4. Funções

W. Celes e J. L. Rangel

4.1. Definição de funções

As funções dividem grandes tarefas de computação em tarefas menores. Os programas em C geralmente consistem de várias pequenas funções em vez de poucas de maior tamanho. A criação de funções evita a repetição de código, de modo que um procedimento que é repetido deve ser transformado numa função que, então, será chamada diversas vezes. Um programa bem estruturado deve ser pensado em termos de funções, e estas, por sua vez, podem (e devem, se possível) esconder do corpo principal do programa detalhes ou particularidades de implementação. Em C, tudo é feito através de funções. Os exemplos anteriores utilizam as funções da biblioteca padrão para realizar entrada e saída. Neste capítulo, discutiremos a codificação de nossas próprias funções.

A forma geral para definir uma função é:

```
tipo_retornado  nome_da_função (lista de parâmetros...)
{
   corpo da função
}
```

Para ilustrar a criação de funções, consideraremos o cálculo do fatorial de um número. Podemos escrever uma função que, dado um determinado número inteiro não negativo `n`, imprime o valor de seu fatorial. Um programa que utiliza esta função seria:

```
/* programa que le um numero e imprime seu fatorial */

#include <stdio.h>

void fat (int n);

/* Função principal */
int main (void)
{
   int n;
   scanf("%d", &n);
   fat(n);
   return 0;
}

/* Função para imprimir o valor do fatorial */
void fat ( int n )
{
   int i;
   int f = 1;
   for (i = 1; i <= n; i++)
      f *= i;
   printf("Fatorial = %d\n", f);
}
```

Notamos, neste exemplo, que a função `fat` recebe como parâmetro o número cujo fatorial deve ser impresso. Os parâmetros de uma função devem ser listados, com seus respectivos tipos, entre os parênteses que seguem o nome da função. Quando uma função não tem parâmetros, colocamos a palavra reservada `void` entre os parênteses. Devemos notar que `main` também é uma função; sua única particularidade consiste em ser a função automaticamente executada após o programa ser carregado. Como as funções `main` que temos apresentado não recebem parâmetros, temos usado a palavra `void` na lista de parâmetros.

Além de receber parâmetros, uma função pode ter um valor de retorno associado. No exemplo do cálculo do fatorial, a função `fat` não tem nenhum valor de retorno, portanto colocamos a palavra `void` antes do nome da função, indicando a ausência de um valor de retorno.

```
void fat (int n)
{
   . . .
}
```

A função `main` obrigatoriamente deve ter um valor inteiro como retorno. Esse valor pode ser usado pelo sistema operacional para testar a execução do programa. A convenção geralmente utilizada faz com que a função `main` retorne zero no caso da execução ser bem sucedida ou diferente de zero no caso de problemas durante a execução.

Por fim, salientamos que C exige que se coloque o *protótipo* da função antes desta ser chamada. O *protótipo* de uma função consiste na repetição da linha de sua definição seguida do caractere (`;`). Temos então:

```
void fat (int n);     /* obs: existe ; no protótipo */

int main (void)
{
   . . .
}

void fat (int n)      /* obs: nao existe ; na definição */
{
   . . .
}
```

A rigor, no protótipo não há necessidade de indicarmos os nomes dos parâmetros, apenas os seus tipos, portanto seria válido escrever: `void fat (int);`. Porém, geralmente mantemos os nomes dos parâmetros, pois servem como documentação do significado de cada parâmetro, desde que utilizemos nomes coerentes. O protótipo da função é necessário para que o compilador verifique os tipos dos parâmetros na chamada da função. Por exemplo, se tentássemos chamar a função com `fat(4.5);` o compilador provavelmente indicaria o erro, pois estaríamos passando um valor real enquanto a função espera um valor inteiro. É devido a esta necessidade que se exige a inclusão do arquivo `stdio.h` para a utilização das funções de entrada e saída da biblioteca padrão. Neste arquivo, encontram-se, entre outras coisas, os protótipos das funções `printf` e `scanf`.

Uma função pode ter um valor de retorno associado. Para ilustrar a discussão, vamos re-escrever o código acima, fazendo com que a função `fat` retorne o valor do fatorial. A função `main` fica então responsável pela impressão do valor.

```
/* programa que le um numero e imprime seu fatorial (versão 2) */

#include <stdio.h>

int fat (int n);

int main (void)
{
   int n, r;
   scanf("%d", &n);
   r = fat(n);
   printf("Fatorial = %d\n", r);
   return 0;
}

/* funcao para calcular o valor do fatorial */
int fat (int n)
{
   int i;
   int f = 1;
   for (i = 1; i <= n; i++)
      f *= i;
   return f;
}
```

4.2. Pilha de execução

Apresentada a forma básica para a definição de funções, discutiremos agora, em detalhe, como funciona a comunicação entre a função que chama e a função que é chamada. Já mencionamos na introdução deste curso que as funções são independentes entre si. As variáveis locais definidas dentro do corpo de uma função (e isto inclui os parâmetros das funções) não existem fora da função. Cada vez que a função é executada, as variáveis locais são criadas, e, quando a execução da função termina, estas variáveis deixam de existir.

A transferência de dados entre funções é feita através dos parâmetros e do valor de retorno da função chamada. Conforme mencionado, uma função pode retornar um valor para a função que a chamou e isto é feito através do comando `return`. Quando uma função tem um valor de retorno, a chamada da função é uma expressão cujo valor resultante é o valor retornado pela função. Por isso, foi válido escrevermos na função `main` acima a expressão `r = fat(n);` que chama a função `fat` armazenando seu valor de retorno na variável `r`.

A comunicação através dos parâmetros requer uma análise mais detalhada. Para ilustrar a discussão, vamos considerar o exemplo abaixo, no qual a implementação da função `fat` foi ligeiramente alterada:

```
/* programa que le um numero e imprime seu fatorial (versão 3) */

#include <stdio.h>

int fat (int n);

int main (void)
{
   int n = 5;
   int r;
   r = fat ( n );
   printf("Fatorial de %d = %d \n", n, r);
   return 0;
}

int fat (int n)
{
   int f = 1.0;
   while (n != 0)
   {
      f *= n;
      n--;
   }
   return f;
}
```

Neste exemplo, podemos verificar que, no final da função `fat`, o parâmetro `n` tem valor igual a zero (esta é a condição de encerramento do laço `while`). No entanto, a saída do programa será:

```
Fatorial de 5 = 120
```

pois o valor da variável `n` não mudou no programa principal. Isto porque a linguagem C trabalha com o conceito de **passagem por valor**. Na chamada de uma função, o valor passado é atribuído ao parâmetro da função chamada. Cada parâmetro funciona como uma variável local inicializada com o valor passado na chamada. Assim, a variável `n` (parâmetro da função `fat`) é local e não representa a variável `n` da função `main` (o fato de as duas variáveis terem o mesmo nome é indiferente; poderíamos chamar o parâmetro de `v`, por exemplo). Alterar o valor de `n` dentro de `fat` não afeta o valor da variável `n` de `main`.

A execução do programa funciona com o **modelo de pilha**. De forma simplificada, o modelo de pilha funciona da seguinte maneira: cada variável local de uma função é colocada na pilha de execução. Quando se faz uma chamada a uma função, os parâmetros são copiados para a pilha e são tratados como se fossem variáveis locais da função chamada. Quando a função termina, a parte da pilha correspondente àquela função é liberada, e por isso não podemos acessar as variáveis locais de fora da função em que elas foram definidas.

Para exemplificar, vamos considerar um esquema representativo da memória do computador. Salientamos que este esquema é apenas uma maneira didática de explicar o que ocorre na memória do computador. Suponhamos que as variáveis são armazenadas na memória como ilustrado abaixo. Os números à direita representam endereços (posições)

fictícios de memória e os nomes à esquerda indicam os nomes das variáveis. A figura abaixo ilustra este esquema representativo da memória que adotaremos.

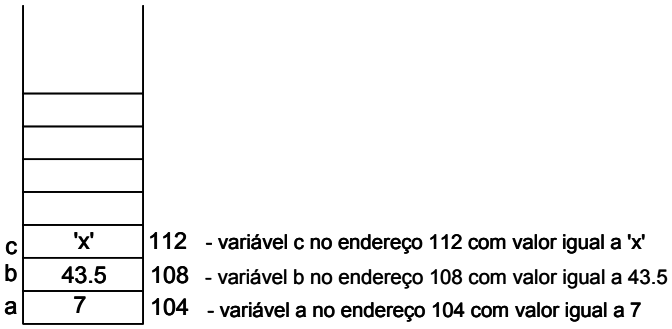

Figura 4.1: Esquema representativo da memória.

Podemos, então, analisar passo a passo a evolução do programa mostrado acima, ilustrando o funcionamento da pilha de execução.

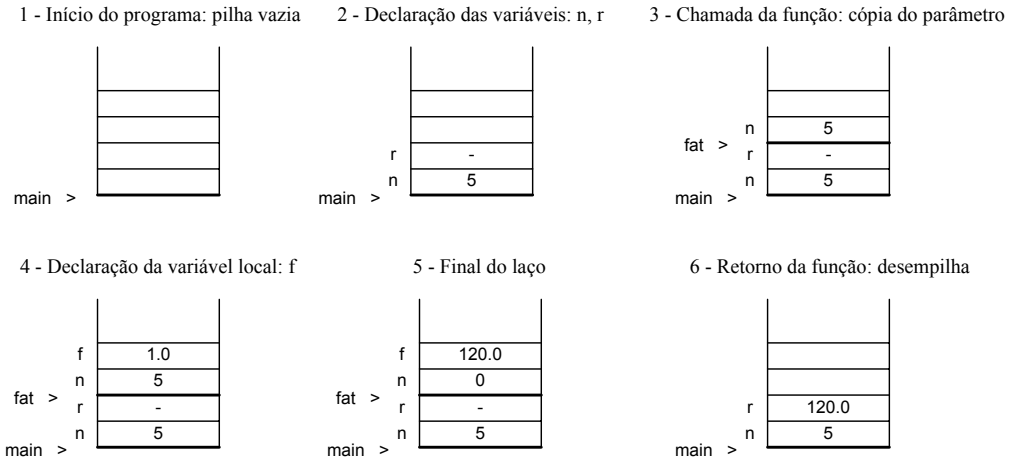

Figura 4.2: Execução do programa passo a passo.

Isto ilustra por que o valor da variável passada nunca será alterado dentro da função. A seguir, discutiremos uma forma para podermos alterar valores por passagem de parâmetros, o que será realizado passando o endereço de memória onde a variável está armazenada.

Vale salientar que existe outra forma de fazermos comunicação entre funções, que consiste no uso de variáveis globais. Se uma determinada variável global é visível em duas funções, ambas as funções podem acessar e/ou alterar o valor desta variável diretamente. No

entanto, conforme já mencionamos, o uso de variáveis globais em um programa deve ser feito com critério, pois podemos criar códigos com uma alto grau de interdependência entre as funções, o que dificulta a manutenção e o reuso do código.

4.3. Ponteiro de variáveis

A linguagem C permite o armazenamento e a manipulação de valores de endereços de memória. Para cada tipo existente, há um tipo ponteiro que pode armazenar endereços de memória onde existem valores do tipo correspondente armazenados. Por exemplo, quando escrevemos:

```
int a;
```

declaramos uma variável com nome `a` que pode armazenar valores inteiros. Automaticamente, reserva-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes).

Da mesma forma que declaramos variáveis para armazenar inteiros, podemos declarar variáveis que, em vez de servirem para armazenar valores inteiros, servem para armazenar valores de endereços de memória onde há variáveis inteiras armazenadas. C não reserva uma palavra especial para a declaração de ponteiros; usamos a mesma palavra do tipo com os nomes das variáveis precedidas pelo caractere `*`. Assim, podemos escrever:

```
int *p;
```

Neste caso, declaramos uma variável com nome `p` que pode armazenar endereços de memória onde existe um inteiro armazenado. Para atribuir e acessar endereços de memória, a linguagem oferece dois operadores unários ainda não discutidos. O operador unário `&` (“endereço de”), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. O operador unário `*` (“conteúdo de”), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro. Para exemplificar, vamos ilustrar esquematicamente, através de um exemplo simples, o que ocorre na pilha de execução. Consideremos o trecho de código mostrado na figura abaixo.

```
/*variável inteiro */
 int a;

/*variável ponteiro p/ inteiro */
 int *p;
```

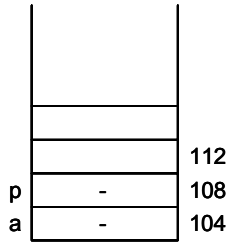

Figura 4.3: Efeito de declarações de variáveis na pilha de execução.

Após as declarações, ambas as variáveis, `a` e `p`, armazenam valores "lixo", pois não foram inicializadas. Podemos fazer atribuições como exemplificado nos fragmentos de código da figura a seguir:

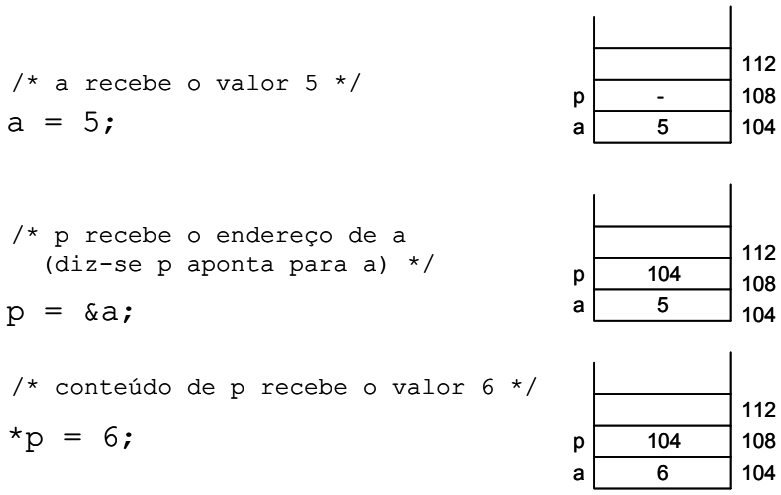

Figura 4.4: Efeito de atribuição de variáveis na pilha de execução.

Com as atribuições ilustradas na figura, a variável `a` recebe, indiretamente, o valor 6. Acessar `a` é equivalente a acessar `*p`, pois `p` armazena o endereço de `a`. Dizemos que `p` *aponta* para `a`, daí o nome *ponteiro*. Em vez de criarmos valores fictícios para os endereços de memória no nosso esquema ilustrativo da memória, podemos desenhar setas graficamente, sinalizando que um ponteiro aponta para uma determinada variável.

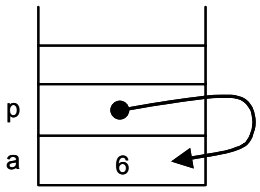

Figura 4.5: Representação gráfica do valor de um ponteiro.

A possibilidade de manipular ponteiros de variáveis é uma das maiores potencialidades de C. Por outro lado, o uso indevido desta manipulação é o maior causador de programas que "voam", isto é, não só não funcionam como, pior ainda, podem gerar efeitos colaterais não previstos.

A seguir, apresentamos outros exemplos de uso de ponteiros. O código abaixo:

```
int main ( void )
{
   int a;
   int *p;
   p = &a;
   *p = 2;
   printf(" %d ", a);
   return;
}
```

imprime o valor 2.

Agora, no exemplo abaixo:

```
int main ( void )
{
   int a, b, *p;
   a = 2;
   *p = 3;
   b = a + (*p);
   printf(" %d ", b);
   return 0;
}
```

cometemos um ERRO típico de manipulação de ponteiros. O pior é que esse programa, embora incorreto, às vezes pode funcionar. O erro está em usar a memória apontada por `p` para armazenar o valor 3. Ora, a variável `p` não tinha sido inicializada e, portanto, tinha armazenado um valor (no caso, endereço) "lixo". Assim, a atribuição `*p = 3;` armazena 3 num espaço de memória desconhecido, que tanto pode ser um espaço de memória não utilizado, e aí o programa aparentemente funciona bem, quanto um espaço que armazena outras informações fundamentais – por exemplo, o espaço de memória utilizado por outras variáveis ou outros aplicativos. Neste caso, o erro pode ter efeitos colaterais indesejados.

Portanto, só podemos preencher o conteúdo de um ponteiro se este tiver sido devidamente inicializado, isto é, ele deve apontar para um espaço de memória onde já se prevê o armazenamento de valores do tipo em questão.

De maneira análoga, podemos declarar ponteiros de outros tipos:

```
float *m;
char *s;
```

Passando ponteiros para funções

Os ponteiros oferecem meios de alterarmos valores de variáveis acessando-as indiretamente. Já discutimos que as funções não podem alterar diretamente valores de variáveis da função que fez a chamada. No entanto, se passarmos para uma função os valores dos endereços de memória onde suas variáveis estão armazenadas, a função pode alterar, indiretamente, os valores das variáveis da função que a chamou.

Vamos analisar o uso desta estratégia através de um exemplo. Consideremos uma função projetada para trocar os valores entre duas variáveis. O código abaixo:

```
/* funcao troca (versao ERRADA) */

#include <stdio.h>

void troca (int x, int y )
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}

int main ( void )
{
   int a = 5, b = 7;
   troca(a, b);
   printf("%d %d \n", a, b);
   return 0;
}
```

não funciona como esperado (serão impressos 5 e 7), pois os valores de `a` e `b` da função `main` não são alterados. Alterados são os valores de `x` e `y` dentro da função `troca`, mas eles não representam as variáveis da função `main`, apenas são inicializados com os valores de `a` e `b`. A alternativa é fazer com que a função receba os endereços das variáveis e, assim, alterar seus valores indiretamente. Reescrevendo:

```
/* funcao troca (versao CORRETA) */
#include <stdio.h>

void troca (int *px, int *py )
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}


int main ( void )
{
   int a = 5, b = 7;
   troca(&a, &b);     /* passamos os endereços das variáveis */
   printf("%d %d \n", a, b);
   return 0;
}
```

A Figura 4.6 ilustra a execução deste programa mostrando o uso da memória. Assim, conseguimos o efeito desejado. Agora fica explicado por que passamos o endereço das variáveis para a função `scanf`, pois, caso contrário, a função não conseguiria devolver os valores lidos.

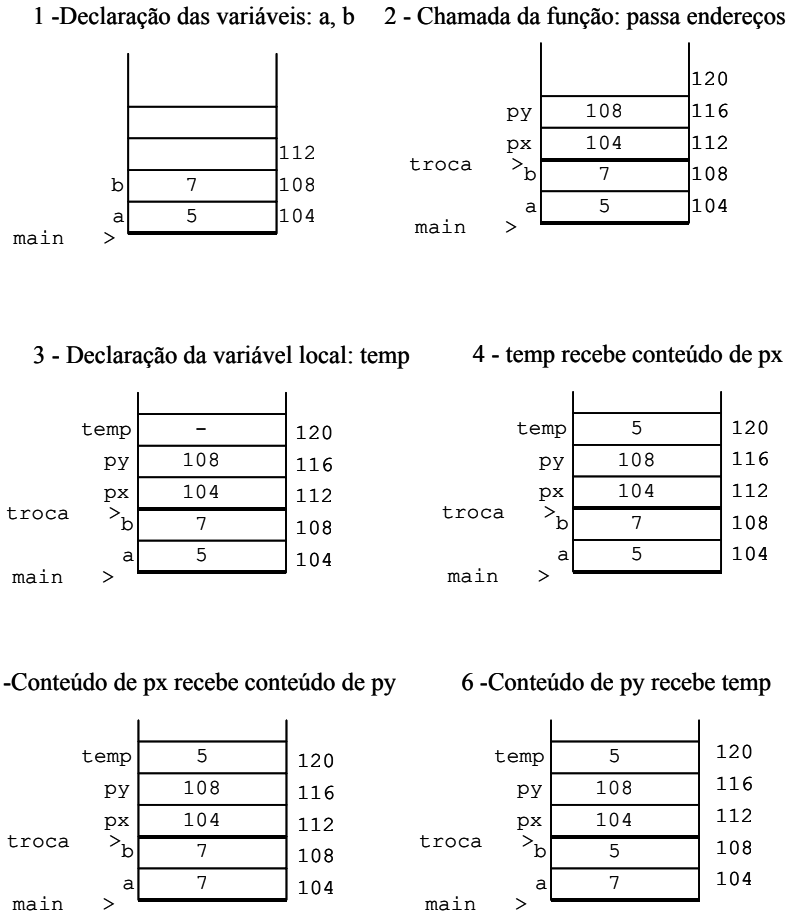

Figura 4.6: Passo a passo da função que troca dois valores.

4.4. Recursividade

As funções podem ser chamadas recursivamente, isto é, dentro do corpo de uma função podemos chamar novamente a própria função. Se uma função `A` chama a própria função `A`, dizemos que ocorre uma recursão direta. Se uma função `A` chama uma função `B` que, por sua vez, chama `A`, temos uma recursão indireta. Diversas implementações ficam muito mais fáceis usando recursividade. Por outro lado, implementações não recursivas tendem a ser mais eficientes.

Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução. Assim, mesmo quando uma função é chamada recursivamente, cria-se um ambiente local para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivéssemos chamando funções diferentes.

As implementações recursivas devem ser pensadas considerando-se a definição recursiva do problema que desejamos resolver. Por exemplo, o valor do fatorial de um número pode ser definido de forma recursiva:

$$n! = \begin{cases} 1, \, se \; n = 0 \\ n \times (n-1)!, \, se \; n > 0 \end{cases}$$

Considerando a definição acima, fica muito simples pensar na implementação recursiva de uma função que calcula e retorna o fatorial de um número.

```
/* Função recursiva para calculo do fatorial */

int fat (int n)
{
   if (n==0)
      return 1;
   else
      return n*fat(n-1);
}
```

4.5. Variáveis estáticas dentro de funções**

Podemos declarar variáveis estáticas dentro de funções. Neste caso, as variáveis não são armazenadas na pilha, mas sim numa área de memória estática que existe enquanto o programa está sendo executado. Ao contrário das variáveis locais (ou automáticas), que existem apenas enquanto a função à qual elas pertencem estiver sendo executada, as estáticas, assim como as globais, continuam existindo mesmo antes ou depois de a função ser executada. No entanto, uma variável estática declarada dentro de uma função só é visível dentro dessa função. Uma utilização importante de variáveis estáticas dentro de funções é quando se necessita recuperar o valor de uma variável atribuída na última vez que a função foi executada.

Para exemplificar a utilização de variáveis estáticas declaradas dentro de funções, consideremos uma função que serve para imprimir números reais. A característica desta função é que ela imprime um número por vez, separando-os por espaços em branco e colocando, no máximo, cinco números por linha. Com isto, do primeiro ao quinto número são impressos na primeira linha, do sexto ao décimo na segunda, e assim por diante.

```
void imprime ( float a )
{
   static int n = 1;

   printf(" %f   ", a);
   if ((n % 5) == 0) printf(" \n ");
   n++;
}
```

Se uma variável estática não for explicitamente inicializada na declaração, ela é automaticamente inicializada com zero. (As variáveis globais também são, por *default*, inicializadas com zero.)

4.6. Pré-processador e macros**

Um código C, **antes** de ser compilado, passa por um pré-processador. O pré-processador de C reconhece determinadas diretivas e altera o código para, então, enviá-lo ao compilador.

Uma das diretivas reconhecidas pelo pré-processador, e já utilizada nos nossos exemplos, é `#include`. Ela é seguida por um nome de arquivo e o pré-processador a substitui pelo corpo do arquivo especificado. É como se o texto do arquivo incluído fizesse parte do código fonte.

Uma observação: quando o nome do arquivo a ser incluído é envolto por aspas ("*arquivo*"), o pré-processador procura primeiro o arquivo no diretório atual e, caso não o encontre, o procura nos diretórios de *include* especificados para compilação. Se o arquivo é colocado entre os sinais de menor e maior (<*arquivo*>), o pré-processador não procura o arquivo no diretório atual.

Outra diretiva de pré-processamento que é muito utilizada e que será agora discutida é a diretiva de definição. Por exemplo, uma função para calcular a área de um círculo pode ser escrita da seguinte forma:

```
#define PI  3.14159

float area (float r)
{
   float a = PI * r * r;
   return a;
}
```

Neste caso, antes da compilação, toda ocorrência da palavra `PI` (desde que não envolvida por aspas) será trocada pelo número `3.14159`. O uso de diretivas de definição para representarmos constantes simbólicas é fortemente recomendável, pois facilita a manutenção e acrescenta clareza ao código. C permite ainda a utilização da diretiva de definição com parâmetros. É válido escrever, por exemplo:

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

assim, se após esta definição existir uma linha de código com o trecho:

```
v = 4.5;
c = MAX ( v, 3.0 );
```

o compilador verá:

```
v = 4.5;
c = ((v) > (4.5) ? (v) : (4.5));
```

Estas definições com parâmetros recebem o nome de **macros**. Devemos ter muito cuidado na definição de macros. Mesmo um erro de sintaxe pode ser difícil de ser detectado, pois o

compilador indicará um erro na linha em que se utiliza a macro e não na linha de definição da macro (onde efetivamente encontra-se o erro). Outros efeitos colaterais de macros mal definidas podem ser ainda piores. Por exemplo, no código abaixo:

```
#include <stdio.h>

#define DIF(a,b)   a - b

int main (void)
{
   printf(" %d ", 4 * DIF(5,3));
   return 0;
}
```

o resultado impresso é 17 e não 8, como poderia ser esperado. A razão é simples, pois para o compilador (fazendo a substituição da macro) está escrito:

```
   printf(" %d ", 4 * 5 - 3);
```

e a multiplicação tem precedência sobre a subtração. Neste caso, parênteses envolvendo a macro resolveriam o problema. Porém, neste outro exemplo que envolve a macro com parênteses:

```
#include <stdio.h>

#define PROD(a,b)   (a * b)

int main (void)
{
   printf(" %d ", PROD(3+4, 2));
   return 0;
}
```

o resultado é 11 e não 14. A macro corretamente definida seria:

```
#define PROD(a,b)   ((a) * (b))
```

Concluímos, portanto, que, como regra básica para a definição de macros, devemos envolver cada parâmetro, e a macro como um todo, com parênteses.