos declarados dentro de um bloco têm **escopo local** ao mesmo. Estes conceitos serão explorados em profundidade mais adiante.

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

2.2.1.3 Valores de Retorno

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

return <expressão>;

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

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

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

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

Exercício: Escreva o corpo da função do último exemplo em uma única linha de instrução utilizando o operador condicional (**`? :`**).

É importante observar que o tipo resultante da avaliação da expressão de retorno deve ser compatível com o tipo da função declarado no cabeçalho da mesma. Conforme foi visto anteriormente, todos os tipos aritméticos em C são compatíveis. Portanto, isto significa, que, por exemplo, se o tipo de uma função é **int**, uma expressão que resulte em qualquer tipo numérico pode ser utilizada. Note, entretanto, que nos casos onde o tipo declarado de retorno não coincide com o tipo resultante da expressão e for possível uma conversão entre estes tipos, o valor retornado será implicitamente convertido para o tipo de retorno declarado. Por exemplo, na função a seguir:

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

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

Finalmente, o corpo de uma função pode não conter nenhuma instrução **return**. Neste caso, a função será encerrada quando o fecha-chaves (}) representando o final do corpo da função for atingido e nenhum valor válido será retornado.

2.2.2 Parâmetros e Chamadas de Funções

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

```
(void) printf("Esta função retorna o número de argumentos impressos no
meio de saída");
```

Na prática, entretanto, isto raramente é utilizado, como mostra o uso comum de **printf()** e **scanf()** que retornam valores que raramente são utilizados.

Chamadas de funções, cujos tipos de retorno são diferentes de **void**, podem também ser utilizadas como parte de expressões. Numa tal situação, o valor resultante da chamada da função irá substituir a própria chamada da função na expressão. Por exemplo, suponha que uma função `f()` retorne `1` como resultado da execução de uma chamada; então, após a execução da chamada `f()` na expressão:

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

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

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

Passagens de Parâmetros

Argumentos ou **parâmetros** provêem o meio normal de comunicação entre funções em C (e em outras linguagens de programação). Normalmente, não se devem utilizar variáveis e constantes globais para fazer esta comunicação a não ser que haja realmente um bom motivo para assim proceder. Se você tem dúvidas se um determinado valor deve ser representado por um parâmetro ou por uma variável global, é muito mais provável que ele deva ser representado por um parâmetro. Além disso, se uma variável é necessária apenas dentro de uma função, ela deve ser uma variável local à função e, portanto, declarada dentro do corpo da mesma. Não esqueça, ainda, que variáveis locais a uma função não têm validade fora da mesma.

Os argumentos que aparecem numa declaração de função são denominados **parâmetros formais**, enquanto que argumentos utilizados numa chamada de função são denominados **parâmetros reais**. Na chamada de uma função ocorre uma **ligação** (ou **casamento**) entre parâmetros reais e parâmetros formais. A primeira regra a ser seguida para que esta ligação seja bem sucedida é que o número de parâmetros formais seja igual ao número de parâmetros reais. Isto é, se não há parâmetros formais na declaração da função, também não deve haver parâmetros reais na chamada; se houver três parâmetros formais na definição da função, deve haver três parâmetros reais na chamada, e assim por diante. A segunda regra de casamento diz que os respectivos parâmetros devem ser compatíveis e, portanto, conforme já foi visto, um valor de qualquer tipo aritmético pode ser parâmetro real para um parâmetro formal de qualquer tipo aritmético. Em casos onde o tipo de um parâmetro formal e o tipo do parâmetro real correspondente diferem, e uma conversão de tipos é possível, haverá uma conversão implícita do tipo do parâmetro real para o tipo do parâmetro formal. Por exemplo, se uma função é declarada como:

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

a chamada a seguir seria legal:

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

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

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

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

ambas as chamadas a seguir seriam ***ilegais*** :

```
int     i;
long   *ponteiroParaLong;

g(i);
g(ponteiroParaLong);
```

Para precaver-se contra surpresas desagradáveis como resultado de uma chamada de uma função, 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. Este conselho é semelhante àquele apresentado anteriormente com relação à mistura de tipos em expressões.

Em algumas linguagens, existem dois tipos de passagem de parâmetros: (1) por valor e (2) por referência. Na passagem de parâmetros por valor, os parâmetros reais são copiados antes de serem utilizados pela função (ou procedimento), e qualquer modificação sofrida por um parâmetro de valor não é comunicada ao parâmetro real correspondente fora do procedimento. Na passagem por referência, não é feita cópia de parâmetros reais, de modo que qualquer modificação num parâmetro variável, altera o valor do parâmetro real correspondente. Passagem de parâmetros por referência é obrigatória em algumas linguagens para parâmetros de saída ou de entrada/saída (porque qualquer modificação deve ser comunicada ao exterior do procedimento), enquanto que parâmetros apenas de entrada podem ser passados por valor ou por referência. Em C, estritamente falando, existe apenas passagem de parâmetros **por valor**, de modo que nenhum parâmetro real tem seu valor modificado como conseqüência da execução de uma função. Este fato pode parecer estranho à primeira vista e uma questão que surge é: *Como é, então, que uma variável, utilizada como parâmetro numa chamada de função, pode ter seu valor modificado em conseqüência da execução da função?*

A resposta à questão acima é simples: através da utilização de ponteiros e endereços de variáveis é possível modificar o valor de variáveis. O exemplo a seguir ilustra este fato. Considere a função a seguir que tem por finalidade trocar os valores de dois inteiros do tipo **int**:

```
void   Troca(int  *ptrParaInteiro1, int *ptrParaInteiro2)
{
   int   aux = *ptrParaInteiro1; /* Esta variavel é     */
                                 /* local a esta função */
```

```
   *ptrParaInteiro1 = *ptrParaInteiro2;
   *ptrParaInteiro2 = aux;
}
```

Então, dadas as declarações de variáveis a seguir:

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

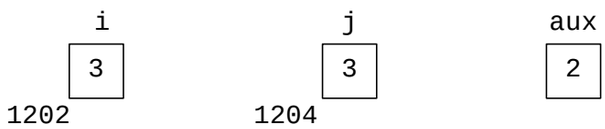
a função `Troca()` poderia ser chamada como:

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

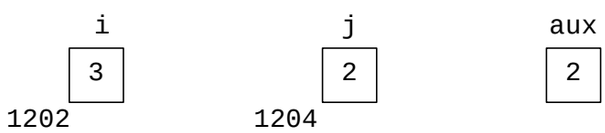
e o efeito após a chamada seria a troca de valores entre `i` e `j` (i.e., `i` passaria a conter 3 e `j` passaria a conter 2). Note que, apesar de `i` e `j` terem seus valores modificados, os parâmetros reais (`&i` e `&j`) mantêm seus valores inalterados, como seria esperado numa passagem de parâmetros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

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

Quando ocorre a chamada da função, os parâmetros passados são os endereços de `i` e `j`; ou seja `1202` e `1204`, respectivamente. Portanto, os parâmetros formais `ptrParaInteiro1` e `ptrParaInteiro2` recebem uma cópia dos endereços `1202` e `1204`, respectivamente. Na primeira instrução da função `Troca()`, que é a inicialização da variável local `aux`, esta variável recebe o valor do conteúdo do endereço apontado por `ptrParaInteiro1`, que é `2`. Na segunda instrução da função, `*ptrParaInteiro1 = *ptrParaInteiro2`, o conteúdo do endereço apontado por `ptrParaInteiro1` é substituído pelo conteúdo do endereço apontado por `ptrParaInteiro2`, de modo que, neste ponto, ter-se-ía a seguinte situação esquemática (o endereço da variável local `aux` não nos interessa aqui):



Finalmente, na terceira e última instrução, o conteúdo do endereço apontado por `ptrParaInteiro2` é substituído pelo valor da variável `aux`, resultando no seguinte esquema:



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

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

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

```
int    *ponteiroParaInt = &i;

Troca(ponteiroParaInt, &j);
```

teria exatamente o mesmo efeito da chamada do exemplo anterior. O perigo em utilizar ponteiros em chamadas de funções (ou em atribuições de referenciação) surge quando um desses ponteiros não foi ainda inicializado com um endereço válido. Por exemplo, suponha novamente que é feita a chamada `Troca(ponteiroParaInt, &j)` do último exemplo, mas agora sem que `ponteiroParaInt` ter sido inicializado com o valor do endereço da variável `i`. Como qualquer variável que não tem um valor explicitamente atribuído, `ponteiroParaInt` contém um valor indeterminado que, neste caso, é um suposto endereço qualquer em memória, que pode, inclusive, nem existir num dado computador (por exemplo, `ponteiroParaInt` pode conter um valor aleatório de `10.000.000` e o computador no qual o programa está sendo executado possui apenas 2MB de memória). Os resultados de uma operação deste tipo são imprevisíveis e perigosos pois se estará modificando aleatoriamente o conteúdo da memória do computador. Este problema aflige não apenas programadores inexperientes, como também aqueles mais experientes (mas descuidados). Para evitar estes acontecimentos discipline-se: sempre que declarar um ponteiro, inicialize-o com um valor conhecido. Se não houver nenhum valor válido para ser atribuído a um ponteiro no instante de sua declaração, inicialize-o com o valor constante inválido **`NULL`**, que é reconhecido pelo compilador como sendo inválido. Desta maneira, se você esquecer de atribuir um endereço válido a um ponteiro e tentar acessar este endereço, o computador indicará uma operação inválida e impedirá que a execução do programa prossiga. Pode até parecer que isto não seja uma boa idéia, mas pior seria permitir que o programa prosseguisse e causasse o mal funcionamento de seu programa, de outros programas, ou até mesmo do próprio computador. Para utilizar a constante **`NULL`**, você deve incluir em seu programa o arquivo de cabeçalho de biblioteca `stddef.h` por meio da diretiva:

```
#include   <stddef.h>
```

2.2.3 Protótipos e Alusões de Funções

Uma **alusão** a uma função contém informações sobre a função que permitem ao compilador reconhecer uma chamada da função como sendo legal. Freqüentemente, a função aludida é definida num arquivo outro que não aquele aonde é feita a alusão. A forma de alusão de uma função é muito parecida com o cabeçalho da própria função, exceto que numa alusão não é necessário especificar nomes de argumentos (embora isto seja recomendável) e pode-se, ainda, iniciar a alusão com a palavra reservada **extern** (também recomendável). Portanto, uma alusão deve ter o seguinte formato[6]:

extern <tipo de retorno> <nome da função> (<tipos dos argumentos>);

Por exemplo, a função `Troca()` do exemplo anterior poderia ter a seguinte alusão:

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

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

```
extern   void   Troca(int  *prt1, int  *ptr2);
```

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

---

[6] Para compiladores que não aceitam o formato 2 de cabeçalho apresentando anteriormente, uma alusão deve conter apenas o tipo de retorno da função, seguida do nome da função, e seguida por abre e fecha parênteses.

2.2.4 Funções com Listas de Argumentos Variáveis

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

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

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

Uma pergunta óbvia neste ponto seria: *Como é possível, na definição de uma função com argumentos variáveis, acessar tais argumentos no corpo da função?* A resposta é que uma tal função deve preencher alguns requisitos. O primeiro requisito que uma função de argumentos variáveis deve preencher é que ela deve possuir pelo menos um argumento fixo (i.e., com identificador e tipo definidos, conforme foi visto anteriormente). O segundo requisito é que os argumentos fixos de uma função (por exemplo, o argumento `formato` na função `printf()` cujo protótipo foi apresentado acima) devem preceder os argumentos variáveis (representados por `...`) no cabeçalho da função. O último requisito é utilizar as macros `va_arg`, `va_end` e `va_start`, definidas em `stdarg.h`, para acessar os argumentos não especificados. A lista de argumentos variáveis é considerada como sendo uma única variável do tipo `va_list`, também definido em `stdarg.h`. A manipulação de argumentos variáveis no corpo de uma função com o uso deste tipo e destas macros será apresentada a seguir.

Na seção de declarações de uma função de argumentos variáveis, deve-se declarar uma variável do tipo `va_list`. Esta variável (que, na realidade, é um ponteiro) conterá (ou melhor, apontará para) *todos* os argumentos passados para a função quando esta for chamada e é utilizada para acessar cada um dos argumentos variáveis da função. Por exemplo, a variável `argumentos` declarada como:

```
va_list     argumentos;
```

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

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

```
void  FArgumentosVariaveis(int x, double y, ...)
{
   va_list  argumentos;

            /* Neste ponto, os argumentos variáveis */
            /* ainda não podem ser acessados      */

   va_start(argumentos, y); /* y é o último argumento fixo */

            /* Agora, os argumentos variáveis já  */
            /* podem ser acessados com  va_arg  */
}
```

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

A macro `va_list` permite o acesso seqüencial a cada um dos argumentos variáveis da função. Esta macro possui dois argumentos: o primeiro argumento é a variável do tipo `va_list` que aponta para a lista de argumentos (a variável `argumentos` no esquema acima) e o segundo argumento é o ***nome do tipo*** do próximo argumento a ser acessado[8]. Esta macro expande para um valor do mesmo tipo do seu segundo argumento e passa a apontar para o próximo argumento variável (se houver mais algum). Isto é, na primeira invocação desta macro, ela retorna o valor do primeiro argumento variável e passa a apontar para o segundo argumento variável; na segunda invocação da macro, ela retorna o valor do segundo argumento variável e passa a apontar para o terceiro argumento variável, e assim por diante[9].

Finalmente, após processar todos os argumentos variáveis[10], para permitir um ***retorno normal*** da função[11], deve-se utilizar a macro `va_end`. Esta macro recebe como único argumento a variável do tipo `va_list` que aponta para a lista de argumentos (a variável `argumentos` no exemplo anterior). Esta macro deve ***sempre*** ser utilizada para encerrar o processamento da lista de argumentos variáveis, caso contrário, o programa poderá ter um comportamento errático indefinido.

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

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

int SomaComArgumentosVariveis(int arg1, ...)
{
   int      soma = arg1;
   va_list  argumentos;
   int      umArgumento;

   va_start(argumentos, arg1);      /* Inicializa lista de  */
                                    /* argumentos variáveis */

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

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

   return soma;
}

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

---

[8] Como os tipos **`char`** (**`signed`** ou **`unsigned`**) e **`short`** (**`signed`** ou **`unsigned`**) são implicitamente promovidos (v. **Seção 1.3.3**) para **`int`** (**`signed`** ou **`unsigned`**), os tipos **`char`** e **`short`** não devem ser utilizados aqui.

[9] Para que os valores dos argumentos variáveis sejam retornados corretamente, é importante que se informe para a esta macro o tipo do argumento (no segundo argumento da macro) com exatidão. Quando todos os argumentos são de um mesmo tipo conhecido em tempo de programação, esta tarefa é simples, mas, quando estes tipos são desconhecidos em tempo de programação, o programador deve elaborar meios para descobrir os tipos corretos dos argumentos variáveis. Por exemplo, o tipo de cada argumento variável da função **`printf`**`()` é descoberto através de uma busca pelo especificador de formato no primeiro argumento (string de formatação) correspondente ao dado argumento variável.

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

[11] Entenda-se por ***retorno normal da função*** o correto desmonte da pilha de execução da função com o subseqüente retorno ao ponto de chamada da mesma.

```
}
```

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

2.3 Recursividade

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

- **Caso não-recursivo**, que estabelece uma **condição de parada** da recursividade e sem o qual a recursividade será infinita. Esta parte da definição da função não deve fazer referência à própria função.

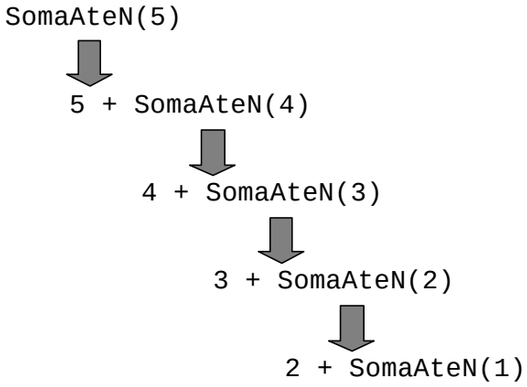
- **Caso recursivo**, no qual a função chama a si mesma. O programador deve garantir que uma das chamadas recursivas atinja eventualmente a condição de parada.

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

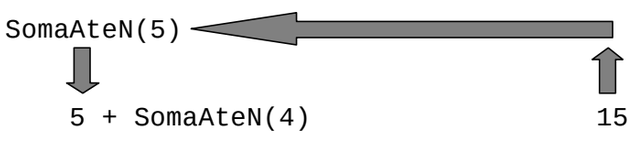
A função do exemplo a seguir ilustra o processo de recursividade em C:

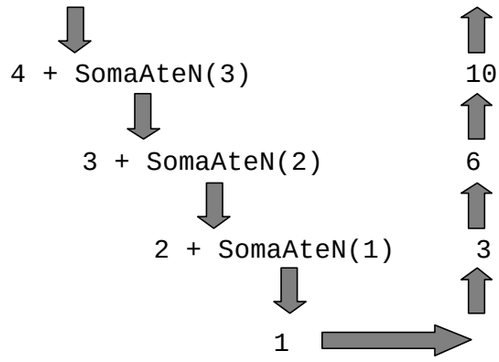
```
int   SomaAteN(int   n)
{
   if  (n <= 1)
      return  n;                    /* Condição de parada */
   else
      return  (n + SomaAteN(n - 1));      /* Caso recursivo */
}
```

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



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





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

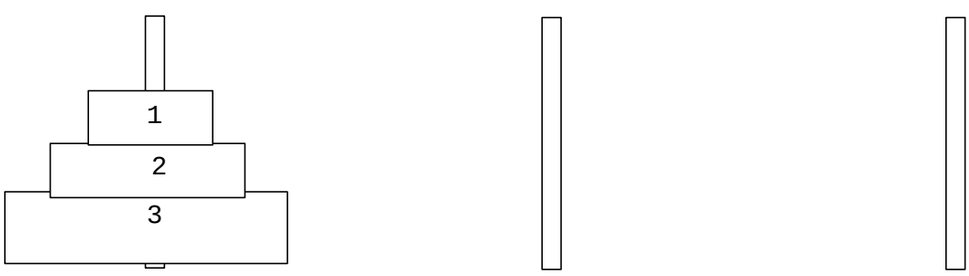
```
int   SomaAteN( int   n)
{
   if  (n < 1){
       printf("Erro: Esta funcao nao aceita numeros menores do que 1.");
       return  0;
   }
   else  if  (n == 1)
       return  n;                          /* Condição de parada */
   else
       return  (n + SomaAteN(n - 1));    /* Caso recursivo */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

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

Exercício: Escreva uma versão iterativa da função `SomaAteN()` utilizando um laço de repetição **for** no corpo da função.

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

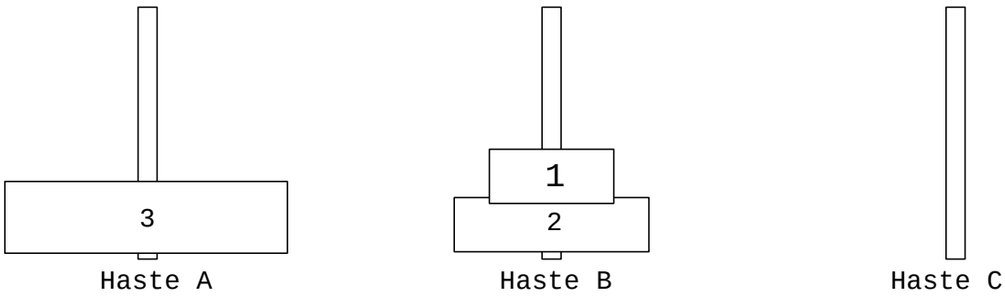
Inicialmente, no problema das Torres de Hanói, existem três hastes e um determinado número de discos de diâmetros diferentes empilhados uns sobre os outros numa das hastes. O que o problema requer é que os discos sejam movidos de uma haste para outra obedecendo duas restrições. Uma restrição do problema é que *nenhum disco pode ser colocado sobre um outro disco de diâmetro menor*. A outra restrição é que *apenas o disco do topo de uma haste pode ser movido* num dado instante (ou, em outras palavras, para mover-se um dado disco, deve-se primeiro mover os discos que estão sobre ele). A figura a seguir ilustra a condição inicial do problema das Torres de Hanói para três discos (os discos foram numerados para facilitar referências aos mesmos):



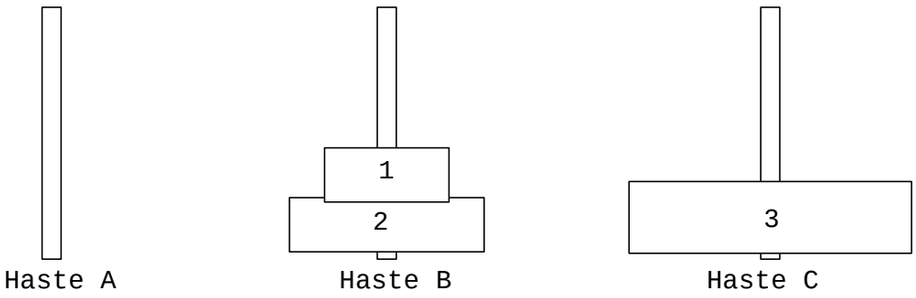
```
Haste A                    Haste B                    Haste C
```

Considerando o diagrama anterior, o problema então consiste em deslocar os três discos na `Haste A` (denominada de ***haste de origem***) para a `Haste C` (denominada de ***haste de destino***), utilizando a `Haste B` como haste auxiliar. Por enquanto, não se preocupe com entrada e saída de dados do programa a ser desenvolvido e concentre-se na solução do problema para o caso geral aonde existem `n` discos.

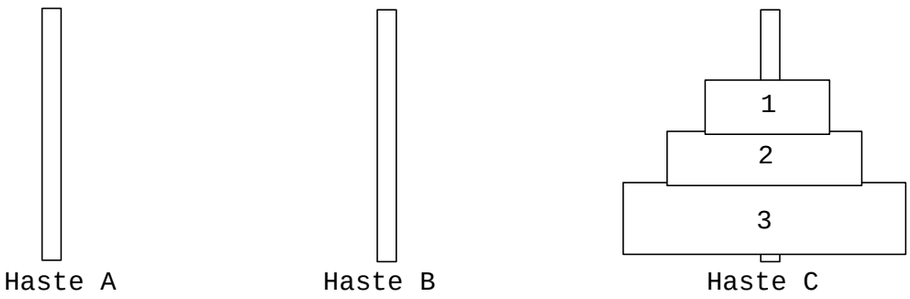
Para começar, suponha que a solução do problema para `n - 1` discos seja conhecida. Então, se for possível descrever a solução para `n` discos em termos da solução para `n - 1` discos, o problema será facilmente resolvido. De fato, isto é verdade porque, movendo-se `n - 1` discos para a `Haste B` (auxiliar)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



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



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



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

> *1. Se n = 1, mova o único disco da haste A para a haste C e pare.*
> *2. Mova os n - 1 discos do topo da haste A para a haste B utilizando C como auxiliar.*
> *3. Mova o disco remanescente na haste A para a haste C.*
> *4. Mova os n - 1 discos da haste B para a haste C utilizando A como auxiliar.*

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

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

Mova o disco D da haste H1 para a haste H2

Também, serão adotadas as seguintes convenções: (1) a haste de origem é denominada A, a haste de destino é denominada C, e a haste auxiliar é denominada B; (2) os discos são numerados de 1 a n a partir do disco do topo na haste de origem na situação inicial (em outras palavras, os discos são numerados de 1 a n do menor para o maior). Um programa completo para o problema das Torres de Hanói é apresentado a seguir:

```
#include <stdio.h>

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

main()
{
   int       numeroDeDiscos;

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

   TorresDeHanoi(numeroDeDiscos, 'A', 'C', 'B');
}

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

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

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

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

```
Introduza o numero de discos: 4

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

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

Exercício: Modifique o programa acima considerando que os discos são numerados do maior para o menor.

Note que, no programa acima, a alusão:

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

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

É interessante observar ainda que a função `TorresDeHanoi()` poderia se tornar mais geral se as apresentações de resultados, representadas pelas chamadas da função **printf()**, fossem colocadas separadamente em outra função. Esta função, que será aqui denominada de `Mova()`, seria responsável pela apresentação dos resultados. A função `Mova()` deve ter como argumentos os dados necessários para a apresentação destes resultados, que são: o número do disco sendo movido, e as hastes de origem e destino do movimento. A grande vantagem desta mudança é que, com ela, a função `TorresDeHanoi()` torna-se independente das suposições sobre a apresentação dos resultados. Se, posteriormente, fosse decidido que a saída do programa seria efetuada com animação gráfica (ou mesmo sonora), ao invés de textual, toda a mudança necessária seria confinada à função `Mova()`.

Exercício: Implemente a função `Mova()` e substitua as chamadas da função **printf()** dentro da função `TorresDeHanoi()` por chamadas da função `Mova()`.

Para concluir, observe que, da mesma forma que na elaboração da função `SomaAteN()`, a solução obtida aqui para o problema das Torres de Hanói foi desenvolvida identificando-se um caso trivial (i.e., quando o número de discos é igual a 1) e uma solução para o caso geral (i.e., para `n` discos) em termos de uma caso mais simples (i.e., para `n - 1` discos). Em termos de estratégia de resolução de problemas, entretanto, existe uma diferença fundamental entre os dois problemas aqui descritos. O problema de encontrar a soma dos

números entre 1 e n pode facilmente ser resolvido sem o uso de recursividade, e a solução iterativa não apenas é mais clara como também é mais eficiente em termos de recursos computacionais. O problema das Torres de Hanói, por outro lado, não possui solução não-recursiva trivial, e, portanto, representa uma situação prática aonde o uso de recursividade é mandatário.

Cadeias Recursivas

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

O perigo de se ter recursividade infinita é maior em cadeias recursivas do que com funções que são diretamente recursivas. Também, em termos de estilo, cadeias recursivas não são fáceis de serem identificadas como tais, pois examinando-se apenas uma das funções envolvidas não dá para perceber que a mesma chama indiretamente a si mesma.

2.4 Classes de Armazenamento

Em projetos de programação de grande porte, nos quais os programas são divididos em múltiplos arquivos, é importante que se definam os locais aonde identificadores têm validade. Em C, pode-se definir se uma variável deve ser compartilhada e aonde a mesma tem validade através da especificação de seu **escopo**. Mais genericamente, o escopo de um identificador é a região do programa que o contém aonde a declaração do identificador é ativa. Em outras palavras, o escopo de um identificador é o local aonde o identificador faz sentido.

Outra propriedade de uma variável é a sua **duração**, que descreve o tempo de vida do espaço em memória reservado para a variável. Variáveis de **duração fixa** retêm seus valores mesmo após a saída de seu escopo. Por outro lado, variáveis de **duração automática** deixam de existir ao final da execução de seu escopo.

Em conjunto, o escopo e a duração de uma variável são denominados de a **classe de armazenamento** da mesma. As várias classes de armazenamento permitidas para uma variável serão discutidos a seguir, mas antes de prosseguir, é importante definir melhor o conceito de **bloco**.

Um bloco é qualquer conjunto válido de instruções e declarações colocados entre abre- e fecha-chaves. Isto inclui instruções compostas bem como corpos de funções. Um bloco pode ser colocado em qualquer local de um programa aonde uma única instrução é permitida. Objetos declarados dentro de um bloco são conhecidos como **locais** ao bloco. Por exemplo, a variável `x` no trecho de programa a seguir é uma variável local ao bloco do corpo do laço **while**:

```
while (1) {
   int  x = 0;
        .
        :
}
```

Blocos podem ainda ser arbitrariamente aninhados dentro de outros blocos, embora o aninhamento de mais de um nível não seja recomendado pois prejudica a legibilidade do programa.

2.4.1 Duração Fixa e Duração Automática de Variáveis

Uma variável de **duração fixa** (ou **estática**) é uma variável cujo tempo de vida é todo o tempo de execução do programa. Em outras palavras, uma variável de duração fixa tem memória alocada para si no início da execução do programa e permanece associada a esta mesma posição de memória até o final da execução do programa. Por outro lado, uma variável de **duração automática** tem espaço em memória alocado para si apenas quando o fluxo de execução do programa entra no escopo da variável. Quando o escopo de uma variável de duração automática é abandonado, a variável deixa de existir e o espaço em memória ocupado pela mesma é liberado. Portanto, é possível que uma variável de duração automática ocupe diferentes posições em memória cada vez que o fluxo de execução do programa entra no escopo da mesma. Consequentemente, não existe nenhuma garantia de que uma variável de duração automática assuma o mesmo valor entre uma saída e uma subseqüente reentrada de seu escopo.

Pode-se especificar se uma variável terá duração fixa ou automática por meio de um **especificador de classe de armazenamento**. Por *default*, variáveis locais a um bloco são de duração automática, mas pode-se indicar isto explicitamente através do especificador **auto**. Como variáveis de duração automática aparecem apenas dentro de blocos e são consideradas automáticas por *default*, o especificador **auto** raramente é utilizado porque ele é sempre redundante. Por outro lado, pode-se informar ao compilador que uma variável deve ser tratada como sendo de duração fixa por meio do especificador **static**.

Um uso comum de variáveis locais de duração fixa é o de registrar as vezes que uma função é executada e modificar o comportamento da função de acordo com este registro. Considere, por exemplo, a seguinte função:

```
void   MinhaFuncaoDeComportamentoVariavel( void )
{
    static  unsigned  char  indicadorDeComportamento = 0;

    if ( indicadorDeComportamento == 0 ){
       /* Especifique aqui o que a função deve fazer */
      /* quando indicadorDeComportamento  é 0       */
       indicadorDeComportamento = 1;    /* Modifique o valor de        */
                           /* indicadorDeComportamento de modo que da */
    }                       /* próxima vez a parte else será executada */
    else {
       /* Especifique aqui o que a função deve fazer */
      /* quando indicadorDeComportamento  é 1       */
       indicadorDeComportamento = 0;    /* Modifique o valor de          */
                          /* indicadorDeComportamento de modo que     */
    }                       /* da próxima vez a parte if será executada */
}
```

A função `MinhaFuncaoDeComportamentoVariavel()` do último exemplo alterna entre a execução da parte **if** e da parte **else** em seu corpo a cada chamada. Este comportamento é regido pela variável `indicadorDeComportamento` que tem duração fixa. Se esta variável tivesse duração automática, esta mudança de comportamento a cada chamada não seria possível. Algumas linguagens de programação não permitem variáveis locais de duração fixa; portanto, numa tal linguagem, num caso como o da função `MinhaFuncaoDeComportamentoVariavel()`, a variável indicadora de comportamento teria de ser global. Um exemplo prático semelhante a este último exemplo seria o de uma função para formatação de cabeçalhos de páginas. Neste caso, a função seria chamada alternadamente para imprimir ora uma página par, ora uma página ímpar[12]. Ainda neste exemplo específico, uma variável como `indicadorDeComportamento` poderia ser utilizada para indicar se uma dada página é par ou ímpar e modificar o comportamento da função adequadamente.

Inicialização de Variáveis

[12] Note que cabeçalhos e rodapés de páginas impressas são apresentados normalmente de formas diferentes de acordo com o fato de a página ser par ou ímpar.

Existe uma diferença fundamental entre variáveis fixas e automáticas com relação à inicialização das mesmas. Variáveis de duração fixa são inicializadas apenas uma vez, enquanto que uma variável de duração automática é inicializada sempre que o fluxo do programa entra no escopo da mesma. Considere, por exemplo, a seguinte função:

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

    i++;
    j++;

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

Chamadas sucessivas da função `Incrementa()` produziriam o seguinte no meio de saída:

```
Valor de i = 2          Valor de j = 2
Valor de i = 2          Valor de j = 3
Valor de i = 2          Valor de j = 4
      .                       .
      .                       .
      .                       .
```

Os resultados de saída acima são conseqüências das definições de `i` e `j` na função `Incrementa()`. A variável `i` é automática (por *default*) e, portanto, é alocada e inicializada cada vez que a função é chamada. Por outro lado, a variável `j` tem duração fixa e é alocada e inicializada apenas uma vez[13]. A cada chamada da função `Incrementa()`, é utilizado o valor atual da variável `j` que é retido entre uma chamada e outra.

Uma importante diferença entre variáveis de duração fixa e automática é que variáveis de duração fixa são inicializadas por *default* (i.e., na ausência de inicialização explícita) com o valor zero. Por outro lado, variáveis de duração automática nunca são automaticamente inicializadas pelo compilador. Isto é, variáveis de duração automática que não são explicitamente inicializadas recebem, quando são alocadas, um valor aleatório (que é o conteúdo sem significado que se encontra na respectiva posição de memória alocada).

Exercício: Modifique a função `Incrementa()` apresentada no último exemplo de modo que as variáveis não sejam mais inicializadas, escreva um programa que chame várias vezes esta função modificada e verifique qual é a saída resultante.

Outra diferença entre variáveis de duração fixa e automática é que, no caso de variáveis de duração fixa, não são permitidas inicializações envolvendo expressões contendo quaisquer tipos de variáveis. Esta exigência não se aplica ao caso de variáveis de duração automática; neste caso, pode-se incluir variáveis numa expressão de inicialização, desde que estas variáveis já tenham sido previamente declaradas. Por exemplo:

```
int   i = 1;
int   j = 2*i + 7; /* Legal pois j é de duração automática e      */
                   /* i já é conhecida neste ponto               */
static  int  k = i; /* ILEGAL porque k é de duração fixa e sua    */
                    /* inicialização não pode envolver variáveis  */
```

É importante notar ainda que, apesar de existir durante todo o tempo de execução de um programa, o escopo de uma variável estática não é alterado pelo fato de a mesma ser estática. Por exemplo, o escopo das variáveis `i` e `j` declaradas no corpo da função `Incrementa()` apresentada anteriormente é exatamente o corpo daquela função, não importando o fato de uma ser automática e a outra ser fixa.

[13] Isso ocorre quando a execução do programa é iniciada, e não quando a função `Incrementa()` é chamada.

2.4.2 Escopo

Em C, escopos podem ser classificados em quatro tipos diferentes:

• **Escopo de Programa**. Uma variável com este tipo de escopo é ativa em todos os arquivos e blocos que compõem o programa. Variáveis com escopo de programa são conhecidas como **variáveis globais**. Qualquer variável declarada fora de funções tem escopo de programa, a não ser que ela seja precedida pela palavra **static** (ver abaixo).

• **Escopo de Arquivo**. Uma variável com este tipo de escopo tem validade em todos os blocos do arquivo na qual ela é declarada e a partir do ponto de declaração da variável. Para ter escopo de arquivo, uma variável deve ser declarada fora de qualquer função e ter sua declaração precedida pela palavra reservada **static**. É importante notar que usada neste contexto, a palavra **static** ***não tem o mesmo significado*** visto na seção anterior. Isto é, aqui **static** não se refere a duração de variáveis como antes, mas sim a definição de escopo. Em qualquer circunstância, uma variável declarada fora de qualquer função tem duração fixa (quer ela venha acompanhada de **static** ou não). O sentido de **static** aqui é o de delimitar o escopo de uma variável ao arquivo na qual a mesma é declarada; sem ser qualificada com **static**, a variável será tratada como uma variável global. Este mesmo sentido de **static** é utilizado para delimitar escopos de funções, como será visto mais adiante.

Variáveis com escopo de arquivo são úteis quando existem várias funções num arquivo que utilizam uma mesma variável. Ao invés de passar esta variável como parâmetro para as várias funções do arquivo, utiliza-se a variável com escopo de arquivo e, assim, todas as funções do arquivo poderão utilizá-la. Esta variável não pode entretanto ser acessada por funções em outros arquivos.

• **Escopo de Função**. Um identificador com este tipo de escopo tem validade do início ao final da função na qual é definida. Apenas rótulos utilizados em conjunção com instruções **goto** têm este tipo de escopo. Rótulos devem ser únicos dentro de uma função e permanecem ativos do início ao final da mesma.

• **Escopo de Bloco**. Uma variável com este tipo de escopo tem validade do início ao final do bloco no qual é definida. Variáveis declaradas dentro do corpo de uma função ou parâmetros de funções têm este tipo de escopo. Uma conseqüência disto é que não se pode ter numa mesma função um parâmetro e uma variável local com o mesmo nome.

Uma variável com escopo de bloco não pode ser acessada fora do seu bloco. Esta capacidade de limitar o escopo de uma variável é, na realidade, uma vantagem do ponto de vista de legibilidade e manutenção de um programa. O programador pode escrever porções do programa sem ter que se preocupar com conflito com variáveis declaradas em outras partes do programa, e o leitor do programa saberá que uma dada variável é confinada numa dada região.

Em geral, o escopo de uma variável é determinado pela localização de sua declaração. Variáveis declaradas dentro de um bloco têm escopo de bloco; variáveis declaradas fora de blocos têm escopo de arquivo se estiverem acompanhadas do especificador **static**, ou escopo de programa quando não estiverem acompanhadas de **static**.

Podem-se imaginar os diversos tipos de escopo como parte de uma hierarquia, com o escopo de programa ocupando a mais alta posição e o escopo de bloco ocupando a mais baixa posição, conforme mostra a **Figura 3**.

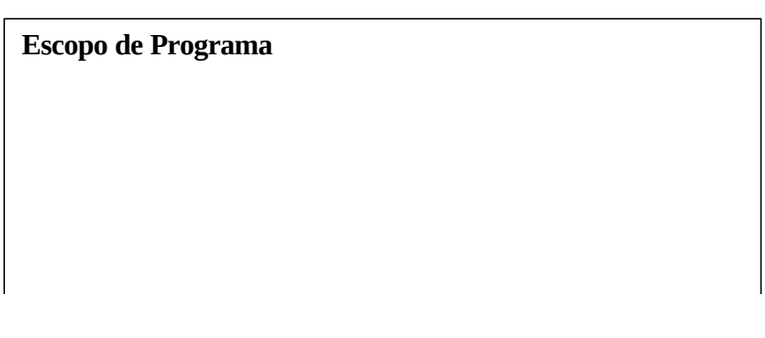


Figura 1: Hierarquia de Escopos

Examinando-se o diagrama acima, pode-se concluir, por exemplo, que uma variável com escopo de programa também tem escopo de arquivo, de função e de bloco, enquanto que uma variável com escopo de bloco tem validade apenas neste tipo de escopo. O exemplo a seguir ilustra os vários tipos de escopo:

```
int          i;          /* i tem escopo de programa e duração fixa */
static  int  j;          /* j tem escopo de arquivo e duração fixa */

void f( int  k ) /* f tem escopo de programa; k tem escopo de bloco */
{
   int  m;        /* m tem escopo de bloco e duração automática */
   static  short  n;   /* n tem scopo de bloco e duração fixa */

   istoEhUmRotulo :     /* Escopo de função */
      .
      :
}
```

Como outras linguagens, C permite o uso de identificadores iguais para variáveis em escopos diferentes. Neste caso, duas funções diferentes podem utilizar o mesmo identificador x sem perigo de ambigüidade, como no exemplo a seguir:

```
void  f1( void )
{
    int   x;
     .
     :
}


void  f2( void )
{
    float   x;
      .
      :
}
```

Menos evidente é fato de também ser permitido o uso de identificadores iguais mesmo em escopos que se sobrepõem. Em caso de conflito, a declaração mais próxima do ponto de conflito é utilizada. Por exemplo:

```
float  x = 2.5;      /* x tem escopo de programa (i.e., é global ) */

void  f ( void )
{
    int  x  = 1;   /* Esta é a declaração de x que vale neste bloco */
    printf("Valor de x = %f", x); /* Aqui será impresso o valor 1.0 */
                                  /* e não 2.5                      */
}
```

No último exemplo, o escopo da variável `x` declarada como **float** abrange o bloco da função `f`, e esta variável poderia ser utilizada dentro de `f` se não fosse o fato de uma nova variável `x` ser declarada como **int** no bloco de `f`. Com isso, a variável `x` declarada como **int** será aquela considerada dentro do bloco de `f`; i.e., a variável `x` declarada como **float** deixa de ser reconhecida dentro deste bloco.

2.4.3 Variáveis Globais

Em geral, deve-se evitar o uso de variáveis globais tanto quanto possível, pois isto acarreta em programas mais complexos e difíceis de serem mantidos. O uso da palavra **static** para limitar o escopo de uma variável a um arquivo facilita a leitura do programa pois, quando se deseja examinar todas as alterações sofridas pela variável, tudo que se tem a fazer é concentrar-se no arquivo que a contém. Na ausência da palavra **static**, no entanto, o leitor do programa deve assumir a pior situação e procurar por modificações da variável em todos os arquivos que compõem o programa.

Variáveis globais também são uma fonte de conflito potencial entre os módulos componentes de um programa desenvolvidos por programadores diferentes. Por exemplo, dois programadores trabalhando em partes distintas do programa podem, por coincidência, escolher o mesmo nome para representar diferentes variáveis globais (isto é pouco provável se o projeto for bem gerenciado, mas não deixa de ser possível).

Quando inevitável, o uso de variáveis globais deve seguir alguma convenção de nomes que as tornem distintas das demais. Uma convenção freqüentemente utilizada é iniciar o nome de cada variável global com a letra `g` (por exemplo, `gMinhaVariavelGlobal`)[14].

A única vantagem advinda do uso de variáveis globais é que o código resultante é usualmente mais eficiente. Em muitos casos, entretanto, este ganho em eficiência vem às custas de perda em termos de manutenibilidade do programa.

Definições e Alusões

Em C, variáveis globais aprecem sob a forma 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 (como usual para outros tipos de variáveis). Uma alusão a uma variável global, que pode aparecer em vários arquivos que fazem parte do programa, é similar a uma definição, mas não aloca memória para a variável. Em vez disto, uma alusão serve para informar ao compilador que a variável aludida é uma variável global definida em outro arquivo. Em outras palavras, alusões a variáveis globais têm basicamente o mesmo significado de alusões a funções vistas anteriormente.

Sempre que for necessário utilizar uma variável global definida num arquivo diferente daquele no qual se está trabalhando, deve-se fazer uso de uma alusão à variável. Para fazer uma alusão a uma variável, utiliza-se a palavra reservada **extern** seguida do tipo da variável e do nome da variável. Por exemplo:

```
extern  long  gMinhaGlobal;
```

As regras para definições e alusões de variáveis globais são pouco padronizadas e, portanto, o programador deve adotar consistentemente a seguinte estratégia em seus programas:

- Para **definir** uma variável global, ***omita*** a palavra **extern** e inclua uma inicialização.

[14] **Cuidado**: Alguns compiladores reconhecem apenas os seis primeiros caracteres do nome de uma variável global, embora não impeçam que o programador utilize nomes mais longos. Nestes compiladores, não faz diferença se o programador utilize como nomes, por exemplo, `gMinhaVariavelGlobal` ou `gMinhaVariavel`; em ambos os casos o nome será considerado como `gMinha`.

• Para **aludir** a uma variável global, ***omita*** qualquer inicialização e inclua a palavra **extern**.

Esta estratégia pode ser resumida na **Tabela 1**, apresentada a seguir para facilidade de referência.

	DEFINIÇÃO	**ALUSÃO**
extern	Não	Sim
Inicialização	Sim	Não

Tabela 1: Definição x Alusão de Variáveis Globais

2.4.4 Especificadores de Registradores

Todo computador possui um pequeno número de registradores que são pequenas unidades de armazenamento dentro da CPU. Tipicamente, um registrador é capaz de conter apenas dois ou quatro bytes. Um computador utiliza seus registradores para efetuar operações aritméticas. Por exemplo, a instrução seguinte:

```
k = n + m;
```

pode fazer com que os valores de `n` e `m` sejam carregados em dois registradores. Então, o computador adiciona os valores contidos nestes dois registradores e armazena o valor resultante na posição de memória representada por `k`.

Geralmente, operações envolvendo registradores são muito mais rápidas do que operações envolvendo posições de memória. Infelizmente, o número de registradores em qualquer computador é muito limitado se comparado com a capacidade de memória do computador. Também, muito freqüentemente, o número de variáveis ativas num programa é muito maior do que o número de registradores disponíveis. Portanto, normalmente, não é possível manter todas as variáveis de um programa em registradores, como seria ideal. Bons compiladores são dotados de estratégias para decidir que variáveis manter em registradores de forma a minimizar o acesso à memória principal do computador.

A palavra reservada **register** serve para o programador ***sugerir*** ao compilador as variáveis que devem ser armazenadas em registradores. Entretanto, o compilador tem liberdade para aceitar ou não esta sugestão. Os compiladores comportam-se de maneiras bastante variadas neste aspecto. Por exemplo, em alguns compiladores, existe uma opção, que o programador pode escolher numa caixa de diálogo, que especifica se o compilador deve tentar seguir a sugestão do programador em primeiro lugar ou usar sua própria estratégia de uso de registradores. Outros compiladores podem rejeitar categoricamente qualquer sugestão do programador e utilizar suas próprias estratégias de utilização de registradores.

É interessante notar que uma variável declarada com **register** pode nunca ter um endereço em memória (i.e., pode ser que ela seja mantida num registrador durante todo seu tempo de vida). Portanto, como registradores não possuem endereço, não se pode fazer referência ao endereço de uma variável declarada com o especificador **register**. Isto resultaria em erro de compilação, independentemente do fato de a variável ser realmente atribuída a um registrador ou não.

O especificador **register** pode ser utilizada apenas com *argumentos de funções* ou *variáveis locais a funções*, e deve ser usada para variáveis *automáticas* que são acessadas com freqüência. Um caso típico de uso do especificador **register** é aquele de variáveis utilizadas como contadores em laços **for**. Por exemplo,

```
register  int  i;

for  (i = 0; i <= 10000; i++){
```

```
      .
      :
}
```

Em princípio, não existe nenhum limite quanto ao número de variáveis que podem ser declaradas com a palavra **register**. Na prática, se houver mais variáveis declaradas com **register** do que o número de registradores disponíveis, e o compilador aceitar as sugestões de alocação de registradores, ele irá considerar apenas as primeiras declarações deste tipo até que o número de registradores disponíveis seja atingido. Deve-se observar também que, conforme foi afirmado antes, a palavra **register** pode também ser utilizada em declarações de parâmetros formais de funções. Por exemplo, a declaração de parâmetro da função a seguir é perfeitamente legal:

```
void   f( register  int  *p )
{
      .
      :
}
```

2.4.5 Modificadores de Classes de Armazenamento

Além dos especificadores de classes de armazenamento **auto**, **static**, **extern**, e **register** descritos acima, existem dois **modificadores** de classes de armazenamento, denominados **const** e **volatile**, que serão descritos a seguir.

*2.4.5.1 Modificador **const***

O modificador **const**, quando aplicado na declaração de uma variável, especifica que esta variável não pode ser modificada após sua inicialização. Por exemplo, após a definição:

```
const  long  int  minhaVarConstante = 0L;
```

não seria mais permitida a modificação da variável `minhaVarConstante`. Pode parecer estranho à primeira vista o fato de se denominar um objeto deste tipo como uma ***variável*** de valor constante e não simplesmente uma constante. Mas, o fato é que uma variável declarada com o modificador **const** não é exatamente a mesma coisa que uma declaração de constante simbólica vista anteriormente. Por exemplo, na declaração de constante a seguir:

```
#define   MINHA_CONSTANTE   0L;
```

`MINHA_CONSTANTE` difere de `minhaVarConstante` basicamente porque `MINHA_CONSTANTE` não tem espaço em memória alocado para si, enquanto que `minhaVarConstante` terá espaço alocado. Em conseqüência disto, não se pode, por exemplo, atribuir o endereço de uma constante simbólica a um ponteiro, mas pode-se fazer isto com variáveis constantes, como o exemplo a seguir mostra:

```
long  int  *ptr1 = &minhaVarConstante; /* Legal */
long  int  *ptr2 = &MINHA_CONSTANTE;   /* ILEGAL, pois constantes          */
                                       /* simbólicas não possuem  endereço */
```

É interessante notar ainda que um dado compilador pode não garantir que uma variável constante não seja modificada ***indiretamente***. Por exemplo, considerando a definição de `ptr1` acima, muitos compiladores permitem que a variável `minhaVarConstante` seja modificada através de uma instrução como:

```
*ptr1 = 1L;
```

Um aspecto interessante do uso da palavra **const** é que, numa declaração de ponteiro, ela pode aparecer precedida pelo símbolo * ou não. Nos dois casos, os significados das declarações são diferentes. Por exemplo, na segunda declaração a seguir[15]:

```
int           x;
int   *const  ponteiroConstante = &x;
```

`ponteiroConstante` é declarado como sendo um ponteiro que deve apontar ***sempre*** para a variável `x` (i.e., o valor do endereço para onde o ponteiro aponta não muda); o valor do conteúdo para onde o ponteiro aponta pode ser modificado usando o ponteiro (ou a própria variável `x`, obviamente). Por outro lado, na segunda declaração a seguir[16]:

```
int           x;
int   const   *ponteiroParaConstante = &x;
```

`ponteiroParaConstante` é declarado como sendo um ponteiro para uma variável que não pode ser modificada ***através deste ponteiro*** (mas pode ser modificada usando a própria variável, obviamente). Neste caso, o valor do ponteiro em si pode ser modificado de modo que o ponteiro possa apontar para outras variáveis do tipo **int** (ou **const int**).

O principal propósito de **const** é assegurar que dados que não devem ser modificados não serão realmente modificados. Em particular, isto é útil quando ponteiros são passados para funções. A declaração de um argumento com a palavra **const** garante que o objeto apontado por aquele ponteiro não será modificado pela função. Por exemplo, na declaração de função abaixo:

```
void CopiaString(const char *origem, char *destino)
{
      .
      :
}
```

a declaração do argumento `origem` como sendo um ponteiro para um objeto constante garante que a função `CopiaString` não irá modificar o objeto apontado[17].

*2.4.5.2 Modificador **volatile***

O modificador **volatile** é utilizado para informar ao compilador que a variável precedida por esta palavra pode ser modificada no programa de uma maneira desconhecida pelo compilador. Em termos práticos, o modificador **volatile** informa ao compilador que ele não deve utilizar seu conhecimento aparente sobre a variável para otimizar trechos de programa que a envolvem. Por exemplo, suponha que `TECLADO` representa um registrador de dispositivo do teclado do computador (i.e., um endereço especial em memória aonde será colocado cada valor digitado pelo usuário), e que se tenha o seguinte trecho de programa:

```
extern  char  TECLADO;
int     i;
char    c;

for (i = 1; i <= 10; i++){
   c = TECLADO;
   putchar(c);
}
```

---

[15] Não é obrigatório inicializar a variável `ponteiroConstante`, mas, na prática, esta é a única maneira de atribuir um valor a esta variável.

[16] Essa declaração é eqüivalente à declaração: `const int *ponteiroParaConstante;`

[17] Você pode ficar intrigado com o fato de que se um objeto não deve ser modificado é suficiente que não se acesse o mesmo utilizando um ponteiro, pois passagem de parâmetros em C é sempre por valor. A justificativa é que, como será visto mais adiante, alguns tipos de parâmetros devem sempre ser passados como ponteiros, mesmo que eles não sofram mudança dentro da função.

O propósito do laço **for** acima é ler 10 caracteres do registrador de dispositivo `TECLADO` e passá-los para a função **putchar**(). Acontece que como não há forma de o compilador saber que `TECLADO` representa uma posição em memória que é atualizada continuamente, ele pode considerar a instrução `c = TECLADO` como sendo invariante dentro do laço. Por causa disso, ele poderá compilar este trecho de programa como se esta instrução tivesse sido colocada antes do laço. Alguns compiladores assim procedem com o objetivo de otimizar o código resultante do programa. Em outras palavras, estes compiladores consideram que, como `c = TECLADO` atribui sempre o mesmo valor a `c`, o laço seria otimizado se esta instrução fosse colocada fora do laço. Portanto, num tal compilador, o laço **for** seria compilado como se tivesse sido escrito como:

```
c = TECLADO;
for (i = 1; i <= 10; i++){
   putchar(c);
}
```

e, em conseqüência disso, apenas um caractere seria lido do registrador `TECLADO` e impresso dez vezes pela função **putchar()**. Para impedir que o compilador proceda desta maneira, o programador deve informá-lo para não otimizar nenhuma expressão envolvendo a variável `TECLADO` através da declaração:

```
extern volatile char TECLADO;
```

2.5 Programas Distribuídos em Múltiplos Arquivos

2.5.1 Funções Globais e Locais

Muitos programas práticos são constituídos de vários arquivos-fonte. Em programas distribuídos em múltiplos arquivos-fonte, cada arquivo constituinte do programa tipicamente contém (entre outras coisas) uma única função de grande porte (i.e., que ocupa várias páginas impressas) ou, mais comumente, várias funções com afinidades entre si. Algumas funções podem ser necessárias apenas nos arquivos que as contém, enquanto outras precisam ser chamadas em outros arquivos do programa. Uma função que é necessária apenas no arquivo que contém sua definição é denominada uma **função local** (ou **estática**); uma função que é chamada em outro arquivo diferente daquele que contém sua definição é chamado de **função global** (ou **externa**).

Pode-se especificar se uma função será local ou global por meio de um especificador de classe de armazenamento que antecede o tipo de retorno da função no cabeçalho de sua definição. Assim, acrescentando-se este item de informação, a sintaxe geral de cabeçalho de função torna-se[18]:

<classe de armazenamento> <tipo de retorno> <nome da função> (<declaração de argumentos>)

A classe de armazenamento de uma função pode ser **static**, para funções locais, ou **extern**, para funções globais. Na ausência de um especificador de classe de armazenamento na definição da função, o especificador **extern** é assumido.

Quando uma função global precisa ser chamada num arquivo outro que não aquele no qual foi definida, o arquivo que faz a chamada precisa incluir uma alusão à função, conforme já foi visto anteriormente. Esta alusão, que usualmente é colocada no início do arquivo que utiliza a função, informa que a função aludida é uma função externa definida em outro arquivo. É considerado bom estilo de programação utilizar sempre a palavra **extern** numa

[18] Apenas o cabeçalho do Formato 2 de função apresentado anteriormente é utilizado aqui, mas especificadores de classe de armazenamento podem de forma análoga anteceder cabeçalhos de funções que utilizam o Formato 1.

alusão, embora tal especificador não seja estritamente necessário, pois o mesmo é assumido por *default*.

Obviamente, pode ser que uma função considerada global pelo compilador seja acessada apenas no arquivo que a contém. Tais funções devem ser declaradas como locais com o uso do especificador apropriado (**static**). Isto não apenas torna o programa mais legível como também o código gerado pelo compilador será menor se este for informado que uma dada função será utilizada apenas no arquivo que contém sua definição (mais especificamente, neste caso, a tabela de símbolos gerada pelo compilador será menor).

2.5.2 Arquivos de Cabeçalho

É prática comum em programas constituídos de múltiplos arquivos-fonte incluir um arquivo de cabeçalho para cada um destes arquivos, exceto talvez para o arquivo principal (usualmente denominado `main.c`). Um arquivo de cabeçalho deve conter definições de funções, de tipos e de constantes simbólicas que tenham alguma afinidade.

Um programa de grande porte pode conter ainda outros tipos de arquivos de cabeçalhos contendo, por exemplo, apenas definições de tipos ou constantes simbólicas utilizadas por vários arquivos que constituem o programa.

Um programa contendo um grande número de variáveis globais pode também conter um arquivo próprio apenas para conter as definições destas variáveis. Este arquivo não é usualmente denominado arquivo de cabeçalho, mas sim arquivo de programa (extensão `.c`), pois o mesmo contém instruções (i.e., as inicializações implícitas ou explícitas das variáveis globais). Esta estratégia, no entanto, pode ser difícil de implementar em programas grandes desenvolvidos por vários programadores.

2.6 Tipos de Dados Definidos pelo Programador

Umas das características mais importantes da linguagem C é que ela permite que o programador crie seus próprios tipos de dados. Isto é possível com o uso da palavra reservada **typedef** que tem a seguinte sintaxe:

typedef <tipo> <nome do tipo>;

Note que a sintaxe de uma definição de tipo é semelhante àquela utilizada na definição de variáveis. Entretanto, ao contrário de uma declaração de variável, uma definição de tipo não causa a alocação de nenhum espaço em memória, ela apenas declara <*nome do tipo*> como sinônimo de <*tipo*>. Por exemplo, a declaração de tipo a seguir:

```
typedef   long  int  tInteiroLongo;
```

define um novo tipo de dados, denominado `tInteiroLongo` que é sinônimo do tipo **long int**. Em conseqüência desta definição, a declaração de variável a seguir:

```
tInteiroLongo  k;
```

é idêntica a:

```
long  int    k;
```

A convenção adotada aqui para identificadores de tipos é começar o nome do tipo com `t` (por exemplo, `tInteiroLongo`)[19].

[19] Alguns programadores preferem que o identificador seja terminado por `_t` (por exemplo, `InteiroLongo_t`).

Alguns programadores inexperientes podem confundir o uso de **typedef** com a diretiva **#define**, pois, em algumas situações, seus usos são, de fato, eqüivalentes. Por exemplo, a definição de tipo do último exemplo poderia ser substituída pela seguinte diretiva com o mesmo efeito daquela definição:

```
#define   tInteiroLongo   long  int
```

Entretanto, a diretiva **#define** é inadequada para substituir declarações de tipos mais complexas. Por exemplo, suponha que você deseje definir um tipo que represente um ponteiro para o tipo **char**. Então, você declararia corretamente este tipo como:

```
typedef   char  *tPonteiroParaChar;
```

No entanto, a tentativa de declaração deste tipo por meio da diretiva **#define**:

```
#define   tPonteiroParaChar  char *
```

é inadequada. Para entender isso melhor, suponha por exemplo que você deseje declarar dois ponteiros do tipo `tPonteiroParaChar`. Obviamente, com a primeira definição de tipo não haveria problema, mas com a segunda, o pré-processador faria a expansão de:

```
tPonteiroParaChar  p1, p2;
```

para:

```
char *p1, p2;
```

ou seja, apenas a primeira variável seria reconhecida como ponteiro; a segunda variável seria considerada como sendo do tipo **char** (e não do tipo **char ***).

Definições de tipos são usadas mais freqüentemente para nomear tipos estruturados mais complexos construídos pelo programador, conforme será visto nos próximos capítulos deste livro.

2.7 O Pré-processador de C

O pré-processador de C é um programa *conceitualmente* independente do compilador que é usualmente executado antes deste último. Apesar de preparar um ou mais arquivos para serem compilados e de ser executado automaticamente pelo compilador, o pré-processador de C é considerado um programa distinto e que funciona separadamente do compilador. Em particular, o pré-processador de C não *entende* a linguagem C; ele entende apenas sua própria linguagem. As instruções desta linguagem começam sempre com o símbolo # e duas delas (i.e., **#define** e **#include**) já foram introduzidas em seções anteriores. Tais instruções são denominadas **diretivas** e podem aparecer em qualquer ponto de um programa-fonte. Diferentemente de instruções da linguagem C, uma diretiva termina quando se introduz uma nova linha, e não com ponto-e-vírgula. Quando deseja-se que uma diretiva ocupe mais de uma linha, utiliza-se uma barra invertida (\) para indicar este 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 # na primeira coluna e que não exista espaço entre este símbolo e a palavra que identifica a instrução. Apesar de estas restrições não mais existirem em compiladores mais modernos, esta forma de escrever diretivas ainda é usual.

O pré-processador de C provê as facilidades enumeradas a seguir:

- Processamento de macros
- Inclusão de arquivos
- Compilação condicional
- Processamento de diretivas **#error**

- Processamento de diretivas **#pragma**

Estas facilidades serão discutidas nas seções a seguir.

2.7.1 Macros

Uma macro é um identificador que possui usualmente uma seqüência de símbolos associada denominada **corpo da macro**. Convencionalmente, nomes de macros são constituídos apenas de letras maiúsculas e sublinha, pois esta convenção facilita a identificação de macros espalhadas pelo programa. Uma macro é definida através da diretiva **#define**, como, por exemplo:

```
#define  NUMERO_DE_MESES_POR_ANO  12
```

Macros são usualmente definidas no início de um arquivo, e cada macro tem validade de seu ponto de definição até o final do arquivo. Em outras palavras, uma macro não tem validade em arquivos diferentes daquele na qual a mesma foi definida. Se você deseja que uma macro seja utilizada em vários arquivos, coloque-a num arquivo de cabeçalho e inclua-o nos arquivos que desejar.

Quando o nome de uma macro aparece num local diferente daquele de sua definição, o nome é substituído ***literalmente*** pelo corpo da macro. Este ato de substituição é denominado **expansão de macro**. Por exemplo, na expressão a seguir, a macro `NUMERO_DE_MESES_POR_ANO` seria expandida para `12` (considerando a definição anterior):

```
numeroDeAnos = totalDeMeses / NUMERO_DE_MESES_POR_ANO;
```

2.7.1.1 Macros com Argumentos

O uso mais simples e comum de macros é a representação de constantes simbólicas, o que já foi suficientemente comentado anteriormente. Um outro uso comum de macros envolve a utilização de argumentos, o que as tornam semelhantes às funções. Este tipo de macro tem a seguinte sintaxe:

#define <nome da macro>(<argumentos da macro>) <corpo da macro>

Os argumentos de uma macro devem ser separados por vírgulas e usualmente são representados por letras minúsculas. Deve-se notar ainda que argumentos de macros não são variáveis; i.e., eles não têm nem tipo nem memória alocada para si. Como exemplo de macro com argumento, considere a seguinte definição de macro:

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

A diretiva apresentada acima define a macro `QUADRADO` com um argumento, e tem por objetivo definir o quadrado de um número. Uma macro com argumentos 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));
```

Neste ponto, você deve estar se perguntando por que são utilizados tantos parênteses, que aparentemente são redundantes, na definição da macro acima. A resposta a esta questão é que, devido ao fato de toda expansão de macro ser *ipsis litteris*, a ausência desses parênteses pode fazer com que a macro não funcione em certas circunstâncias. Por exemplo, suponha que os parênteses em torno do corpo da macro do último exemplo sejam deixados de fora, resultando em:

```
#define  QUADRADO_ERRADO1(a) (a)*(a)
```

então, se esta macro fosse utilizada na instrução:

```
y = 100/QUADRADO_ERRADO1(5);
```

a expansão resultaria em:

```
y = 100/(5)*(5);
```

que é equivalente a (por que?):

```
y = (100/5)*5;
```

Em resumo, no último exemplo, a intenção do programador era certamente dividir `100` pelo quadrado de `5`, o que deveria resultar em `4` (`=100/25`). Entretanto, devido à ausência de parênteses em torno do corpo da macro, a expressão foi expandida para `(100/5)*5`, que resulta em `100`.

Agora suponha que se decida deixar de fora os parênteses em torno de cada argumento da macro, resultando na seguinte macro:

```
#define  QUADRADO_ERRADO2(a) (a*a)
```

O problema aqui surgiria quando a macro acima fosse utilizada numa expressão como a seguinte:

```
z = QUADRADO_ERRADO2(3 + 1);
```

A macro deste último exemplo seria expandida para:

```
z = (3 + 1*3 + 1);
```

que resultaria, finalmente, em `7`, enquanto que de acordo com a mais provável intenção do programador a expressão deveria resultar em `16` (verifique isto). Além destes problemas decorrentes da ausência de parênteses em definições de macros, existem outros problemas potenciais que poderão se manifestar com o uso indevido de macros. Algumas recomendações para evitar estes problemas serão apresentadas no final desta seção.

2.7.1.2 Comparações entre Macros e Funções

Normalmente, macros são executados mais rapidamente do que funções eqüivalentes, 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 algumas vantagens e desvantagens decorrentes do uso de macros. Você deve fazer um balanço entre vantagens e desvantagens antes de decidir utilizar uma macro em substituição a uma função numa situação particular.

Vantagens de Macros

• **Macros são mais rápidas** pois não requerem o ônus associado com chamadas de funções (por exemplo, empilhamento/desempilhamento de variáveis locais e parâmetros).

• **Uma macro pode servir para vários tipos de dados**, pois não existe restrição quanto aos tipos de dados dos argumentos de uma macro. Em contraste, é muito difícil escrever uma função igualmente adequada para quaisquer tipos de dados.

Desvantagens de Macros

• **Argumentos são avaliados a cada menção dos mesmos no corpo da macro**. Isto pode levar a resultados inesperados quando o argumento é substituído na invocação da macro por argumentos contendo operadores com efeitos colaterais (v. mais adiante).

• **O corpo de uma função é compilado apenas uma vez, enquanto que o corpo de uma macro é compilado sempre que a mesma é invocada**. O resultado é que um programa contendo muitas macros pode ocupar muito mais espaço de armazenamento do que um programa que utiliza funções eqüivalentes.

• **É mais difícil depurar programas contendo macros** porque o código-fonte passa por mais de um nível de tradução (v. mais adiante).

• **Macros não checam os tipos de seus argumentos**. Este fato torna o uso de macros mais sensível a erros do que o uso de funções.

• **Macros não podem chamar recursivamente a si mesmas**. Conseqüentemente, elas não podem implementar naturalmente uma solução de natureza recursiva.

• **Macros não possuem endereços**. Conseqüentemente, elas não podem ser representadas por ponteiros[20].

Pode-se concluir das observações acima que existe uma compensação entre macros e funções com relação à rapidez de execução e tamanho do programa-objeto resultante. De um lado, macros são executadas mais rapidamente, mas o programa-objeto poderá ocupar muito espaço; de outro lado, funções são mais lentas, mas o programa resultante ocupará menos espaço. 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 freqüência.

Certamente, o uso de macros é mais vantajoso em substituição a funções que aparecem em poucas instruções do programas, mas que são chamadas um número repetido de vezes. Por exemplo, se uma operação é executada em 100 locais diferentes de um programa, é melhor utilizar uma função para implementar tal operação. Mas, se uma operação aparece numa única instrução que é executada 100 vezes (por exemplo, uma operação dentro de um laço de repetição), é melhor usar uma macro. Também, funções pequenas (i.e., cujos corpos contém poucas instruções) são melhores candidatas à substituição por macros do que funções grandes (i.e., contendo muitas instruções).

Muitos compiladores de C oferecem certas operações, contidas em arquivos de biblioteca, em dois formatos: (1) como função, e (2) como macro. Em tal situação, o programador tem a liberdade de escolher o formato que melhor convém a um determinado programa.

Finalmente, devido às suas diferentes características, uma macro e uma função aparentemente eqüivalentes podem produzir resultados diferentes. Por exemplo, considere a seguinte definição de função:

```
int  Quadrado(int   x)
{
    return  x*x;
}
```

À primeira vista, esta função parece ser equivalente à macro `QUADRADO` definida anteriormente. Realmente, ambas produzirão os mesmos resultados se os argumentos utilizados forem sempre do tipo **int**. Agora, observe o que acontece quando um argumento do tipo **float**, digamos `2.5`, é passado para a função e para a macro. A macro produzirá o resultado correto (`6.25`), enquanto que a função retornará `4` (por que?), que não seria o resultado esperado.

[20] Funções possuem esta propriedade, que pode ser muito útil em algumas aplicações, conforme será visto no próximo capítulo.

2.7.1.3 Macros Produtoras de Strings

Quando colocado antes de um argumento no corpo de uma macro, o símbolo # faz com que o pré-processador envolva o argumento com aspas que identificam uma cadeia de caracteres constante em C.

O uso do símbolo "#" para produção de *strings* é indicado quando se deseja que um dado argumento de uma macro seja tratado tanto como uma expressão quanto como uma cadeia de caracteres. Considere, por exemplo, a seguinte macro:

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

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

```
double var1 = 1.0, var2 = 2.0, var3 = 3.0;

...

IMPRIME_VARIAVEL(var1);
IMPRIME_VARIAVEL(var2);
IMPRIME_VARIAVEL(var3);
```

o pré-processador expande as chamadas da macro para:

```
printf("Valor de %s = %lf\n", "var1", var1);
printf("Valor de %s = %lf\n", "var2", var2);
printf("Valor de %s = %lf\n", "var3", var3);
```

É interessante notar ainda que é permitido o uso de outros *strings* constantes seguindo o corpo de uma macro produtora de *strings*. Neste caso, os *strings* constantes adicionais serão automaticamente concatenados com o *string* resultante da expansão. Por exemplo:

```
#define CADEIA(s) #s " concatenada com outra."
```

Quando esta última macro é invocada, como em:

```
printf(CADEIA(Isto e' uma cadeia de caracteres));
```

o pré-processador expande esta macro para:

```
printf("Isto e' uma cadeia de caracteres concatenada com outra.");
```

2.7.1.4 Concatenação de Tokens

O operador ## faz com que dois itens (*tokens*) numa definição de macro sejam concatenados para formar um item único. Por exemplo:

```
#define   CRIA_VAR(n)  variavel##n
```

Esta macro concatena o token `variavel` com o argumento da macro. Assim, se esta macro fosse invocada como:

```
double CRIA_VAR(2);
```

ela seria expandida para:

```
double variavel2;
```

Note que este resultado não seria possível utilizando outros tipos de macros vistos anteriormente.

2.7.1.5 Removendo a Definição de uma Macro

Uma vez definida, uma macro retém sua definição até o final do arquivo fonte que a contém, mas pode-se remover a definição de uma macro por meio da diretiva **#undef**. Esta diretiva é necessária quando se deseja redefinir uma macro com um valor diferente do valor atual da macro, pois, caso contrário, isto não seria permitido. Isto é, podem existir várias definições de uma mesma macro num arquivo, mas estas definições devem ser iguais. Se você precisar redefinir uma macro com um valor diferente, você precisa antes remover sua antiga definição por meio de **#undef.**

2.7.1.6 Macros Predefinidas

Existem algumas macros que são predefinidas (ou embutidas) no pré-processador de C, e não é permitido remover ou redefinir estas macros. Os nomes destas macros começam e terminam com um duplo sublinha (__). Estas macros, juntamente com suas expansões, são apresentadas na **Tabela 13**.

MACRO	EXPANSÃO
__LINE__	O número da linha no arquivo fonte na qual a macro é invocada
__FILE__	O nome do arquivo fonte no qual a macro é invocada
__TIME__	A hora na qual o arquivo atual foi compilado
__DATE__	A data na qual o arquivo atual foi compilado
__STDC__	1, se o compilador é 100% ANSI/ISO; 0 ou indefinida se o compilador não for ANSI/ISO

Tabela 2: Macros Predefinidas

As macros `__LINE__` e `__FILE__` podem ser muito úteis durante a fase de depuração de um programa. As macros `__TIME__` e `__DATE__` servem para registrar a hora e a data da última vez que um arquivo foi compilado. Finalmente, a macro `__STDC__` serve para indicar se um compilador obedece à especificação ANSI/ISO ou não, e é útil quando utilizada em conjunto com compilação condicional (v. mais adiante).

Um exemplo prático de uso das macros `__LINE__` e `__FILE__` é a definição da macro `ASSERT` a seguir:

```
#define ASSERT( x ) if (!(x) ) {\
                   printf("A condicao %s na linha %d do arquivo %s falhou.",\
                            #x, __LINE__, __FILE__); \
                      exit(1); \
                    }
```

O último exemplo também ilustra o uso do produtor de *string* # numa situação na qual o argumento da macro é utilizado tanto como expressão quanto como *string*. Como ilustração de uso dessa última macro, suponha que na linha número 100 do arquivo `main.c` de um programa a macro `ASSERT` seja invocada como:

```
ASSERT( i > 0 )
```

A macro `ASSERT` seria então expandida para:

```
if (!( i > 0) ) {
    printf("A condicao %s na linha %d do arquivo %s falhou.", "i > 0",
            100, "main.c");
    exit(1);
}
```

Suponha ainda que, durante a execução do programa, neste ponto do mesmo, `i` seja menor do que `0`. Então, a condição da instrução anterior resultante da expansão da macro seria satisfeita, o programa imprimiria o seguinte no meio de saída padrão:

```
A condição i > 0 na linha 100 do arquivo main.c falhou.
```

e o programa seria então abortado devido à chamada da função **`exit`**`(1)`[21]. A macro `ASSERT` é uma macro extremamente útil na depuração de programas, conforme será visto mais adiante. Uma macro similar a esta apresentada aqui pode ser encontrada na biblioteca padrão de C, no arquivo `<assert.h>` , com o nome **assert** (em minúsculas).

Como exemplo de uso das macros `__TIME__` e `__DATE__`, considere a seguinte função que, quando invocada, imprime a data e a hora de compilação do arquivo que a contém:

```
void  ImprimeDataEHoraDeCompilacao( void )
{
   printf("Este arquivo %s foi compilado às %s do dia %s\n",
            __FILE__, __TIME__,__DATE__);
}
```

2.7.1.7 Macros Vazias

O corpo de uma macro pode ser vazio dando origem a uma **macro vazia**. Uma macro vazia não é expandida para nada e é útil apenas quando utilizada 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.

2.7.1.8 Cuidados com Usos de Macros

Macros devem ser utilizadas com cautela, pois algum descuido pode acarretar num erro difícil de depurar. A seguir estão enumerados alguns cuidados especiais que você deve seguir quando definir e invocar macros.

1. **Nunca termine uma macro com ponto-e-vírgula**.

   Este erro é comum e muitas vezes é inócuo, mas algumas vezes pode acarretar em *bugs* que são difíceis de serem consertados. Por exemplo, considere a seguinte definição (errônea) de macro:

   ```
   #define TAMANHO_MAXIMO   100;
   ```

   Se esta macro for utilizada numa instrução tal como:

   ```
   x = TAMANHO_MAXIMO;
   ```

   após a expansão, a instrução aparecerá como:

   ```
   x = 100;;
   ```

[21] A função `exit()` provoca o encerramento imediato do programa. Normalmente, utiliza-se o valor 0 como parâmetro desta função para indicar encerramento normal do programa, e um valor diferente de zero em caso contrário. Maiores detalhes sobre esta função podem ser encontrados no **Apêndice C**.

e o ponto-e-vírgula excedente será interpretado pelo compilador como uma instrução vazia. Nesta situação, portanto, não haveria nenhum problema. Agora, suponha que esta macro fosse utilizada na instrução a seguir:

```
y = TAMANHO_MAXIMO / 10;
```

Esta última instrução apareceria após a expansão como:

```
y = 100; / 10;
```

Quando o compilador encontrar esta instrução, obviamente ele irá indicar a existência de um erro sintático, pois esta instrução não é legal em C. A dificuldade que o programador inexperiente encontra para corrigir este erro é que o compilador irá indicá-lo exatamente nesta linha de instrução, e não na diretiva **#define**, aonde o erro foi realmente introduzido. Afinal, tudo que o programador vê como indicação de erro é a instrução: "`y = TAMANHO_MAXIMO / 10;`", que aparentemente é legal[22].

Este erro causado pelo uso indevido de ponto-e-vírgula na macro é às vezes difícil de ser consertado, mas ainda não é o pior, pois, como ocorre com todo erro sintático, o programa será executado apenas quando o erro for consertado. Um erro muito mais difícil de consertar é aquele que o compilador não o indica. Suponha, por exemplo, a seguinte definição (novamente errônea) de macro:

```
#define  CONDICAO_LEGAL  (indicador == 1);
```

Se esta macro for utilizada na instrução:

```
while  CONDICAO_LEGAL {
       .
       :
}
```

ela irá ser expandida para:

```
while  (indicador == 1); {
       .
       :
}
```

O problema aqui é similar àquele no qual coloca-se indevidamente ponto-e-vírgula antes do corpo de uma instrução **while** (v. **Seção 1.5.3**). Entretanto, aqui o problema é pior, pois a instrução original (i.e., a primeira instrução **while** acima) parece ser perfeitamente legal.

Para concluir, se você deparar-se com *bugs* não identificados no programa, um dos itens que você deve examinar são todas as macros para verificar se alguma delas possui ponto-e-vírgula no final.

2. **Não coloque na definição de uma macro sinal de igualdade separando o corpo da macro de seu identificador**.

Outro erro comum em definições de macros (que aflige em particular programadores de Pascal) é o uso indevido do símbolo de igualdade como se uma definição da macro fosse uma inicialização. Por exemplo, considere a seguinte definição errônea de macro:

```
#define  VALOR_MAXIMO = 100
```

[22] Lembre-se que o compilador começa a atuar após a expansão de todas as macros, o que é realizado pelo pré-processador.

Este tipo de engano pode levar a erros obscuros. Por exemplo, suponha a utilização desta última macro na instrução a seguir:

```
for  (i=VALOR_MAXIMO; i > 0; i--){
    ...
}
```

Esta instrução seria expandida para:

```
for  (i== 100; i > 0; i--){
      .
      :
}
```

Note que a expressão `i=VALOR_MAXIMO` que o programador esperava que fosse uma atribuição foi transformada numa expressão relacional. O pior neste tipo de erro é que para o compilador a instrução é perfeitamente legal, mas o programa certamente não irá funcionar conforme o esperado.

Exercício: Especule sobre o que poderia acontecer neste último exemplo se a variável `i` fosse: (a) inicializada antes com zero; (b) não fosse inicializada anteriormente.

3. **Nunca coloque espaço em branco entre o nome da macro e o parêntese esquerdo que aparece antes dos argumentos da macro**.

Muitas vezes, este tipo de engano resulta em erros que são detectados pelo compilador, mas isto pode também resultar num erro lógico difícil de ser encontrado. Suponha, por exemplo, a seguinte definição errônea de macro:

```
#define  MINHA_MACRO_ERRADA (a) (2*(a))
```

Então, se essa macro fosse invocada na instrução:

```
y = MINHA_MACRO_ERRADA(5);
```

o compilador não entenderia a macro `MINHA_MACRO_ERRADA` como sendo uma macro definida com argumento e indicaria o erro na chamada da mesma.

4. **Sempre envolva cada argumento bem como todo o corpo da macro com parênteses.**

A não observância desta regra pode levar a resultados inesperados, conforme já foi discutido anteriormente. Portanto, mesmo que você tenha certeza que a macro não será utilizada em situações que acarretariam em erro, acostume-se a sempre seguir esta regra.

5. **Valores constantes precedidos por sinal devem ser envolvidos por parênteses em declarações de constantes simbólicas**.

Isto evita que algum compilador interprete o sinal da constante erroneamente quando este for combinado com outro operador. Por exemplo, a macro:

```
#define  MINHA_CONSTANTE_ERRADA -1
```

quando invocada em:

```
x = -MINHA_CONSTANTE_ERRADA;
```

poderia ser expandida para:

```
x = --1;
```

o que evidentemente é uma construção ilegal (note que os dois menos unários foram unidos para formar o operador de decremento).

6. **Cuidado com o uso de macros em conjunção com operadores com efeitos colaterais**.

Este problema refere-se à utilização de macros e não a definições em si. Considere, por exemplo, a seguinte definição de macro:

```
#define  MAXIMO(a, b) ((a) > (b) ? (a) : (b))
```

Esta última macro, se utilizada corretamente, resulta no maior entre dois números. Agora, suponha que esta macro é utilizada como a seguir:

```
z = MAXIMO(x++, y);
```

Esta última instrução será expandida como:

```
z = ((x++) > (y) ? (x++) : (y));
```

O que acontece aqui é que, quando `x++` (que é igual a `x`, lembram?) for maior do que `y`, a variável `x` será incrementada duas vezes, o que provavelmente não é aquilo que se deseja. Portanto, por segurança, nunca utilize macros com argumentos contendo operadores com efeitos colaterais.

2.7.2 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 um conjunto de diretivas similares à instrução condicional **if-else** de C. Estas diretivas são **#if**, **#else**, **#elif**, e **#endif**, e possuem a seguinte sintaxe:

#if <condição 1>
    <trecho de programa que será compilado se condição 1 for satisfeita>
#elif <condição 2>
    <trecho de programa que será compilado se condição 2 for satisfeita>
  .
  :
#elif <condição N>
    <trecho de programa que será compilado se condição N for satisfeita>
#else
    <trecho de programa compilado se nenhuma das condições anteriores for satisfeita>
#endif

A expressão condicional que segue um **#if** ou **#elif** deve ser uma constante (usualmente representada por uma macro sem argumentos), e sua interpretação é igual àquela vista anteriormente para expressões condicionais em C (com a exceção de que a expressão é convertida para **long int**). Existem, entretanto, algumas diferenças sintáticas adicionais com relação ao **if-else** de C que não são aparentes no esquema sintático apresentado acima:

• A expressão condicional não precisa estar entre parênteses (mas, estes podem ser acrescentados se você os desejar).

• **#elif** é análogo a uma construção **else if** de C.

• Os trechos de programa não são delimitados por { e }, como blocos dentro de uma instrução **if-else**.

• Cada bloco de diretivas **#if** deve ser terminado com **#endif**.

Outra diferença importante entre blocos de diretivas **#if** e o **if-else** do C é que diretivas não determinam se uma instrução será executada ou não: ***elas apenas especificam aquilo que será efetivamente compilado posteriormente pelo compilador***.

O pré-processador expande macros antes da avaliação de qualquer bloco de diretivas **#if**. Se uma expressão condicional contém um nome que não tenha ainda sido definido, ele será expandido para zero. Por exemplo:

```
#undef  x
#if  x
```

é expandida para:

```
#if  0
```

Um uso comum de compilação condicional é o de escolher entre os dois formatos de alusões de funções de acordo com o compilador utilizado, como ilustrado a seguir:

```
#if  (__STDC__ == 1)
    extern  int  MinhaFuncao(char  a, long  b);
#else
    extern  int  MinhaFuncao();
#endif
```

De acordo com as diretivas acima, se o programa estiver sendo compilado por um compilador ANSI/ISO (`__STDC__` é definido como sendo 1), o formato moderno de alusão será utilizado; caso contrário, o estilo antigo será compilado (i.e., apenas uma das duas alusões será incluída no programa final).

Compilação condicional também é particularmente útil durante o estágio de depuração de um programa, conforme será visto mais adiante.

Testando a Existência de Macros

Pode-se ainda especificar compilação condicional com base na existência ou não de uma macro (independentemente do valor da macro), utilizando, respectivamente, as diretivas **#ifdef** ou **#ifndef**[23]. 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), a primeira instrução **printf**() será compilada; caso contrário, a segunda chamada de **printf**() será compilada.

Na maioria das situações, pode-se usar **#if** ao invés de **#ifdef** ou **#ifndef**, uma vez que o nome de uma macro expande para zero se ela não for definida. A única situação aonde isto não é possível é quando a macro já é definida para zero. Suponha, por exemplo, que você deseja definir a macro `FALSE` para zero. Se você utilizar **#if** para testar se `FALSE` é definida, como no exemplo abaixo:

```
#if  !FALSE
#   define  FALSE   0
#endif
```

[23] Estas diretivas podem ainda ser escritas como **#if defined** e **#if !defined**.

`FALSE` será redefinida mesmo que ela já tivesse sido definida com zero. Mas, o mais importante é que ela não será redefinida se ela tiver sido definida com um valor diferente de zero (verifique isso). Uma forma de evitar este problema é através do seguinte conjunto de diretivas:

```
#ifndef  FALSE
#   define  FALSE   0
#elif   FALSE
#   undef  FALSE
#   define  FALSE   0
#endif
```

Uma outra situação em que as diretivas **#ifdef** e **#ifndef** são úteis é quando se testa se uma dada macro vazia está definida ou não. Um exemplo prático de tal situação é o de um esquema que previne (ou pelo menos minimiza) a inclusão múltipla de um arquivo de cabeçalho[24]. Esta situação é esquematizada a seguir:

```
#ifndef _ArquivoC_H_ /* arquivoc.h é o nome do arquivo de cabeçalho */
#define _ArquivoC_H_

/* O conteúdo do arquivo vem aqui  */

#endif
```

Suponha que o arquivo de cabeçalho é denominado `ArquivoC.h`. Então, se um outro arquivo ainda não incluiu `ArquivoC.h`, a macro `_ArquivoC_H_` não é ainda definida. Assim, esta macro é definida na segunda linha do arquivo `ArquivoC.h` (o valor desta macro não interessa; por isso, ela é definida como uma macro vazia) e a inclusão do arquivo ocorre normalmente. Agora, suponha que um arquivo tente incluir o arquivo `ArquivoC.h` mais de uma vez. Neste caso, como a macro `_ArquivoC_H_` foi definida na primeira inclusão do arquivo `ArquivoC.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 da macro de controle (aqui denominada `_ArquivoC_H_`) de modo que esta macro não coincida com alguma outra macro do programa. Usualmente, esta macro é escolhida como uma combinação do nome do arquivo com caracteres de sublinha.

2.7.3 Inclusão de Arquivos

A diretiva **#include** para inclusão de arquivos já foi suficientemente discutida anteriormente (v. **Seção 1.7**), e sua referência foi incluída aqui apenas por uma questão de complemento.

2.7.4 A Diretiva #error

A diretiva **#error** permite que um erro seja indicado durante o estágio de pré-processamento do programa. Aquilo que segue a diretiva (usualmente um texto explicativo) será impresso no meio de saída e a compilação será encerrada. Tipicamente, esta diretiva é utilizada para verificar se alguma condição ilegal de compilação está sendo utilizada. Por exemplo:

```
#if  __WIN32__
#  error  Este programa nao deve ser compilado para \
          sistemas de 32 bits
#endif
```

Portanto, se alguém tentasse compilar o programa contendo as diretivas acima com a macro `__WIN32__` com o valor 1, o pré-processador emitiria a mensagem de erro:

---

[24] Isto é útil e às vezes necessário. Por exemplo, suponha que um arquivo denominado `c1.h` inclua um arquivo denominado `c2.h`, e que um arquivo denominado `arq.c` inclui ambos `c1.h` e `c2.h`; como `c1.h` já está incluído em `c2.h`, `c1.h` seria incluído duas vezes em `arq.c` na ausência de uma estratégia que previna inclusão múltipla de arquivos.

```
Este programa nao deve ser compilado para sistemas de 32 bits
```

e a compilação não prosseguiria.

2.7.5 A Diretiva #pragma

A diretiva **#pragma** executa tarefas dependentes da implementação utilizada. Isto é, cada compilador tem a liberdade de incluir identificadores que têm significados especiais quando precedidos pela diretiva **#pragma**. Por exemplo, o compilador Borland C++ contém a diretiva **#pragma warn** que, quando utilizada com um nome que representa uma mensagem de advertência, faz com que o compilador suprima (nome da mensagem precedido com '-') ou ative (nome da mensagem precedido com '+') a apresentação da respectiva mensagem de advertência.

Consulte o manual da implementação de C que você está utilizando para verificar quais são as diretivas **#pragma** suportadas por seu compilador.

2.8 Comentários e Legibilidade de Programas

2.8.1 Comentários

Comentários em C são quaisquer seqüências de símbolos colocada entre os símbolos **delimitadores de comentários** `/*` e `*/`[25]. O objetivo principal de comentários é o de explicar o programa para outras pessoas e para você mesmo quando for lê-lo algum tempo após a escrita do programa. Comentários podem também ser utilizados para teste e depuração de programas, conforme será analisado mais adiante. A seguir, serão apresentados alguns conselhos sobre **quando**, **como** e **aonde** comentar um programa a fim de clarificá-lo.

O melhor momento para comentar um trecho de um programa é exatamente quando este trecho está sendo escrito. Isto aplica-se especialmente para aqueles trechos de programa contendo truques, inspirações momentâneas ou coisas do gênero, que o próprio programador terá dificuldade em entender algum tempo depois. Alguns comentários passíveis de serem esquecidos, tal como a data de início da escrita do programa, também devem ser acrescentados no momento da escrita do programa. Outros comentários podem ser acrescentados quando o programa estiver pronto, embora o ideal continue sendo comentar todo o programa à medida em que o mesmo é escrito.

Comentários devem ser claros e dirigidos para programadores com alguma experiência na linguagem. Eles não têm que ser didáticos como alguns comentários apresentados aqui e em outros texto de ensino de programação. Estes comentários didáticos são úteis nestes textos que têm exatamente o objetivo de ensinar, mas comentários num programa real têm o objetivo de explicar o programa para programadores, e não o de ensinar a um leigo na linguagem o que está sendo feito.

Existem dois formatos básicos de comentários: (1) **comentário de bloco**; e (2) **comentário de linha**, que devem ser utilizados conforme é sugerido a seguir ou de acordo com a necessidade.

2.8.1.1 Comentários de Bloco

[25] De acordo com o padrão ANSI/ISO não é permitido o uso de comentários aninhados; i.e., um comentário dentro de outro comentário. Entretanto, algumas implementações de C permitem isto, o que pode ser bastante útil na depuração de programas, pois permite comentar trechos de programa que já contenham comentários.

Blocos de comentário são utilizados no início do programa (i.e., no início do arquivo contendo a função **main()**) com o objetivo informativo de apresentar o propósito geral do programa, data de início do projeto, nome do(s) programador(es), versão do programa, nota de direitos autorais (*copyright*) e qualquer outra informação pertinente. Por exemplo:

```
/****
 *
 * Título Programa: MeuPrograma
 *
 * Autor: José da Silva
 *
 * Data de Início do Projeto: 10/11/2001
 * Última modificação: 19/11/2002
 *
 * Versão: 2.01b
 *
 * Descrição Geral: Este programa faz isto e aquilo.
 *
 * Dados de Entrada: Este programa espera os seguinte dados ...
 *
 * Dados de Saída: Este programa produz como saída o seguinte ...
 *
 * Copyright © 2002 José da Silva Software Ltda.
 *
 ****/
```

Cada arquivo constituinte de um programa multi-arquivo deve conter um comentário em forma de bloco como o apresentado acima, mas com um conteúdo ligeiramente diferente como esquematizado a seguir:

```
/****
 *
 * Nome do Arquivo: MeuArquivo.c
 *
 * Programador: João da Silva
 *
 * Data de Criação: 20/11/2001
 * Última modificação: 18/11/2002
 *
 * Descrição: Este módulo implementa funções que fazem isto e aquilo.
 *
 * Copyright © 2002 José da Silva Software Ltda.
 *
 ****/
```

Um bloco de comentário também deve ser utilizado antes da definição de cada função (exceto, é claro, antes da função **main()**, que deve ser precedida pelo comentário informativo do programa como um todo). Este comentário deve conter, entre outras coisas:

- o nome da função;
- o propósito da função;
- como a função deve ser utilizada;
- descrição dos argumentos da função (salientando o que é entrada, saída e entrada/saída);
- descrição do valor retornado pela função (não confunda isto com argumento de saída ou argumento de entrada/saída)
- descrição sucinta do algoritmo utilizado (se o algoritmo for um algoritmo clássico, basta que se apresente uma referência ao mesmo).

Considere como exemplo:

```
/****
 *
 * Função OrdenaDados()
 *
 * Entrada: Os dados a serem ordenados no formato tal e tal...
```

```
 *
 * Saída: Os dados ordenados no formato...
 *
 * Valor retornado: 1 se bem sucedida; 0 em caso contrário
 *
 * Descrição do algoritmo: Baseado no método quicksort
 *
 ****/
```

Observe neste último exemplo que, como o algoritmo (*quicksort*) utilizado pela função é um algoritmo bastante conhecido, não é necessário descrevê-lo; uma referência é suficiente. Se um programador que estiver lendo este programa desconhecer este algoritmo, ele poderá encontrá-lo em muitos livros de programação.

Finalmente, outro local aonde blocos de comentário se fazem necessários é em trechos de programas difíceis de serem entendidos por conterem truques, sutilezas, otimizações em código de baixo nível, etc. Este tipo de comentário deve ter uma estética diferente dos blocos de comentário apresentados anteriormente para distingui-lo dos demais. Por exemplo,

```
      /* O trecho de programa a seguir faz isso e aquilo    */
      /* Descrição de como isso é implementado...           */
      /* Descrição de como aquilo é implementado...         */
```

2.8.1.2 Comentários de Linha

Comentários de linha são utilizados para comentar uma única linha de instrução, mas, em si, eles podem ocupar mais de uma linha por razões de estética e legibilidade. Como o propósito aqui é o de explicar uma instrução que não é clara para um programador de C, não se deve comentar aquilo que é óbvio para um programador da linguagem. Por exemplo, o seguinte comentário não deve ser feito, pois é evidente para qualquer programador com alguma experiência em C:

```
x = ++y;    /* x recebe o valor de y incrementado de 1 */
```

Comentários como o do último exemplo não são apenas irrelevantes ou redundantes: pior, eles tornam o programa mais difícil de ler pois desviam a atenção do leitor para informações inúteis. Entretanto, algumas instruções ou declarações mais complexas ou confusas do que a anterior devem ser comentadas não apenas para facilitar a leitura como também para que o programador verifique se a instrução faz exatamente aquilo que o comentário diz que ela faz. Por exemplo,

```
int  (*(*f())[])();   /* f é declarada como uma função que retorna um */
                      /* ponteiro para um arranjo de ponteiros para   */
                      /* funções que retornam o tipo int              */
```

Formatar comentários de modo que estes sejam legíveis e ao mesmo tempo não interrompam o fluxo de escrita do programa é difícil e muitas vezes exige alguma habilidade. Quando sobra pouco espaço à direita da instrução para a escrita de um comentário de linha, pode-se colocar o mesmo precedendo a instrução e sem espaço vertical entre os dois, como por exemplo:

```
      /* A instrução a seguir calcula tal coisa de tal e tal modo */
minhaVariavel1 = minhaVariavel2 + 10*MinhaFuncao(minhaVariavel2);
```

Na dúvida entre o que é óbvio e o que não é óbvio para ser comentado, comente, pois, neste caso, é melhor pecar por redundância do que por omissão. Também, lembre-se que comentários não devem ser utilizados para compensar um programa mal escrito. A melhor forma de se adquirir prática na escrita de comentários é ler programas (encontrados, por

exemplo, em tutoriais e programas-exemplo) escritos por programadores profissionais e observar os vários estilos de comentários utilizados[26].

2.8.2 Endentação

Endentação deve ser utilizada para indicar que as instruções endentadas estão sob controle da instrução anterior não-endentada. O uso ***consistente*** de endentação é essencial para uma boa legibilidade do programa.

Use sempre endentação para instruções dentro de blocos (i.e., entre um par de colchetes), ou para instruções que fazem parte de alguma estrutura de controle ou seguindo um rótulo. Exemplos:

```
                    {
                         <instrução 1>;
                         .
                         :
                         <instrução N>;
                    }

                    while  (x) {
                         <instrução 1>;
                         .
                         :
                         <instrução N>;
                    }

istoEhUmRotulo:
               /* Escreva aqui as instruções a serem executadas  */
               /* quando o programa for desviado para cá         */
```

Outras sugestões para endentação são apresentadas ao longo do texto. Você não precisa seguir essas sugestões, mas qualquer que seja sua escolha seja consistente (i.e., use-a coerentemente em todo o programa).

2.8.3 Uso de Espaços em Branco

O uso judicioso de **espaços em branco** é essencial para uma boa legibilidade do programa. Isso inclui **espaços horizontais** e **espaços verticais**.

Além de servirem como endentação, espaços horizontais devem ainda ser utilizado em torno de identificadores e operadores para melhorar a legibilidade de expressões. A seguir, alguns conselhos úteis sobre o uso de espaços horizontais:

• Use espaços horizontais para enfatizar precedência de operadores. Por exemplo,

```
5*3 + 4
```

é melhor do que:

```
5 * 3 + 4
```

• Use espaços horizontais para alinhar identificadores numa seção de declaração. Por exemplo,

```
short              var1;
register long  int var2;
```

[26] Repetindo, textos utilizados no ensino de programação não são ideais para este propósito, pois como foi afirmado anteriormente, alguns comentários são didáticos demais para serem utilizados na prática.

Alguns conselhos úteis sobre o uso de espaços verticais incluem:

- Use espaços verticais para separar funções, blocos, conjuntos de instruções com alguma afinidade lógica dentro de um bloco, etc.

- Use espaços verticais em branco para separar seções logicamente diferentes do programa (ou função).

2.8.4 Escolha de Identificadores

2.8.4.1 Convenções

O uso consistente de **convenções** para a criação de identificadores das diversas entidades que compõem um programa facilita bastante a leitura do mesmo. A seguir, são resumidas algumas sugestões para escrita de identificadores que já têm sido utilizadas em partes anteriores do corrente texto.

• **Nomes de variáveis**. Comece com letra minúscula; se o nome da variável for composto, utilize letra maiúscula no início de cada palavra seguinte, inclusive palavras de ligação (por exemplo, preposições e conjunções); não utilize sublinha.

• **Nomes de tipos**. Siga a regra acima para nomes de variáveis, mas comece com a letra `t` ou termine com `_t`.

• **Nomes de funções**. Utilize a mesma regra para nomes de variáveis, mas comece com letra maiúscula.

• **Nomes de macros**. Utilize sempre letras maiúsculas; se o nome for composto, utilize sublinha para separar os componentes.

2.8.4.2 Representatividade de Identificadores

Identificadores que exercem papeis importantes no programa devem ter nomes que sejam significativos com relação à função exercida (por exemplo, `idadeDoAluno` é muito melhor do que simplesmente `x`).

Identificadores com importância menor, não precisam ter nomes significativos. Por exemplo, uma variável utilizada apenas como variável de controle num laço **for** pode ser nomeada `i` (não precisa ser denominada `contador`)

Funções que não retornam nenhum valor (exceto, talvez um código de erro), devem ser nomeadas pelo que elas fazem (por exemplo, `FormataTexto()`). Por outro lado, funções cujo principal objetivo é o de retornar um valor (por exemplo, resultante de algum cálculo) devem ser nomeadas baseadas naquilo que elas retornam (por exemplo, uma função que calcula e retorna o valor do fatorial de um número deve ser nomeada como `Fatorial()`, e não como `CalculaFatorial()`). Funções que retornam um dentre dois valores possíveis (por exemplo, *sim/não*, ou *verdadeiro/falso*) podem ser nomeadas começando com `Eh` (representando "é"; por exemplo, `EhValido()`) ou `Sao` (representando "são"; por exemplo, `SaoIguais()`).

Finalmente, não utilize identificadores nem muito longos nem muito abreviados: encontre um meio-termo que seja sensato.

2.8.5 Guias de Estilo de Programação em C

O mini-manual de estilo apresentado nesta seção está longe de ser completo. Uma fonte interessante e facilmente acessível sobre estilo de programação em C é o documento conhecido como *Indian Hill* da AT&T, encontrado facilmente na Internet.

2.9 Teste e Depuração de Programas

2.9.1 Testando um Programa

Testar um programa significa verificar se o mesmo funciona conforme o esperado (i.e., conforme foi especificado no projeto do mesmo). Normalmente, qualquer programa não-trivial possui erros (***bugs***). Portanto, o objetivo maior de testar um programa é o de encontrar erros que impeçam o seu funcionamento normal[27].

Num software comercial, existem duas fases principais de testes: **alfa** e **beta**. Na fase alfa, os testes são realizados por programadores e engenheiros que compõem a equipe de desenvolvimento do produto, enquanto que na fase beta os testadores são usuários voluntários que utilizam o programa numa situação real. Idealmente, a fase beta deve começar apenas quando a fase alfa está encerrada. É importante observar ainda que quando um programa entra em fase de testes ele deve ser *congelado* no sentido de que as únicas modificações no programa devem ser realizadas apenas com o objetivo de corrigir erros de programação (i.e., adições de novas características não devem ser permitidas durante a fase de testes).

Técnicas utilizadas para testes (*alfa*) de programas complexos fazem parte de uma disciplina isolada em si, e uma apresentação completa destas técnicas está obviamente além do escopo deste livro. Portanto, aqui, apenas duas das técnicas mais comuns e fáceis de serem implementas serão descritas.

A primeira técnica é conhecida como **inspeção de programa** e consiste em ler atenciosamente o programa e responder a uma lista de verificação contendo questões referentes a erros comuns em programação na linguagem de codificação do programa. A segunda técnica comum de verificação de programas é conhecida como **teste exaustivo**, e consiste em utilizar dados típicos de entrada e *simular o computador* através da execução manual do programa. Num teste exaustivo, é importante que sejam utilizados casos de entrada qualitativamente diferentes. Também, é igualmente importante que sejam testadas não apenas entradas válidas, mas também algumas entradas inválidas.

2.9.2 Depuração de Programas

Nem mesmo os programadores mais experientes escrevem programas livres de erros em sua primeira tentativa. Uma grande parcela do tempo gasto em programação, portanto, é dedicada à tarefa de encontrar e corrigir erros. **Depurar** um programa significa localizar e consertar trechos do programa que provocam seu mal funcionamento. Apesar de estarem intimamente relacionados, teste e depuração de um programa não significam a mesma coisa. Um bom teste deve ser capaz de apontar um comportamento errôneo de um programa, mas não indica precisamente aquilo que causa tal comportamento. Por outro lado, a depuração deve determinar precisamente a instrução ou conjunto de instruções que causam o comportamento errôneo do programa e consertar os trechos de programa que causam o problema (por exemplo, através da rescrita dos mesmos).

2.9.3 Classificação de Erros de Programação

Erros de programação são usualmente classificados em três categorias:

---

[27] Outro objetivo importante dos testes é verificar a eficiência do programa em termos de tempo de resposta e espaço ocupado em memória pelo mesmo, mas isto está fora do escopo deste livro.

• **Erro de Compilação** (ou **Erro de Sintaxe**). Este tipo de erro ocorre devido a uma violação das regras de sintaxe da linguagem C. Um programa contendo erros de sintaxe não pode ser nem compilado nem (muito menos) executado. Causas comuns para este tipo de erro são:

◊ falhas de digitação;
◊ omissão de ponto-e-vírgula;
◊ referência a variáveis que não foram declaradas;
◊ chamada de uma função com um número errado de argumentos ou com argumentos de tipos incompatíveis com aqueles na definição da função;
◊ atribuição de um valor de um tipo ilegal para uma variável.

Erros de compilação são relativamente fáceis de serem consertados, pois muitas vezes o compilador indica precisamente a instrução errônea e o tipo de erro que ocorreu.

• **Erro de Execução**. Um erro de execução não impede um programa de ser compilado, mas faz com que a execução do mesmo seja interrompida de maneira anormal (algumas vezes causando até mesmo a queda de todo o sistema operacional no qual o programa está sendo executado). Um exemplo comum deste tipo de erro é uma tentativa de divisão por zero.

• **Erro de Lógica**. Erro de lógica[28] é um erro que nem impede a compilação nem acarreta interrupção da execução de um programa. Entretanto, um programa contendo um erro deste tipo não funciona conforme o esperado. Por exemplo, você pede ao programa para executar uma determinada tarefa e o programa não realiza esta tarefa satisfatoriamente.

2.9.4 Técnicas de Depuração

Por mais incrível que possa parecer, a etapa mais difícil de depuração de um programa consiste em localizar precisamente a instrução ou conjunto de instruções que causam o mal funcionamento do programa. Como já foi dito anteriormente, encontrar erros sintáticos não é difícil, mesmo quando o compilador não é capaz de apontá-los com precisão. A seguir, serão apresentadas algumas técnicas comuns utilizadas para localizar erros lógicos e de execução em programas. Antes de utilizar alguma destas técnicas, porém, é importante que o programador determine precisamente a natureza do erro e quando o mesmo ocorre (por exemplo, sempre que o programa recebe tal entrada, ele apresenta tal e tal comportamento). A situação ideal ocorre quando o programador é capaz de reproduzir o erro sempre que introduz dados possuindo as mesmas características (i.e., quando o erro não é **aleatório**, mas sim **sistemático**).

Antes da apresentação de algumas técnicas utilizadas em depuração, é importante chamar a atenção para o fato de que depuração é uma atividade muito difícil; provavelmente, muito mais difícil do que a escrita de programas. Em resumo, depuração requer paciência, criatividade, esperteza, e, principalmente, muita experiência por parte do programador. Portanto, as técnicas descritas a seguir devem servir apenas como um guia introdutório. À medida em que você se tornar um programador experiente, você será capaz de desenvolver suas próprias técnicas e de utilizar versões mais sofisticadas daquelas apresentadas aqui.

2.9.4.1 Uso do Compilador

O compilador não é propriamente uma ferramenta de depuração, mas pode ser seu primeiro aliado na luta contra o surgimento de *bugs*. Isto é, o uso preventivo do compilador pode ajudá-lo a evitar que alguns erros de ocorram antes mesmo de o programa ser executado pela primeira vez. Para utilizar o compilador de forma preventiva, utilize-o sempre com a

[28] A palavra “lógica” aqui é utilizada como sinônimo de “raciocínio”.

opção de apresentar todas as advertências disponíveis de possíveis erros[29]. Examine cuidadosamente cada mensagem de advertência emitida pelo compilador e conserte todas as instruções que correspondam a uma dada advertência, mesmo que você tenha certeza que uma dada construção não causará problema. Agindo de modo contrário, uma mensagem de advertência importante poderá deixar de ser notada.

2.9.4.2 Uso de printf()

A função **printf()** (bem como outras funções de saída) é uma ferramenta bastante útil em depuração. Existem dois usos principais de **printf()** em depuração:

- Examinar o valor de uma ou mais variáveis em vários pontos do programa
- Verificar o fluxo de controle do programa (por exemplo, para determinar se uma ou mais instruções são executadas).

A técnica consiste em distribuir chamadas de **printf()** em vários pontos do programa. Enquanto distribui chamadas de **printf()** pelo programa, certifique-se de que você será capaz de distinguir cada uma destas chamadas quando a mesma for executada. Como exemplos de uso de **printf()** num programas têm-se:

```
printf("Valor de x antes da instrucao tal e tal eh: %f.\n", x);
printf("Valor de x apos o segundo while eh: %f.\n", x);
printf("Instrucoes seguindo o else do 3o. if serao executadas agora.\n");
```

Evidentemente, no caso de um programa que ***quebra*** devido a um erro de execução, as chamadas de **printf()** que foram executadas estão obviamente antes do erro.

2.9.4.3 Uso de Asserções e Compilação Condicional

Compilação condicional é útil em depuração, uma vez que ela o permite incluir ou excluir trechos de programa de acordo com o valor de uma macro que indica se o programa está em processo de depuração[30].

Quando o programa está em fase de depuração, esta macro é definida como 1, e ela é utilizada para incluir trechos de programa utilizados exclusivamente para depuração. Quando a fase de depuração é encerrada, redefine-se esta macro para 0, e aqueles trechos utilizados apenas na depuração do programa não serão incluídos no código final. Consequentemente, o código gerado pelo compilador quando a fase de depuração estiver encerrada será menor do que durante aquela fase.

O uso da macro `ASSERT`, já apresentada anteriormente, para verificar se certas condições são satisfeitas em vários pontos de um programa também é uma ferramenta extremamente útil em depuração. Aqui, esta macro será definida de modo mais sofisticado em conjunção com compilação condicional como:

```
#define  DEBUG  1

#if  DEBUG
#define ASSERT( x )  if (!(x) ) {\
             printf("A condicao %s na linha %d do arquivo %s, falhou.", \
                    #x, __LINE__, __FILE__); \
             exit(1); \
                    }
#else
```

---

[29] No caso do compilador Borland C++, escolha ***Options***, então ***Project***, então ***Messages***, e finalmente escolha ***All***.

[30] Usualmente, esta macro é denominada `DEBUG` ou `NDEBUG`.

```
#define   ASSERT( x )  0
#endif
```

A diferença entre esta definição e a definição anterior da macro `ASSERT` (v. Seção 2.7.1) é que quando a macro `DEBUG` expande para `0`, indicando final da fase de depuração, a macro `ASSERT` expande simplesmente para `0`.

Suponha, por exemplo, que seu programa contém uma divisão `x/y` que deve sempre ser executada. Então, para que não ocorra erro de execução, a variável `y` deve ser diferente de zero no instante da divisão. Você poderia assegurar isto incluindo uma asserção imediatamente antes desta divisão, como no exemplo a seguir:

```
ASSERT (y != 0);
z = x/y;
```

Com a inclusão desta asserção, quando `y` assumisse o valor 0 indevido, o programa seria abortado e uma mensagem de erro seria emitida indicando a linha e o arquivo nos quais a asserção falhou (i.e., quando a expressão `y != 0` deixou de ser satisfeita). É importante notar a diferença entre o uso de uma asserção, como a do exemplo acima, e o simples uso de um **if** aparentemente eqüivalente. Por exemplo, se a chamada de `ASSERT` fosse substituída pela instrução **if**:

```
if (y != 0)
    z = x/y;
```

talvez não fosse possível determinar quando a instrução `z = x/y` é executada ou não.

2.9.4.4 Uso de Comentários

Comentários também são utilizados para excluir da compilação um trecho de programa que se suspeita esteja defeituoso durante o processo de depuração do programa. Esta técnica de depuração funciona da seguinte maneira:

(1) O programa apresenta um comportamento inesperado e você suspeita que um determinado trecho do programa está provocando este comportamento indesejado.

(2) Comente este trecho de programa para excluí-lo do programa final e veja como o programa resultante se comporta (algumas outras adaptações no programa, como, por exemplo, retirada de comentários pré-existentes, podem ser necessárias antes de recompilá-lo);

(3) Se o programa continua a apresentar o mesmo erro, é provável que este erro não seja provocado pelo trecho de programa comentado. Assim, você deve procurar o erro em outro local do programa.

(4) Se o programa não apresentar o mesmo erro, é provável que sua conjetura sobre a causa do erro tenha sido boa e que o trecho de programa comentado seja realmente o causador do erro. Se este trecho for grande ao ponto de não permitir identificar exatamente qual a instrução causadora do erro, repita o procedimento a partir do passo (2), mas agora comente um trecho de programa menor dentro da porção de programa anteriormente comentada.

2.9.4.5 Depuradores de Alto Nível

Depuradores são programas que dão suporte à tarefa de depuração de outros programas. Existem depuradores de baixo e alto níveis. Os mais difíceis de serem utilizados são os depuradores de baixo–nível que requerem conhecimento de *assembly* e de como o compilador traduz o código-fonte do programa analisado.

A maioria dos compiladores vêm acompanhados de depuradores de alto-nível que são razoavelmente fáceis de utilizar. Apesar de diferirem em algumas facilidades adicionais, estes depuradores têm basicamente o mesmo funcionamento que será descrito a seguir.

Basicamente, depuradores de alto nível permitem que o programador coloque **pontos-de-parada** (***breakpoints***) ao longo do programa. Estes pontos de parada são utilizados em instruções (i.e., não se pode colocar um ponto de parada numa declaração de variável, por exemplo). Quando um programa contendo *breakpoints* é executado sob a supervisão de um depurador, a execução do programa pára imediatamente antes da execução da instrução contendo o primeiro *breakpoint*. Neste instante, o programador tem a opção de examinar valores de variáveis locais e globais, avaliar expressões, etc. Ele tem também a opção de continuar a execução do programa normalmente até que o próximo *breakpoint* seja atingido, ou ele pode ainda decidir executar o programa instrução a instrução. Para executar esta última opção, o programador tem duas alternativas. Uma delas é (usualmente) denominada *step* (ou *step over*), e a outra é (usualmente) denominada *step into*. Ambas as alternativas executam uma única instrução e param a execução do programa antes da próxima instrução. A diferença entre *step* e *step into* é que *step* trata chamadas de funções como se fossem instruções indivisíveis, enquanto que *step into* considera uma função como um conjunto de instruções. Portanto, quando a opção *step into* é utilizada imediatamente antes da chamada de uma função, a execução do programa pára imediatamente antes da primeira instrução no corpo da função. Em contraste, quando a opção *step* é utilizada imediatamente antes da chamada de uma função, a execução do programa pára imediatamente antes da instrução seguindo a chamada da função. Quando a próxima instrução a ser executada não é uma chamada de função, não existe diferença entre *step* e *step into*.

2.10 Exercícios de Revisão

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

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

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

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

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

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

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

8. Escreva protótipos para funções que possuam os seguintes argumentos e tipos de retorno:

   (a) Tipo de retorno: nenhum; argumentos: um **float** e um ponteiro para um **char**;
   (b) Tipo de retorno: um ponteiro para **unsigned int**; argumento: um ponteiro para um **char** constante;

9. O que são funções de biblioteca? Por que estas funções não são (estritamente) consideradas parte integrante da linguagem C?

10. Suponha que você precise classificar os caracteres lidos no meio de entrada nas seguintes categorias:

- Espaço em Branco: espaço, '\n', '\r', '\t'
- Pontuação: ", ', !, ?, ;, :, .
- Letra: a-z, A-Z

- Número: 0-9
- Desconhecido: qualquer outro caractere

(a) Defina constantes para cada uma das categorias acima (por exemplo, `ESPACO_EM_BRANCO`). Idealmente, estas constantes devem fazer parte de uma enumeração.

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

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

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

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

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

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

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

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

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

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

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

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

16. O que é classe de armazenamento de uma variável?

17. Descreva os quatro tipos de escopo de identificadores em C.

18. (a) O que é uma variável automática? (b) Como uma variável automática é definida? (c) Qual é o escopo de uma variável automática? (d) O que acontece quando uma variável automática não é explicitamente inicializada dentro de uma função?

19. (a) O que é uma alusão a uma função? (b) Quando uma alusão é requerida num programa em C? (c) O que é protótipo de uma função? (d) Qual é a relação entre protótipo e alusão de uma função? (e) Qual é a vantagem advinda do uso de protótipos de funções em relação ao uso do estilo antigo de alusões?

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

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

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

23. Escreva uma macro, denominada `ABS(x)`, que deve ser expandida para o valor absoluto do argumento `x`.

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

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

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

(a)
```
#include <stdio.h>

main()
{
    int a, contador;
    int Funcao(int  contador);

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

Funcao(int  x)
{
   int y;

   y += x;
   return y;
}
```

(b)
```
#include <stdio.h>

main()
{
    int a, contador;
    int Funcao(int  contador);

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

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

  y += x;
  return  y;
}
```

(c)
```
#include <stdio.h>

main()
{
    int  a = 0, b = 1, contador;
    int  Funcao1(int  contador);
     int  Funcao2(int  contador);
```

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

Funcao1(int x)
{
  int  b;
  int  Funcao2(int  a);

  b = Funcao2(a)
  return  b;
}

Funcao2(int  a)
{
  static  int  b = 1;

  b += 1;
  return  b + a;
}
```

27. Escreva uma declaração de macro para cada uma das seguintes situações:

(a) Definir a constante simbólica `PI` para representar `3.1415927`

(b) Definir uma macro denominada `AREA` que calcule a área de um círculo em termos de seu raio. Utilize a constante `PI` definida no item (a) nesta macro.

(c) Definir uma macro denominada `CIRCUNFERENCIA` que calcula a circunferência de um círculo em termos de seu raio. Utilize a constante `PI` definida no item (a) nesta macro.

(d) Definir uma macro em múltiplas linhas, denominada `JUROS`, que avalia os juros compostos dados pela fórmula:

$$F = P(1 + i)^n$$

onde `F` é o montante de dinheiro que será acumulado após n anos, `P` é o montante de dinheiro original, `i = 0,01r` e `r` é a taxa anual de juros expressa como percentagem. Suponha que todos os parâmetros representam valores de ponto-flutuante.

28. Explique o propósito de cada uma dos seguintes grupos de diretivas:

(a)
```
# if !defined(FLAG)
    #define FLAG
#endif
```

(b)
```
#if defined(PASCAL)
    #define  BEGIN  {
    #define  END    }
#endif
```

(c)
```
#ifdef  CELSIUS
    #define TEMPERATURA(t)    0.555555 * (t - 32)
#else
    #define TEMPERATURA(t)    1.8 * t + 32
#endif
```

(d)
```
#ifndef  DEBUG
    #define  OUT     printf("x = %f\n", i, y[i])
 #else
    #define OUT for (contador = 1; contador <= n; ++contador)\
                    printf("i = %d     x = %f\n ", i, y[i])
 #endif
```

(e)
```
#if define(DEBUG)
```

```
        #undef DEBUG
    #endif
```

(f)
```
    #ifdef  CHECK_ERRO
        #define  MENSAGEM(linha)   printf(*linha)
    #endif
```

29. (a) Por que algumas instruções são endentadas num programa em C? (b) Endentação é absolutamente necessária num programa em C?

30. Como são classificados os tipos de erros em programação de alto nível?

31. (a) O que são comentários de bloco e aonde estes devem ser utilizados? (b) O que são comentários de linha e aonde estes devem ser utilizados?

32. Qual é a melhor ocasião para escreverem-se comentários e por que?

33. Por que o uso de comentários irrelevantes pode prejudicar a legibilidade de um programa?

34. *Comentários que não correspondem ao código que eles tentam explicar são altamente prejudiciais à legibilidade de um programa*. Apresente situações aonde o programador pode, por falta de atenção ou conhecimento, introduzir comentários desta natureza num programa.

35. (a) Diferencie depuradores de alto nível de depuradores de baixo nível. (b) Por que depuradores de baixo nível são mais difíceis de utilizar do que depuradores de alto nível?

36. (a) Como funciona a técnica de depuração que utiliza **printf()**? (b) Compare essa técnica de depuração com a técnica de depuração que faz uso de comentários.

37. Explique como funciona em depuração o uso da macro `ASSERT` discutida no texto.

38. O que significa um ponto-de-parada (***breakpoint***) em depuração de alto nível?

39. Qual é a diferença entre os comandos ***step over*** e ***step into*** num depurador de alto nível?

2.11 Exercícios de Programação

EP2.1) Escreva um programa em C que calcula a média ponderada de uma lista de n números reais introduzidos no meio de entrada. A média ponderada de n números, x_1, x_2, ..., x_n é dada pela fórmula:

$MP = f_1x_1 + f_2x_2 + ... + f_nx_n$

onde cada f_i é o peso associado ao x_i correspondente, e deve satisfazer as seguintes relações:

$0 \leq f_i < 1$, e $f_1 + f_2 + ... + f_n = 1$. Teste seu programa com os dados da seguinte tabela:

i	**f_i**	**x_i**
1	0.06	27.5
2	0.08	13.4
3	0.08	53.8
4	0.10	29.2
5	0.10	74.5

6	0.10	87.0
7	0.12	39.9
8	0.12	47.7
9	0.12	8.1
10	0.12	63.2

EP2.2) Escreva um programa em C que recebe um número inteiro positivo no meio de entrada e responde como saída se o número é primo ou não.

EP2.3)

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

(b) Escreva um programa em C que receba como entrada um valor real e um valor inteiro, utilize a função descrita em (a) para calcular o valor do real elevado ao inteiro, e, finalmente, imprima o resultado.

EP2.4) Escreva um programa contendo as declarações a seguir que imprima o endereço de cada uma das variáveis:

```
char   ch;
int    k;
float  fl;
```