

Capítulo 4

Funções e ponteiros

Para a construção de programas estruturados, é sempre preferível dividir as grandes tarefas de computação em tarefas menores, e utilizar os resultados parciais das tarefas menores para compor o resultado final desejado. Na linguagem C, a criação de funções é o mecanismo adequado para codificar tarefas específicas. Um programa estruturado em C deve ser composto por diversas funções pequenas. Esta estratégia de codificação traz dois grandes benefícios: primeiro, facilita a codificação, pois codificar diversas funções pequenas, que resolvem problemas específicos, é mais fácil do que codificar uma única função maior; segundo, funções específicas podem ser facilmente reutilizadas em outros códigos. De fato, a criação de funções pode evitar 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 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 procedimentos básicos específicos. Neste capítulo, discutiremos a codificação de nossas próprias funções.

Como veremos, a codificação de funções exigirá, em alguns casos, o conhecimento de variáveis do tipo *ponteiros*. Uma variável do tipo ponteiro armazena endereços de memória; com isso, é possível alterar o valor armazenado em um espaço de memória sem necessariamente estar dentro do escopo da variável que usa o espaço. Este é um conceito crítico da linguagem C: a manipulação de ponteiros traz um grande poder de expressão para a linguagem; por outro lado, devemos manipular ponteiros com disciplina, pois temos que ter controle do espaço de memória que estamos acessando.

4.1 Criação de novas funções

Um programa em C é dividido em pequenas funções. Bons programas em geral são compostos por diversas pequenas funções. Como agrupar o código em funções é um dos desafios que devemos enfrentar. Como o próprio nome diz, uma função implementa uma funcionalidade. A divisão de um programa em funções ajuda na obtenção de códigos estruturados. Fica mais fácil entender o código, mais fácil manter, mais fácil atualizar, e mais fácil reutilizar.

Em C, a codificação de uma função segue a sintaxe mostrada a seguir:

```
_tipo_do_retorno_ _nome_da_função_ ( _lista_de_parâmetros_ )
{
    _corpo_da_função_
}
```

Para cada função que criamos, escolhemos um *nome*. O nome identifica a função, e um programa em C não pode ter duas funções com o mesmo nome. Uma função pode receber dados de entrada, que são os *parâmetros da função*, especificados entre os parênteses que seguem o nome da função, dentro de um mesmo escopo. Se uma função não recebe parâmetros, colocamos a palavra `void` entre os parênteses. Uma função também pode ter um *valor de retorno* associado. Antes do nome da função, identificamos o tipo do valor de retorno. Se a função não tem valor de retorno associado, usamos a palavra `void`. Entre o abre e fecha chaves, codificamos o *corpo da função*, que consiste no bloco de comandos que compõem a função.

Para ilustrar, vamos considerar novamente o programa que converte uma temperatura dada em graus Celsius para graus Fahrenheit. Anteriormente, implementamos este programa codificando tudo dentro da função `main`. Para este exemplo, é fácil identificar uma solução mais estruturada. Podemos dividir nosso programa em duas funções. Uma primeira função fica responsável por fazer a conversão de unidades, enquanto a função `main` fica responsável por capturar o valor fornecido pelo usuário, chamar a função auxiliar que faz a conversão e exibir o valor convertido na tela. Desta forma, identificamos uma funcionalidade bem definida – conversão de Celsius para Fahrenheit – e a codificamos numa função em separado.

Uma grande vantagem de dividir o programa em funções é aumentar seu potencial de reúso. No caso, após nos certificarmos de que a função está correta, podemos usar essa mesma função em qualquer outro programa que precise converter temperatura em graus Celsius para Fahrenheit. Note que, com o uso da função auxiliar, o código da função `main` fica mais simples, facilitando seu entendimento, pois o detalhe de como a conversão é feita está codificado dentro da função auxiliar.

```
#include <stdio.h>

float celsius_fahrenheit (float c)
{
    float f;
    f = 1.8 * c + 32;
    return f;
}

int main (void)
{
    float c;
    float f;
    printf("Entre com temperatura em Celsius: ");
    scanf("%f", &c);
    f = celsius_fahrenheit(c);
    printf("Temperatura em Fahrenheit: %f\n", f);
    return 0;
}
```

A conversão de graus Celsius para Fahrenheit é muito simples, mas é fácil imaginar que esta função *main* não seria alterada se a conversão estivesse baseada em computações sofisticadas, pois na função *main* tudo o que nos interessa é o valor resultante da chamada da função.

No código, as funções devem ser escritas antes de serem usadas. Assim, como a função auxiliar é usada (invocada) pela função *main*, ela deve aparecer antes no código. Na verdade, essa restrição pode ser contornada com o uso de *protótipos*. O protótipo de uma função é o cabeçalho da função, seguido de um ponto-e-vírgula. Por exemplo, o código anterior poderia ser organizado com a função auxiliar após a função *main*:

```
#include <stdio.h>

float celsius_fahrenheit (float c); /* protótipo da função */

int main (void)
{
    ... /* corpo da função principal */
}

float celsius_fahrenheit (float c)
{
    ... /* corpo da função auxiliar */
}
```

A rigor, no protótipo, não há necessidade de indicar os nomes dos parâmetros, apenas os seus tipos; portanto, seria válido escrever:

```
float celsius_fahrenheit (float);
```

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. No exemplo, o compilador verifica se o parâmetro passado é do tipo **float** (ou um valor que possa ser convertido para **float**) e se a chamada da função é usada no contexto em que se espera um valor **float**, que é o tipo do seu valor de retorno. É em razão desta necessidade que se exige a inclusão dos arquivos de interface (**stdio.h**, **math.h** etc.) para a utilização das funções das bibliotecas de C. Nestes arquivos encontram-se, entre outras coisas, os protótipos das funções correspondentes.

É importante saber que a execução do programa sempre se inicia com o código da função *main*, independente da ordem em que escrevemos as funções no nosso código. Os números acrescidos à direita do código mostrado, em forma de comentários de C, ilustram a ordem de avaliação dos comandos quando o programa é executado. Quando invocamos uma função, como no comando **f = celsius_fahrenheit (c);**, a CPU passa a executar os comandos da função *chamada* (**celsius_fahrenheit**, no caso). Quando a função chamada retorna, a CPU prossegue a execução dos comandos da função *que chama* (*main*, no caso). Isso também ocorre quando se chama uma função de uma biblioteca externa.

4.2 Parâmetros e valor de retorno

Uma função deve ter sua interface bem definida, tanto do ponto de vista semântico como do sintático. Do ponto de vista semântico, quando projetamos uma função, identificamos sua funcionalidade e, com isso, definimos que dados de entrada são recebidos e qual o resultado (dado de saída) é produzido pela função. Do ponto de vista sintático, os tipos dos dados de entrada e saída são especificados no cabeçalho da função. Uma função pode receber zero ou mais valores de entrada, chamados *parâmetros da função*, e pode resultar em zero ou um valor, chamado *valor de retorno da função*. Os parâmetros de uma função são listados entre os parênteses que seguem o nome da função. Assim, a função `celsius_fahrenheit` do programa anterior, só recebe um único parâmetro do tipo `float`, ao qual foi associado o nome `c`. Ela tem como retorno um valor do tipo `float`, indicado pelo tipo que precede o nome da função. Note o cabeçalho da função `main`. Esta é uma função que tem como valor de retorno um inteiro e não recebe parâmetros, indicado pela palavra `void` entre os parênteses.

Funções podem receber mais do que um parâmetro. Para ilustrar, considere uma função que calcule o volume de um cilindro de raio r e altura h . Sabe-se que o volume de um cilindro é dado por $\pi r^2 h$. Uma função para calcular o volume de um cilindro deve receber os valores do raio e da altura como parâmetros e ter como retorno o valor do volume calculado. Uma possível implementação dessa função é mostrada a seguir:

```
#define PI 3.14159

float volume_cilindro (float r, float h)
{
    float v;
    v = PI * r * r * h;
    return v;
}
```

Neste exemplo, note a definição da constante simbólica `PI`; esta é uma diretiva de pré-processamento: todas as ocorrências de `PI` são substituídas pelo valor `3.14159` antes do código ser submetido à compilação.

Para testar esta função, podemos escrever uma função `main`. Nessa função, os valores do raio e da altura são capturados do teclado e o volume computado é exibido na tela. Esta função `main` é muito similar à função `main` do exemplo de conversão de temperaturas: capture-se valores fornecidos via teclado, invoca-se a função que fará a computação desejada e, então, exibe-se os resultados.

```
int main (void)
{
    float raio, altura, volume;
    printf("Entre com o valor do raio e da altura:");
    scanf("%f %f", &raio, &altura);

    volume = volume_cilindro(raio, altura);

    printf("Volume do cilindro = %f\n", volume);
    return 0;
}
```

Note que os valores passados na chamada da função devem corresponder aos parâmetros em ordem, número e tipo. No caso, a função `volume_cilindro` espera receber dois parâmetros do tipo `float`: o primeiro, com o valor do raio, e o segundo, com o valor da altura. A chamada da função passa dois valores para a função: os valores armazenados nas variáveis `raio` e `altura`, ambas do tipo `float`. O valor retornado pela função é armazenado na variável `volume`, também do tipo `float`.

Note ainda que a chamada de uma função que tem um valor de retorno associado é uma expressão que, quando avaliada, resulta no valor retornado. Assim, uma chamada de função pode aparecer dentro de uma expressão maior. Por exemplo, se quiséssemos calcular o volume de metade do cilindro, poderíamos ter escrito:

```
volume = volume_cilindro(raio, altura) / 2.0;
```

Por fim, é importante notar que, na chamada da função, passa-se *valores* para a função chamada. Assim, qualquer expressão pode ser usada, desde que resulte num valor do tipo esperado. Por exemplo, é válido a chamada de função a seguir, que pede para calcular o volume do cilindro com altura dobrada em relação ao valor entrado pelo usuário:

```
volume = volume_cilindro(raio, 2*altura);
```

4.3 Escopo de variáveis

Conforme discutido, requisitamos espaços de memória para o armazenamento de valores através da declaração de variáveis. Quando, durante a execução do código, é encontrada uma declaração de variável, o sistema reserva o espaço de memória correspondente. Este espaço de memória permanece disponível para o programa durante o *tempo de vida* da variável. Uma variável declarada dentro de uma função é chamada *variável local*. Uma variável local tem seu tempo de vida definido pela função que a declarou: a variável existe durante a execução da função. Assim que a função retorna, os espaços de memória reservados para suas variáveis locais são liberados para outros usos, e o programa não pode mais acessar estes espaços. Por esse motivo, as variáveis locais são também classificadas como *variáveis automáticas*. Note que uma função pode ser chamada diversas vezes. Para cada execução da função, os espaços das variáveis locais são automaticamente reservados e liberados ao final de cada execução.

As variáveis locais são “visíveis” (isto é, podem ser acessadas) apenas após sua declaração e dentro da função em que foram definidas. Dizemos que o *escopo* da variável local é a função que a define. Dentro de uma função não se tem acesso a variáveis locais definidas em outras funções. Os *parâmetros de uma função* também são variáveis automáticas com escopo dentro da função. Essas variáveis são inicializadas com os valores passados na chamada da função. Como as variáveis locais, os parâmetros representam variáveis que vivem enquanto a função está sendo executada.

Como vimos, variáveis também podem ser declaradas dentro de blocos. A regra é a mesma: a variável só existe enquanto o bloco estiver sendo executado. Se um bloco é definido dentro de uma função, dentro do bloco, tem-se acesso às variáveis locais da função declaradas antes do bloco; no entanto, as variáveis declaradas no bloco não podem ser acessadas na função fora do bloco que as declara.

É importante frisar que as variáveis locais têm escopo dentro da função que as declara. Por exemplo, podemos escolher os nomes `raio` e `altura` para os parâmetros da função `volume_cilindro` (em vez de `r` e `h`). Isso em nada alteraria o funcionamento do programa. Apesar de os nomes das variáveis serem iguais aos da função `main`, são variáveis distintas, alocadas em diferentes áreas de memória. Cada variável está dentro do contexto da função que a declara.

4.3.1 Modelo de pilha

Esta seção explica os detalhes de como os valores são passados para uma função. Essa explicação pode ajudar o leitor a entender melhor a alocação de variáveis automáticas durante a execução de um programa.

A alocação dos espaços de memória dos parâmetros e das variáveis locais seguem um *modelo de pilha*. Podemos fazer uma analogia com uma pilha de pratos. Quando queremos adicionar um prato na pilha, este prato só pode ser colocado no topo da pilha. Se quisermos tirar pratos da pilha, só podemos tirar os que estão no topo da pilha. O sistema gerencia a memória das variáveis automáticas da mesma maneira. Uma área da memória do computador é reservada para armazenar a *pilha de execução* do programa. Quando a declaração de uma variável local é encontrada, o espaço de memória no topo da pilha de execução é associado à variável. Quando o tempo de vida da variável se extingue, o espaço correspondente do topo da pilha é liberado para ser usado por outra variável.

Para ilustrar o modelo de pilha, vamos usar um esquema representativo da memória do computador usado para armazenar a pilha de execução. A Figura 4.1 ilustra a alocação de variáveis automáticas na pilha durante a execução do programa mostrado, que faz o cálculo do volume de um cilindro.

Quando a execução do programa se inicia, a função `main` começa sua execução. No nosso exemplo, no início da função, três variáveis são declaradas (Figura 4.1(a)). As três variáveis são então alocadas no topo da pilha (inicialmente vazia). Como nenhum valor é atribuído às variáveis, os espaços de memória correspondentes armazenam valores indefinidos (“lixos”). Em seguida, o programa requisita que o usuário defina os valores do raio e da altura do cilindro. Estes valores capturados são atribuídos às variáveis correspondentes (Figura 4.1(b)). Após a captura dos valores, a função `main` invoca a função auxiliar para realizar o cálculo do volume. A chamada da função representa primeiramente uma transferência do fluxo de execução para a função. Os parâmetros da função são alocados na pilha e seus valores, inicializados com os valores passados na chamada da função (Figura 4.1(c)). Note que, neste momento, como o controle da execução foi transferido para a função auxiliar, não se tem acesso às variáveis declaradas dentro da função `main`, apesar delas estarem alocadas na base da pilha (a função `main` ainda não terminou, apenas teve sua execução suspensa). Em seguida, dentro da função auxiliar, uma variável local adicional é declarada, sendo alocada no topo da pilha de execução, inicialmente com valor indefinido (Figura 4.1(d)). A função auxiliar então computa o valor do volume e o atribui à variável local (Figura 4.1(e)). Este valor é então retornado pela função. Neste momento, a função auxiliar termina sua execução e o controle volta para a função `main`. Os parâmetros e variáveis locais da função auxiliar são desempilhados e deixam de existir. Por fim, já no retorno do controle para a função `main`, o valor retornado pela função é atribuído à variável `volume` (Figura 4.1(f)).

Este exemplo simples serve para ilustrar o funcionamento do modelo de pilha. Logicamente, programas reais são mais complexos. Em geral, um programa tem várias funções auxiliares. Funções auxiliares podem, evidentemente, chamar outras funções auxiliares, sempre seguindo o modelo de pilha. Ao término da execução de cada função, as variáveis locais correspondentes são desempilhadas e o controle volta para a função que chamou.

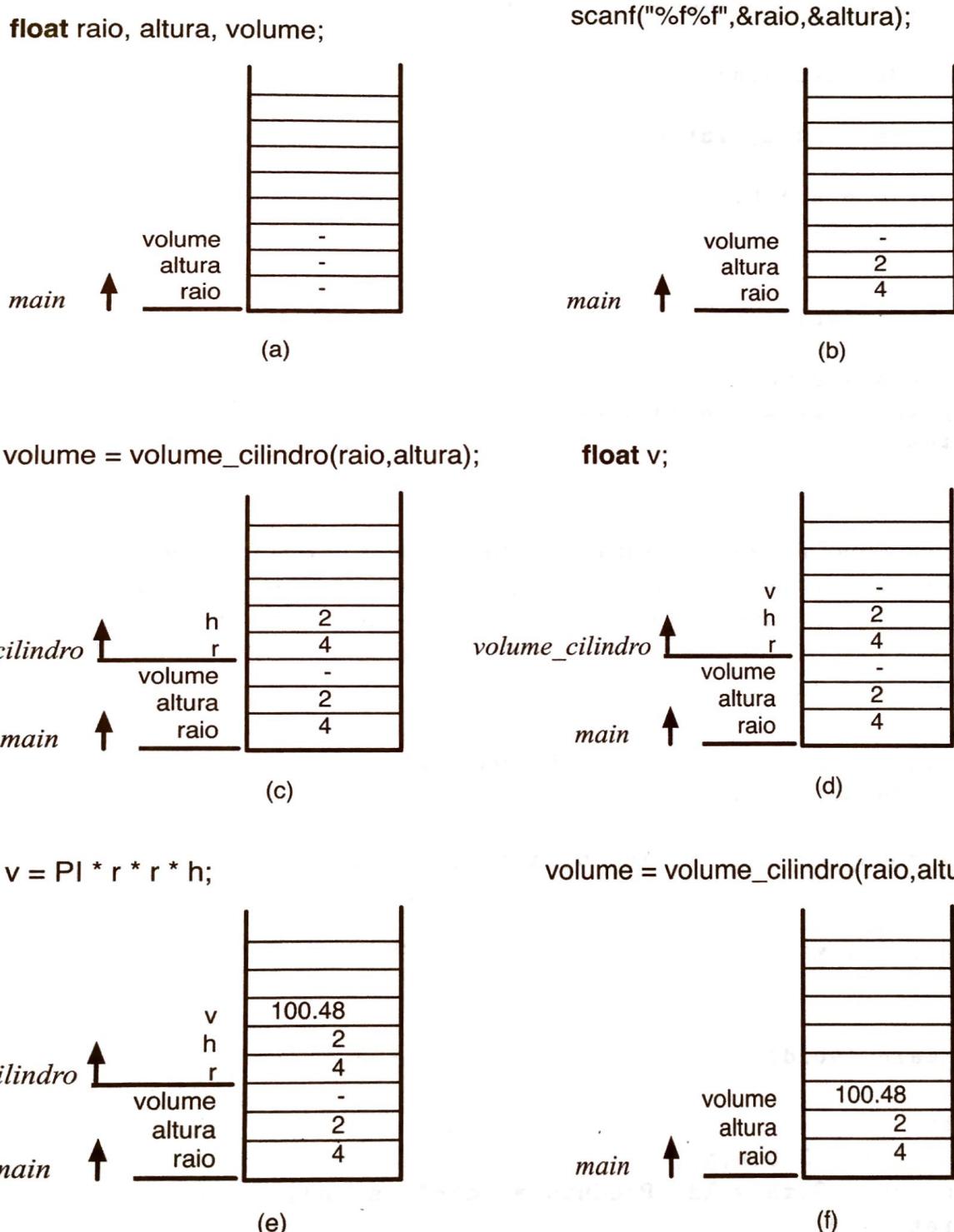


Figura 4.1: Pilha de variáveis durante execução de um programa.

4.4 Ponteiro de variáveis

Conforme ilustrado, uma função pode retornar um valor para a função que chama. No entanto, a possibilidade de retornar um único valor nem sempre é satisfatória. Muitas vezes, precisamos transferir mais do que um valor para a função que chama, o que não pode ser feito via retorno explícito de valores.

Para ilustrar essa discussão, vamos inicialmente considerar uma função muito simples que calcula a soma de dois valores inteiros. Uma implementação dessa função e um exemplo de seu uso são mostrados a seguir:

```
#include <stdio.h>

int soma (int a, int b)
{
    int c = a + b;
    return c;
}

int main (void)
{
    int s = soma(3,5);
    printf("Soma = %d\n", s);
    return 0;
}
```

Este exemplo não apresenta nenhuma dificuldade, pois o resultado da soma pode ser retornado explicitamente. O problema aparece quando desejamos que a função resulte em mais do que um valor. Por exemplo, vamos considerar que queiramos agora fazer uma função para calcular a soma e o produto de dois números. Uma forma *INCORRETA* de implementá-la é ilustrada a seguir:

```
/* função somaprod (versão ERRADA) */
#include <stdio.h>

void somaprod (int a, int b, int c, int d)
{
    c = a + b;
    d = a * b;
}

int main (void)
{
    int s, p;
    somaprod(3,5,s,p);
    printf("Soma = %d  Produto = %d\n", s, p);
    return 0;
}
```

Como sabemos, esse código não funciona como esperado. Serão impressos valores “lixo”, pois *s* e *p* não foram inicializadas na função *main* e seus valores não são alterados.

Alterados são os valores de `c` e `d` dentro da função `somaprod`, mas eles não representam as variáveis da função `main`, sendo apenas inicializadas com os valores de `s` e `p` na chamada da função. (Lembre-se que o comportamento não seria alterado mesmo que os parâmetros tivessem os mesmos nomes, `s` e `p`, das variáveis da função `main`.)

Como fazemos então para que a função que chama tenha acesso aos dois valores calculados? Para resolver esse problema em C, precisamos entender antes o conceito de *ponteiro para variáveis*.

4.4.1 Variáveis do tipo ponteiro

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 nos quais existem valores do tipo correspondente armazenados. Por exemplo, quando escrevemos:

```
int a;
```

declaramos uma variável de 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 nos quais há valores inteiros armazenados. A linguagem não reserva uma palavra especial para a declaração de ponteiros; usamos a mesma palavra do tipo com os nomes das variáveis precedidos pelo caractere `*`. Assim, podemos escrever:

```
int *p;
```

Neste caso, declaramos uma variável de nome `p` que pode armazenar endereços de memória nos quais existe um inteiro armazenado. Para atribuir e acessar endereços de memória, a linguagem oferece dois operadores unários que ainda não foram discutidos aqui. 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 este trecho de código:

```
int a;           /* variável do tipo inteiro */
int *p;          /* variável do tipo ponteiro para inteiro */

a = 5;           /* a recebe o valor 5 */
p = &a;           /* p recebe o endereço de a */

*p = 6;          /* o conteúdo de p recebe */
```

A Figura 4.2 ilustra o que ocorre na pilha de execução se este trecho de código for executado. Nesta figura, os números à direita são valores fictícios dos espaços de memória.

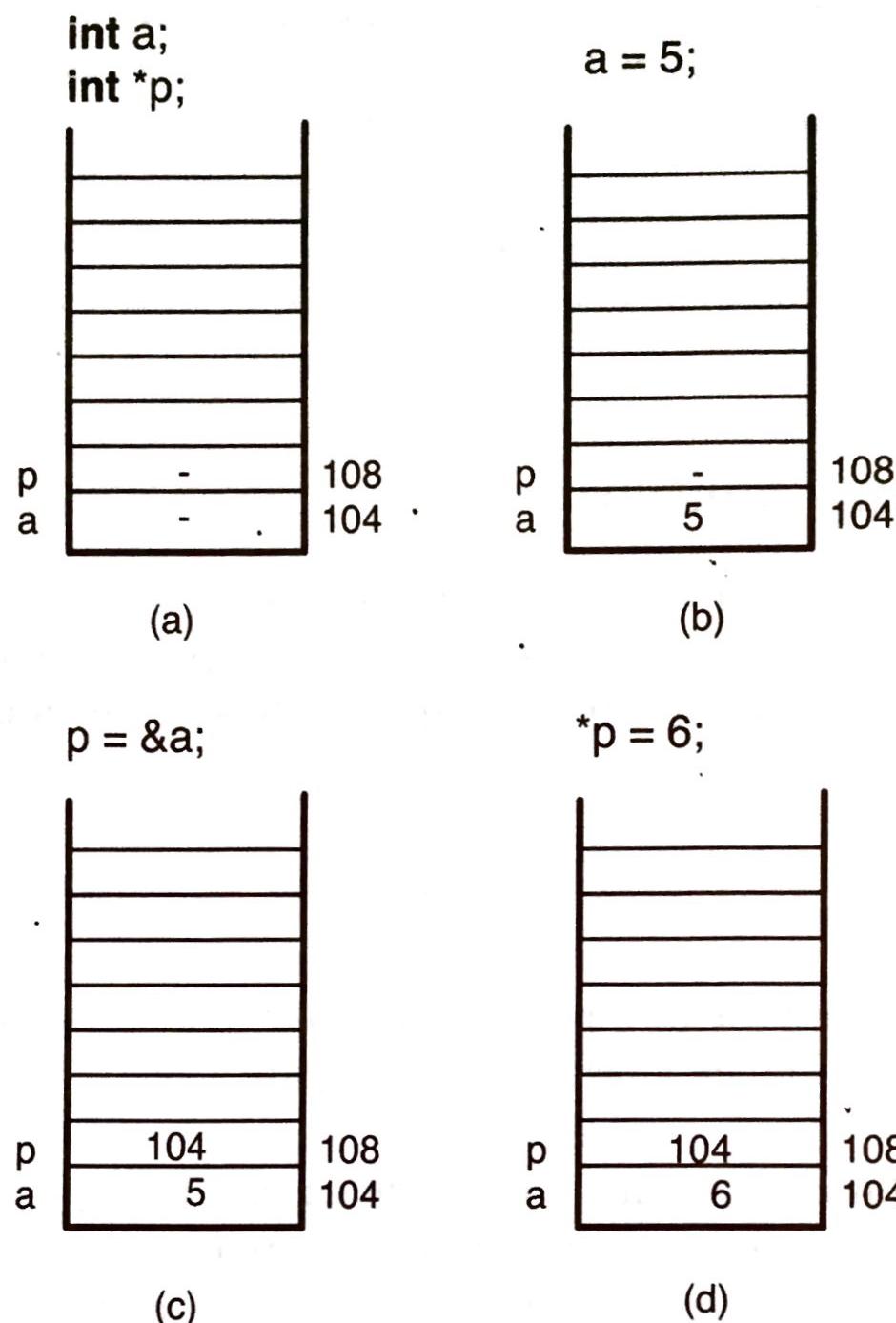


Figura 4.2: Variável ponteiro na pilha de execução.

No início, após as declarações, ambas as variáveis, *a* e *p*, armazenam valores “lixo”, pois não foram inicializadas. No final, a variável *a* terá valor igual a 6, porque a última atribuição alterou indiretamente seu valor. Acessar *a* é equivalente a acessar **p*, pois *p* armazena o endereço de *a*. Dizemos que *p* aponta para *a*, daí o nome ponteiro.

Note que a variável do tipo ponteiro é uma variável como outra qualquer; sua particularidade é que só pode armazenar endereços de memória do tipo correspondente (da mesma forma que uma variável do tipo *int* só pode armazenar valores inteiros). Em vez de criar valores fictícios para os endereços de memória no nosso esquema ilustrativo da memória, podemos desenhar setas graficamente, sinalizando que uma variável do tipo ponteiro aponta para determinada área de memória, como ilustra a Figura 4.3.

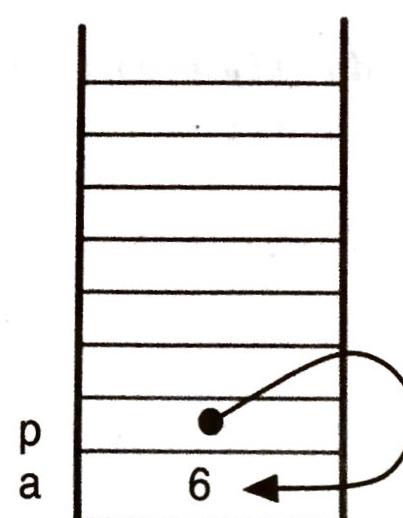


Figura 4.3: Representação gráfica do valor de um ponteiro.

A possibilidade de manipular ponteiros de variáveis é uma das maiores potencialidades de C. Por outro lado, o uso indevido desta manipulação é uma das principais causas 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:

```
int main (void)
{
    int a;
    int *p;
    p = &a;
    *p = 2;
    printf("%d\n", a);
    return 0;
}
```

imprime o valor 2.

Agora, em:

```
/* código com ERRO */
int main (void)
{
    int a, b, *p;
    a = 2;
    *p = 3;
    b = a + (*p);
    printf("%d\n", 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. A variável **p** não foi inicializada e, portanto, tem 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, causando efeitos colaterais imprevisíveis.

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 para o qual já se prevê o armazenamento de valores do tipo em questão.

De maneira análoga, podemos declarar ponteiros de outros tipos:

```
float *q;
double *r;
char *s;
```

4.4.2 Passando ponteiros para funções

Os ponteiros oferecem meios de alterar 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 em que suas variáveis estão armazenadas, essa função pode alterar, indiretamente, os valores das variáveis da função que a chamou.

Para ilustrar a discussão, vamos retomar o exemplo da função para calcular a soma e o produto de dois números inteiros. A solução para esse problema é fazer com que a função `somaprod` receba os endereços das variáveis da função `main` e, assim, altere seus valores indiretamente. A seguir, é ilustrada uma implementação usando essa estratégia:

```
/* função somaprod (versão CORRETA) */
#include <stdio.h>

void somaprod (int a, int b, int *p, int *q)
{
    *p = a + b;
    *q = a * b;
}

int main (void)
{
    int s, p;
    somaprod(3,5,&s,&p);
    printf("Soma = %d Produto = %d\n", s, p);
    return 0;
}
```

Devemos notar que a função `somaprod` anterior não retorna explicitamente nenhum valor (é uma função do tipo `void`). No entanto, ela recebe o endereço de duas variáveis e armazena os valores calculados nesses endereços de memória, alterando, por conseguinte, os valores das variáveis originais. A Figura 4.4 ilustra a execução deste programa, mostrando o uso da memória.

Como exemplo adicional, podemos considerar uma função que troca os valores entre duas variáveis dadas. Para que os valores das variáveis da função principal sejam alterados (trocados) dentro da função auxiliar que faz a troca, precisamos passar para a função os endereços das variáveis. O código a seguir ilustra essa implementação.

```
/* função troca */
#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;
}

```

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. De fato, sempre que desejarmos alterar um valor de uma variável da função que chama dentro da função que é chamada, devemos passar o endereço da variável. Assim, a função chamada tem acesso ao espaço de memória da variável e pode alterar seu valor.

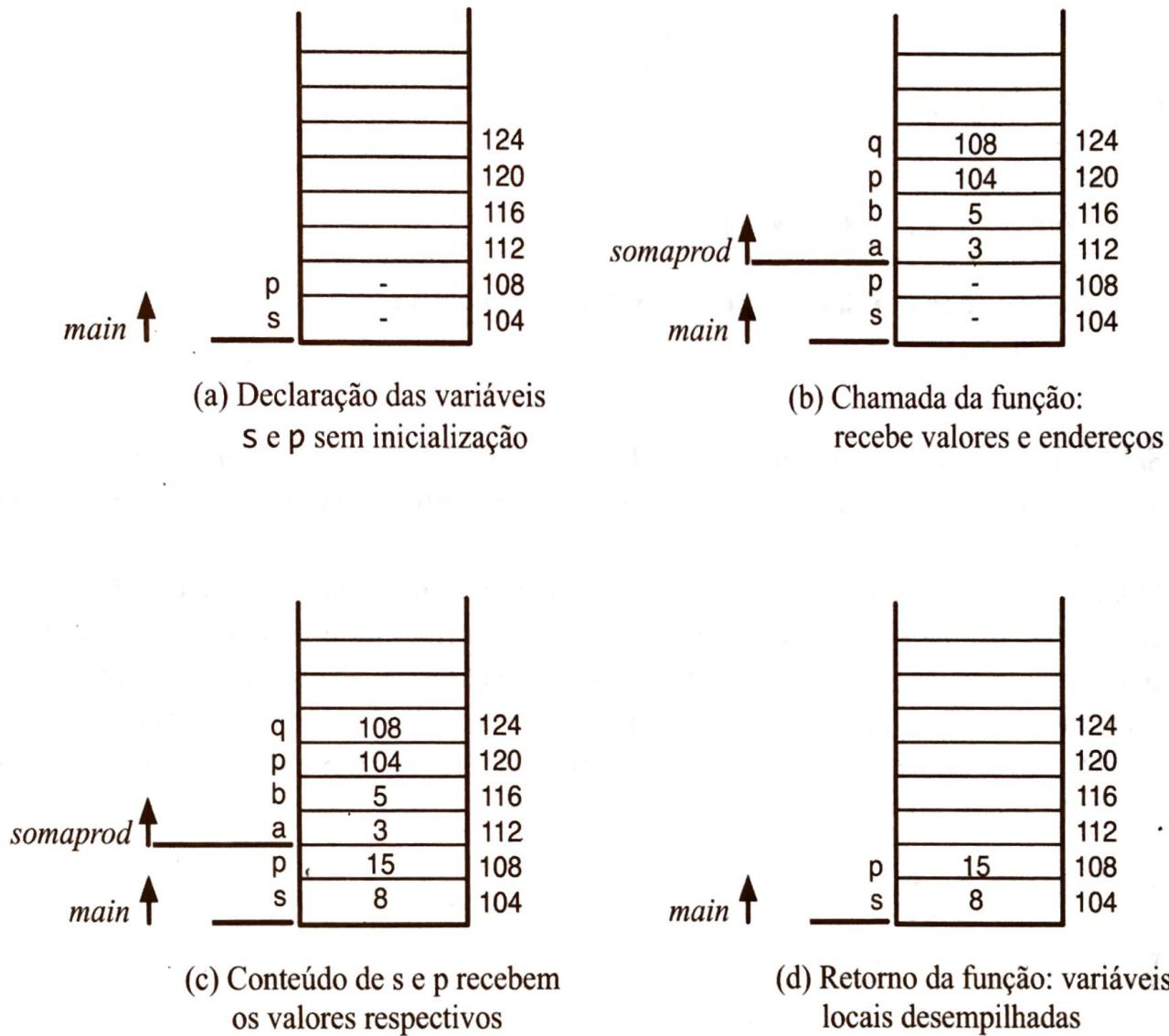


Figura 4.4: Passo a passo da função que calcula soma e produto.

4.5 Variáveis globais e estáticas

Existe outra forma de fazer a comunicação entre funções, que consiste no uso de variáveis globais. Se uma variável é declarada fora do corpo das funções, ela é dita global. Uma variável global é visível a todas as funções subsequentes. As variáveis globais não são

armazenadas na pilha de execução, portanto não deixam de existir quando a execução de uma função termina; elas existem enquanto o programa estiver sendo executado.

Se uma variável global é visível em duas funções, ambas as funções podem acessar e/ou alterar o valor desta variável diretamente. Por exemplo, podemos reescrever o código para cálculo da soma e do produto entre valores fazendo uso de variáveis globais para armazenar os resultados:

```
#include <stdio.h>

int s, p; /* variáveis globais */

void somaprod (int a, int b)
{
    s = a + b;
    p = a * b;
}

int main (void)
{
    int x, y;
    scanf ("%d %d", &x, &y);
    somaprod(x, y);
    printf ("Soma = %d produto = %d\n", s, p);
    return 0;
}
```

Salientamos, no entanto, que o uso de variáveis globais em um programa deve ser feito com critério, pois podemos criar um alto grau de interdependência entre as funções, o que dificulta o entendimento e o reúso do código.

Quando começarmos a escrever programas maiores, nos quais o código é distribuído em diferentes arquivos, o uso de variáveis globais se torna ainda menos recomendado. O nome de uma variável global representa um símbolo externo, acessível em qualquer outro arquivo que compõe o programa. É fácil perder o controle de acessos a variáveis globais em programas grandes. Para amenizar, existe a possibilidade de declarar uma variável global como sendo estática:

```
static int s, p;

void somaprod (int a, int b)
{
    ...
}

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

Neste caso, a variável global é visível apenas dentro do arquivo que a declara. O nome da variável não é exportado para eventuais arquivos adicionais que compõem o programa. Mesmo assim, deve ser usado com critério. Nos nossos exemplos, vamos buscar não usar variáveis globais.

De maneira análoga, as funções também podem ser declaradas como sendo estáticas, não podendo ser acessadas (chamadas) por funções definidas em outros arquivos. Esta é uma prática de programação obrigatória quando estamos desenvolvendo programas com vários arquivos: funções auxiliares usadas apenas dentro de um arquivo devem ser declaradas como estáticas. Com isso, evitamos o acesso indesejável à função e preservamos o ambiente global (o nome da função não é um símbolo exportado).

Podemos também declarar variáveis estáticas dentro de funções. Essas variáveis também 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, diferentemente das variáveis globais, 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 isso, do primeiro ao quinto número são impressos na primeira linha, do sexto ao décimo na segunda e assim por diante. Uma possível implementação desta função é mostrada a seguir:

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

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

Neste código, a variável estática foi inicializada com o valor 1. Se omitirmos a inicialização de uma variável estática, ela é automaticamente inicializada com o valor zero. Isso também vale para variáveis globais. Neste nosso código, então, na primeira vez que a função for executada, a variável `n` terá valor 1, sendo incrementado para 2. Na segunda vez que a função for executada, o valor de `n` será inicialmente 2 (preservado desde a última execução da função) e assim por diante. Sempre que `n` assume um valor múltiplo de 5, o código inclui uma mudança de linha na saída.

4.6 Recursão

Na linguagem C, como na maioria das linguagens de programação, 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.

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 diferente para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivéssemos chamando funções diferentes.

As implementações que usam recursão devem garantir que existe uma *condição de contorno* para evitar que o procedimento não tenha fim. Por exemplo, é fácil perceber que a função:

```
/* Exemplo de ERRO de programação */
void func (int n)
{
    printf ("%d\n", n);
    func(n);
    printf ("* ");
}
```

está errada. Se um procedimento invocar esta função, o programa não termina sua execução, pois `func` chama `func`, que chama `func`, indefinidamente. O número passado aparece na saída várias vezes, até que a pilha de execução chega ao seu limite e o programa é interrompido com erro. Note que a linha que imprime o asterisco nunca é executada. A seguir, um exemplo de função recursiva codificada corretamente (considerando que ela recebe como parâmetro um número inteiro positivo):

```
/* Exemplo correto de recursão */
void func (int n)
{
    printf ("%d ", n);
    if (n > 0) {
        func(n-1);
        printf ("* ");
    }
}
```

Neste caso, impomos uma condição de contorno: a chamada recursiva só é feita se o valor de `n` for positivo. Como o valor de `n` é decrementado a cada chamada da função (e considerando que o valor inicial é positivo), garantimos que a execução do procedimento tem fim. Se esta função for chamada passando o valor 4 (`func (4);`), a saída será:

4 3 2 1 0 * * *

Vamos analisar como fica a pilha de execução durante a execução deste procedimento. Lembre-se que cada chamada da função cria um ambiente local independente. A função é inicialmente invocada com `func (4)`, que exibe o valor 4 e chama `func (3)`, que por sua vez chama `func (2)`, que chama `func (1)`, que chama `func (0)`, que retorna sem chamar a função recursivamente, pois o valor de `n` não é maior do que zero. Até aqui, a saída é composta por 4 3 2 1 0. Quando a chamada de `func (0)` retorna, a execução retorna para o contexto da execução de `func (1)`, que após a chamada recursiva, exibe um asterisco na tela, e retorna. A execução então retorna para o contexto de `func (2)` que

também imprime um asterisco e retorna, e assim por diante, até que o fim da execução de `func(4)` retorna para o contexto da função externa que invocou `func` pela primeira vez. A Figura 4.5 ilustra a pilha de execução deste procedimento. Note que cada contexto tem sua variável `n`, que é independente.

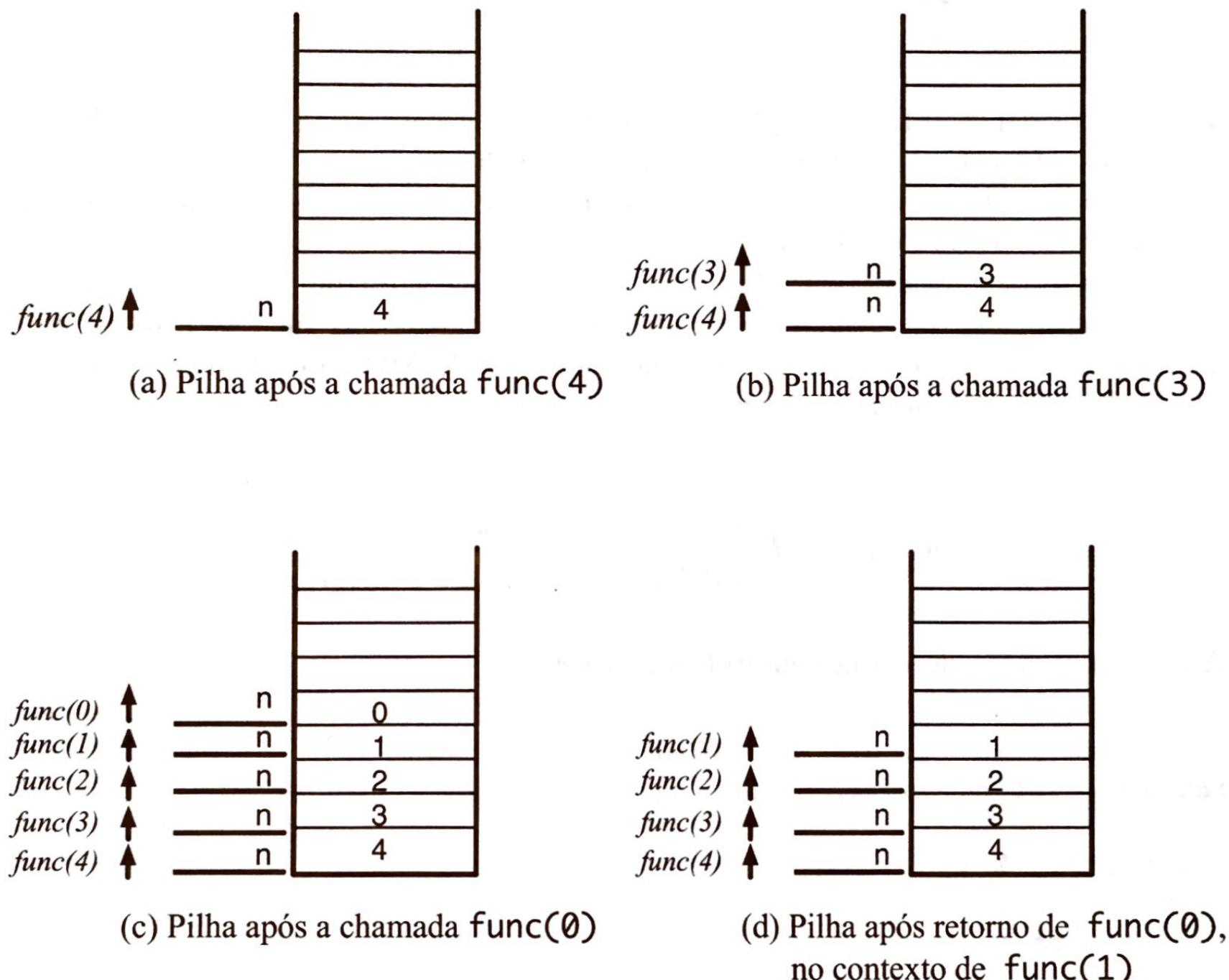


Figura 4.5: Pilha de execução em chamadas recursivas.

Em geral, as implementações recursivas são pensadas considerando-se a definição recursiva do problema que desejamos resolver, lembrando sempre que é fundamental o tratamento da condição de contorno. Como exemplo, podemos considerar o cálculo do fatorial de um número inteiro positivo. O valor do fatorial pode ser definido por:

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

Considerando essa definição recursiva, fica muito simples pensar na implementação recursiva de uma função que calcula e retorna o fatorial de um número; basta traduzir a definição para a linguagem C:

```
/* Função recursiva para cálculo do factorial */
int fat (int n)
{
    if (n==0)
        return 1;
    else
        return n*fat(n-1);
}
```

Vamos considerar outro exemplo: determinar o *máximo divisor comum* (MDC) de dois números. O valor do máximo divisor comum de dois números inteiros positivos, $MDC(x,y)$, pode ser calculado usando o algoritmo de Euclides. Este algoritmo é baseado no fato de que se o resto da divisão de x por y , representado por r , for igual a zero, y é o MDC. Se o resto r for diferente de zero, o MDC de x e y é igual ao MDC de y e r . O processo se repete até que o valor do resto da divisão seja zero, o que garantidamente irá acontecer pois, no caso extremo, chegaremos ao valor do MDC de um valor n e 1, que vale 1. Podemos então definir o $MDC(x,y)$ como sendo:

$$MDC(x,y) = \begin{cases} y & \text{se } x \% y = 0 \\ MDC(y, x \% y) & \text{caso contrário} \end{cases}$$

A implementação do código então fica simples:

```
/* Função para cálculo do máximo divisor comum */
int mdc (int x, int y)
{
    int r = x % y;
    if (r == 0)
        return y;
    else
        return mdc(y, r);
}
```

Note que não é necessário associar o maior número a x na chamada da função. Se inicialmente o valor de x for menor que o valor de y , o algoritmo automaticamente corrige-os na primeira chamada recursiva, pois o resto da divisão de um número menor por um número maior será sempre o número menor.

4.7 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 tipicamente procura o arquivo primeiro no diretório local (em geral, denominado diretório de trabalho) 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 local (os arquivos da biblioteca padrão de C devem ser incluídos com <>).

Outra diretiva de pré-processamento muito utilizada é a diretiva de definição. Já a usamos na sua forma mais simples:

```
#define PI 3.14159F

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

Neste caso, antes da compilação, toda ocorrência do símbolo PI (desde que não envolvida por aspas) será trocada por 3.14159 F. O uso de diretivas de definição para representar constantes simbólicas é fortemente recomendável, pois facilita a manutenção e acrescenta clareza ao código.

A linguagem 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));
```

Essas definições com parâmetros recebem o nome de *macros*. Devemos ser cuidadosos 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:

```
/* Código com definição de macro INADEQUADA */
#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:

```
#define DIF(a,b) ((a)-(b))
```

Porém, esse outro exemplo também ilustra a definição inadequada de macro, mesmo com o uso de parênteses envolvendo a definição:

```
/* Código com definição de macro também INADEQUADA */
#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 (substitua a definição da macro no código e entenda o motivo). 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, além da macro como um todo, com parênteses.

Exercícios

1. Escreva um programa, estruturado em diversas funções, para calcular o volume de uma peça formada por uma esfera com um furo cilíndrico, conforme ilustrado na Figura 4.6. Sabe-se que o volume de uma calota esférica de altura h é dada por $\frac{1}{3}\pi h^2(3R - h)$, onde R representa o raio da esfera.
2. O máximo divisor comum de três números inteiros positivos, $MDC(x,y,z)$, é igual a $MDC(MDC(x,y),z)$. Escreva um programa que capture três números inteiros fornecidos via teclado e imprima o MDC deles, usando a função MDC apresentada no texto.
3. Reescreva o programa que calcula as raízes de uma equação do segundo grau do capítulo anterior. Nesta nova versão, escreva uma função que calcula as raízes da equação. A função deve retornar o número de raízes reais existentes e preencher os endereços passados com os valores das raízes correspondentes, seguindo o protótipo:

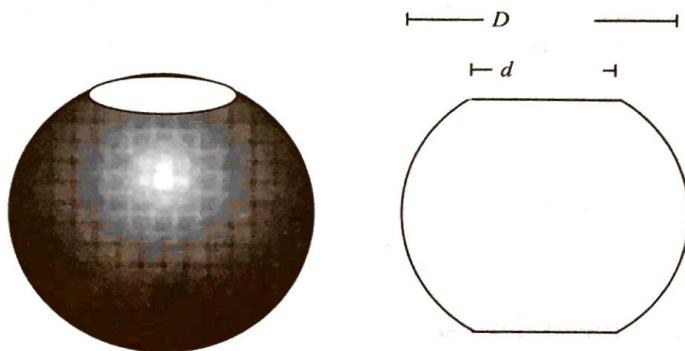


Figura 4.6: Modelo de uma esfera com furo cilíndrico: vistas 3D e 2D.

```
int raízes (double a, double b, double c, double* px1, double* px2);
```

A função *main* deve capturar os valores dos coeficientes, fornecidos via teclado, chamar a função e exibir os resultados.

4. Escreva uma função de potenciação recursiva, considerando o expoente como sendo um valor inteiro positivo ($x^k, k > 0$). A função deve seguir o protótipo:

```
double pot (double x, int k);
```

Escreva uma função *main* para testar sua implementação; compare o resultado da sua função com o valor retornado pela função de potenciação *pow*, existente na biblioteca de matemática padrão de C.

5. A série de Fibonacci é formada pela seguinte sequência de números:

$$0, 1, 2, 3, 5, 8, 13, 21, \dots$$

O primeiro número da série é 0, o segundo é 1 e os demais são a soma dos dois números anteriores. Podemos então definir o valor de um termo da série de Fibonacci como sendo:

$$F(i) = \begin{cases} 0 & \text{se } i = 0 \\ 1 & \text{se } i = 1 \\ F(i - 2) + F(i - 1) & \text{se } i > 1 \end{cases}$$

Escreva uma função recursiva que retorna o valor do i -ésimo termos da série de Fibonacci. Faça uma função *main* para imprimir os primeiros 13 termos da série de Fibonacci. Note que a implementação recursiva para a determinação de um termo da série de Fibonacci não é eficiente computacionalmente: um mesmo termo é avaliado múltiplas vezes. Por exemplo, $F(3)$ é avaliado 55 vezes na avaliação do termo $F(13)$!

6. Implemente uma função que receba como parâmetros dois números inteiros n e k , e calcule o coeficiente binomial $C(n, k)$ correspondente, definido pela seguinte relação recursiva:

$$\begin{aligned} C(n, 0) &= C(n, n) = 1 && \text{para } n \geq 0 \\ C(n, k) &= C(n - 1, k) + C(n - 1, k - 1) && \text{para } n > k > 0 \end{aligned}$$

O protótipo dessa função deve ser:

```
int coeficiente (int n, int k);
```