Programação em C

Um guia para programação em linguagem C, que pretende servir também como uma introdução à Computação. (Baseado, em parte, no curso de Introdução à Computação [MAC-110/115] do IME-USP.)

Este guia é um trabalho em progresso, em constante mutação! Algumas partes ainda estão incompletas ou faltando.

Uma cópia deste guia, em sua versão mais recente, pode ser encontrada na página
`http://fig.if.usp.br/~esdobay/`

EDUARDO S. DOBAY
edudobay@gmail.com
Instituto de Física
Universidade de São Paulo

Versão de 1 de agosto de 2012

Sumário

Prefácio **1**

Introdução **3**

1 Princípios básicos **7**
1.1 O núcleo de um programa 7
1.2 Variáveis 9
1.3 Entrada e saída de dados 11
1.4 Matemática básica 13
1.5 Boas maneiras em C 16

2 Controle de fluxo **19**
2.1 Desvios condicionais: if 19
2.2 Repetindo operações: laços (*loops*) 22
2.3 Contadores e laços for 27
2.4 Condições compostas 31
2.5 Repetições encaixadas 32
2.6 Variáveis booleanas 34

3 Funções **37**
3.1 Introdução 37
3.2 Os parâmetros e o valor de saída 39
3.3 Usando funções 41
3.4 Trabalhando com várias funções 42
3.5 Escopo de variáveis 43

4 Mais sobre números **45**
4.1 Bases de numeração 45
4.2 O armazenamento dos dados 47
4.3 Números reais 50
4.4 Funções matemáticas 58
4.5 O tipo char 61

5 Ponteiros e vetores **63**
5.1 Prolegômenos . . . 63
5.2 Ponteiros como parâmetros de funções . . . 66
5.3 Cuidado com os ponteiros! . . . 68
5.4 Vetores . . . 69
5.5 Vetores como argumentos de funções . . . 72
5.6 Ponteiros e vetores . . . 74
5.7 Strings . . . 75
5.8 Mais sobre entrada e saída . . . 79

6 Algoritmos **81**

7 Mais ponteiros **83**
7.1 Matrizes . . . 83
7.2 Alocação dinâmica de memória . . . 84
7.3 Ponteiros duplos . . . 87
7.4 Ponteiros para funções . . . 88
7.5 Escopo de variáveis . . . 90
7.6 Funções recursivas . . . 90

8 Estruturas **91**
8.1 Structs . . . 91
8.2 Listas ligadas . . . 93

Apêndices

A Compilação **95**

B Tabelas de referência **97**

Referências Bibliográficas **99**

Prefácio

Introdução

Este capítulo introduz alguns conceitos básicos sobre computação, mas sem entrar na linguagem C.

O que é um computador?

A definição mais ampla de computador é a de uma máquina que realiza uma série de operações lógicas ou matemáticas sobre um conjunto de dados e devolve o resultado para o usuário. O computador que conhecemos não é o único exemplo disso; podemos citar diversos aparelhos eletrônicos com essas características, de calculadoras a telefones celulares.

Um computador tem um conjunto de *instruções* que correspondem às operações que podem ser feitas com ele. Uma seqüência lógica dessas instruções — tal qual uma receita de bolo — é o que conhecemos como *programa*.

Nos primeiros computadores, toda a programação era feita diretamente no circuito do aparelho: não se podia modificar o funcionamento do computador sem uma reconstrução física dele ou de alguma de suas partes. Ainda existem computadores desse tipo — é o caso de uma calculadora simples, por exemplo: você não pode criar programas e introduzir nela; é necessário mexer diretamente nas placas de circuito dela.

O computador que conhecemos faz parte de um conjunto especial dentro da nossa definição de computador: nele, os programas são armazenados na *memória* (e não "inseridos" diretamente através dos circuitos), da mesma maneira que os dados, e podem ser criados e modificados utilizando o próprio Computador. A encarnação mais comum desse tipo de funcionamento é um modelo conhecido como *arquitetura de von Neumann*. (A palavra *arquitetura* é usada em geral para descrever o modelo de funcionamento interno de um computador.)

Esse modelo descreve o funcionamento geral do computador em termos da interação entre três componentes:

- uma unidade de processamento — conhecida como CPU (*unidade central de processamento*, na sigla em inglês) ou processador — que é responsável por todas as operações matemáticas e lógicas. É basicamente o "cérebro" do computador;
- um espaço para armazenamento (temporário) de dados, que é a memória RAM (sigla em inglês para *memória de acesso randômico*);

- quaisquer dispositivos de *entrada* e *saída*, que recebem e enviam informações de/para o usuário. Os exemplos mais imediatos são o monitor (dispositivo de saída), o teclado e o mouse (dispositivos de entrada). Também se encaixam aqui os dispositivos de armazenamento de dados, como o disco rígido, que têm papel tanto de entrada quanto de saída.

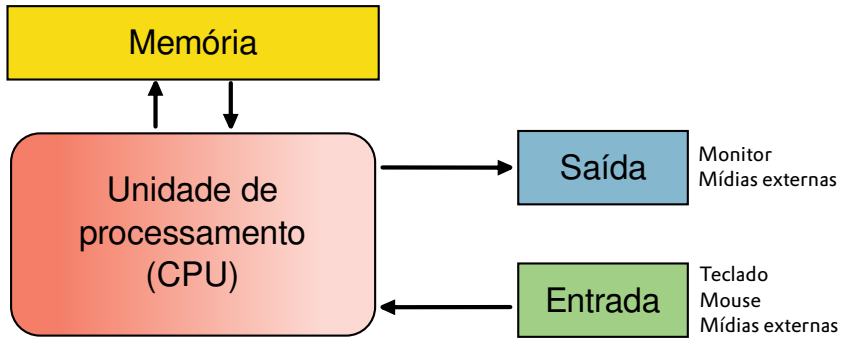


Figura 0.1: Esquema da arquitetura de von Neumann.

A arquitetura de von Neumann é um modelo, de certa forma, abstrato; na prática, para construir um computador, há vários outros detalhes a se estabelecer. Diferentes escolhas desses detalhes, devidas a interesses e propósitos diferentes, levaram a diferentes *arquiteturas* de processadores. Por exemplo, o funcionamento do processador que existe num GameBoy é bem diferente do funcionamento do processador de um computador pessoal. Os computadores pessoais de hoje em dia usam, em sua maioria, processadores baseados numa mesma arquitetura, conhecida como *arquitetura Intel*,[1] e que foi introduzida pela primeira vez em processadores da Intel.

Linguagens de programação

Já vimos que um programa é simplesmente um conjunto de instruções ou comandos que dizem ao computador como realizar as operações que ele deve fazer. Mas como vêm a ser essas instruções? O processador do computador só consegue compreender uma linguagem que chamamos de *linguagem de máquina*; essa linguagem, inclusive, é diferente para cada arquitetura de processadores. Um programa escrito em linguagem de máquina consiste de uma série de *bits* (os famosos zeros e uns), que, para nós, parecem não ter nenhum significado. Uma instrução razoavelmente simples seria escrita assim:

```
10110000 00101010
```

Não seria nada prático usar essa linguagem para fazer programas com alto nível de complexidade; por isso, foram desenvolvidas diversas **linguagens de programação**, que servem como intermediário entre a linguagem humana e a linguagem de máquina.

Uma linguagem de programação deve ser, por um lado, legível e compreensível para um humano, e, por outro lado, suficientemente simples para que possa ser compreendida

[1] Para ser mais preciso, é a *arquitetura Intel de 32 bits*, ou, abreviadamente, IA-32. Mais tarde veremos o que isso significa.

completamente e sem ambigüidades por um computador. Geralmente, uma linguagem de programação é composta de:

- um conjunto de palavras-chave (geralmente em inglês, como `if`, `while`, `function`), associadas a certos comandos e recursos da linguagem;
- um conjunto de símbolos (como `#`, `"`, `{`, etc.); e
- um conjunto de regras que ditam o significado de todas essas coisas e como elas podem ser combinadas e utilizadas.

No entanto, as linguagens de programação não são entendidas diretamente pelo computador (lembre-se, ele só entende a linguagem de máquina). Para que o computador possa executar códigos escritos em linguagens de programação, existem programas que são feitos para entender essas linguagens e traduzir os códigos para a linguagem de máquina. Há dois tipos desses programas: os **compiladores** e os **interpretadores**. O processo de tradução do código para a linguagem de máquina chama-se *compilação*.

Um interpretador lê um programa escrito numa certa linguagem de programação e imediatamente executa as instruções nele contidas — a tradução para a linguagem de máquina é feita na hora da execução. Já um compilador apenas traduz o programa para a linguagem de máquina (*compila* o programa), deixando-o pronto para ser executado posteriormente *sem depender de outro programa*; isso tem a vantagem de tornar a execução do programa mais rápida.

Os programas em linguagem C são sempre compilados. Para aprender a usar um compilador (o que você certamente deve aprender, já que precisará disso para poder executar seus programas), dirija-se ao o Apêndice **??**.

Princípios básicos 1

1.1 O núcleo de um programa

Um programa em C é estruturado em **funções**, que são, basicamente, trechos de código que podem ser chamados várias vezes para realizar uma certa tarefa. Todos os comandos que o seu programa executar estarão dentro de alguma função.

As funções, em geral, podem *devolver valores* — o resultado de algum cálculo ou consulta, por exemplo — e também podem *receber parâmetros* — por exemplo, os dados de entrada para um cálculo, ou os critérios a serem usados na busca. Nesse sentido, funções em C são semelhantes às funções às quais nos referimos em matemática.

Como acabamos de dizer, todos os comandos dum programa devem estar dentro de uma função. Em C, no entanto, existe uma função "privilegiada" — é a função main (*principal*). Ela é como se fosse o roteiro principal do programa: a execução do programa sempre começa por ela. Todas as outras funções (quando houver) são chamadas, direta ou indiretamente, a partir da main.

Todo programa deve ter uma função main. Outras funções podem ser criadas (veremos como um pouco mais adiante), e também podemos usar funções prontas — há uma série de funções que vem disponível em qualquer compilador C, chamada de *biblioteca padrão do C*.

Vamos iniciar nosso estudo com um programa extremamente simples, que apenas imprime[2] uma mensagem na tela:

```
#include <stdio.h>

int main()
{
   printf("Olá, maravilhoso mundo da programação!\n");
   return 0;
}
```

Vamos ver o que acontece em cada linha:

[2]Veja que, aqui, "imprimir" não tem nada a ver com a sua impressora que joga tinta no papel. Nesse livro, sempre que usar essa palavra, entenda como uma referência ao ato de mostrar alguma mensagem na tela.

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

Esta linha pede ao compilador que disponibilize para nós algumas funções de *entrada e saída de dados*, que permitem que você exiba mensagens na tela e leia dados que o usuário digitar no teclado. Mais adiante veremos como é que essa instrução funciona.

```
int main()
```

É aqui que definimos nossa função main. As chaves `{ }` servem para delimitar o seu conteúdo.

Se você está curioso, esse `int` significa que o valor que a função devolve é um número inteiro, e os parênteses vazios indicam que a função não recebe nenhum parâmetro. Não se preocupe se isso não fizer muito sentido; um pouco mais adiante estudaremos funções com detalhe, e você poderá entender melhor o que tudo isso significa. Aceitar isso como uma "fórmula pronta" por enquanto não lhe fará nenhum mal.

```
    printf("Olá, maravilhoso mundo da programação!\n");
```

Aqui printf é o nome uma função da biblioteca padrão (é possível usá-la graças ao #include que colocamos lá em cima!). Essa função simplesmente toma a mensagem (uma sequência de caracteres) que lhe foi passada e mostra-a na tela.

Nessa linha, passamos como parâmetro a essa função a mensagem que queremos imprimir (usando aspas duplas, veja bem). E o que é aquele `\n` intruso no final? Ele é um código especial que representa uma quebra de linha — indicando que qualquer coisa que for impressa depois da nossa mensagem deve ir para a linha seguinte. Se omitíssemos o `\n`, a próxima mensagem que fosse eventualmente impressa sairia grudada nessa; isso é útil em várias situações, mas não particularmente nessa; em geral é melhor terminar a saída de um programa com uma quebra de linha.

Essa coisa entre aspas é chamada de **sequência de caracteres** (adivinhe por quê!), e também é bastante conhecida pelo seu nome em inglês, **string** (literalmente, *cadeia [de caracteres]*). Para falar a verdade, usarei principalmente a palavra *string* daqui em diante.

Note que usamos um **ponto-e-vírgula** aqui. Da mesma maneira que em português usamos um ponto final para encerrar uma frase, em C precisamos usar obrigatoriamente um ponto-e-vírgula para encerrar um comando.

```
    return 0;
```

Essa instrução encerra a execução do programa (por isso, deve ser sempre a última da função main). Além disso, o número zero serve para indicar ao sistema que o programa terminou com sucesso (números diferentes de zero indicariam um erro); é uma convenção da linguagem. Você entenderá melhor como isso funciona quando falarmos detalhadamente de funções, no capítulo 3.

Note novamente o ponto-e-vírgula.

Um comentário adicional sobre a obrigatoriedade do ponto-e-vírgula: veja que ele não foi usado nem com o `#include` nem com a definição da função main. Neste último caso, não se usa ponto-e-vírgula pois a definição da função não é propriamente um comando, mas um bloco com vários comandos.

Já no primeiro caso, o ponto-e-vírgula não é usado porque o `#include` é um comando, de certa maneira, "especial". Por enquanto, apenas guarde isso: linhas começadas com `#` não levam ponto-e-vírgula no final. Veremos mais adiante como funciona esse tipo de comando.

É importante saber que a linguagem C diferencia maiúsculas e minúsculas, ou seja, se você digitar `PRINTF` ou `Printf`, ou trocar `return` por `RETURN`, o programa não irá funcionar.

1.2 Variáveis

Uma das necessidades mais comuns em programação é a de guardar dados em algum lugar da memória. Para isso, a maioria das linguagens de programação usa o conceito de **variáveis**, que são simplesmente pedaços da memória que servem para guardar um certo valor (um número, por exemplo) e que têm um nome. Com isso, você não precisa se preocupar em saber em que lugar da memória você guardará seus dados; todo o trabalho fica para o compilador.

Em C, há três tipos primitivos de dados: **números inteiros**, números de **ponto flutuante** (um nome pomposo para os números fracionários, que tem a ver com a maneira como os computadores trabalham com eles) e **caracteres** (como '`a`', '`#`', '`Z`', '`5`' — o caractere que representa o número 5, e não o número em si). A esses três tipos de dados correspondem os quatro tipos primitivos de variáveis, listados na tabela 1.1.

Na verdade, eu poderia dizer que são apenas *dois* esses tipos: os números inteiros e os de ponto flutuante. O que ocorre é que os caracteres são representados na memória como se fossem números (inteiros); por exemplo, quando eu escrevo os caracteres `abc` em algum lugar (em última instância, isso fica em algum lugar da memória), o que o computador guarda na memória são os três números $97, 98, 99$. Isso está relacionado à conhecida **tabela ASCII**, que nada mais é que uma convenção de quais caracteres correspondem a quais números (consulte o apêndice para ver uma tabela ASCII completa). Por causa disso, o tipo de dados usado para armazenar caracteres também podem ser usado para guardar números, embora isso não seja tão comum, pois ele só permite guardar números pequenos.

Esses dois ou três tipos de dados traduzem-se em quatro tipos primitivos de variáveis, resumidos na tabela 1.1.

Tabela 1.1: Os quatro tipos primitivos de variáveis na linguagem C

Tipo	Utilidade
`int`	Armazena um número inteiro
`float`	Armazena um número de ponto flutuante
`double`	Como float, mas fornece maior precisão
`char`	Guarda um único caractere. Com esse tipo também podemos guardar *sequências* de caracteres (strings), mas isso requer um outro recurso da linguagem que só veremos mais tarde.

Esses tipos são chamados de "primitivos" porque eles podem ser modificados e combinados de certas maneiras, de modo a criar estruturas mais complexas de dados (por exemplo,

as tais sequências de caracteres), como veremos mais tarde. Por enquanto, usaremos apenas os tipos inteiros; trabalharemos com números de ponto flutuante a partir do capítulo **??**.

Cada variável que quisermos usar deve ser "criada" com antecedência; para isso, precisamos dizer o nome e o tipo das variáveis. Para isso, usamos um comando no seguinte formato:

tipo_da_variavel nome_da_variavel;

Esse tipo de comando chama-se **declaração de variáveis**, e faz com que o computador reserve na memória um espaço suficiente para guardar valores do tipo que você pediu. Vejamos alguns exemplos de declarações:

```
int dia;
int mes;
int ano;

float preco_unitario;
float preco_total;
```

Se você tem várias variáveis do mesmo tipo, você pode declará-las todas de uma vez só — simplesmente coloque os vários nomes separados por vírgulas. Poderíamos, assim, condensar o exemplo anterior com apenas duas linhas:

```
int dia, mes, ano;
float preco_unitario, preco_total;
```

Mas atenção: nos nomes de variáveis você não pode usar acentos nem nenhum outro tipo de caractere "especial"; o C só aceita os caracteres alfabéticos de A a Z (minúsculos e maiúsculos), dígitos de 0 a 9 e o traço inferior (_). Há ainda a restrição de que o primeiro caractere não pode ser um número. Além disso, como a linguagem é sensível à diferença entre maiúsculas e minúsculas, a variável `var` é diferente da variável `VAR` (e portanto ambas podem existir ao mesmo tempo).

Após criar variáveis, o próximo passo é aprender a usá-las. Uma das coisas mais simples que podemos fazer é guardar um valor numa variável: usamos um sinal de igual, escrevendo à sua esquerda o nome da variável, e à direita o valor que queremos guardar:

```
dia = 22;
mes = 5;
ano = 2008;
```

Isso é um comando de **atribuição**: ele joga o valor da direita na variável da esquerda. Veja que a posição é importante nesse caso; não faria sentido escrever "`22 = dia`". Você pode pensar no operador `=` como uma flecha que joga o valor da direita na variável à esquerda: `dia` ← `22`.

Os valores que você joga nas variáveis não precisam ser constantes; você pode também usar o conteúdo de uma outra variável:

```
int a, b;
a = 5;
b = a;
```

Na verdade, você pode atribuir a uma variável o valor de uma *expressão* qualquer; por exemplo, expressões aritméticas envolvendo variáveis e constantes, como veremos a seguir.

A primeira atribuição de uma variável costuma ser chamada de **inicialização**. A inicialização de uma variável é muito importante pois, quando você declara uma variável, o lugar da memória que ela ocupa é apenas *reservado* para seu uso, sem atribuir nenhum valor padrão, como 0 ou 23. Ali há inicialmente um valor "aleatório", que foi deixado por algum programa que já usou aquele lugar da memória. Por isso,

> Você não pode tentar acessar o valor de uma variável antes de lhe atribuir um valor.

Muitos dos problemas esquisitos que podem ocorrer em programas estão relacionados a variáveis que não foram inicializadas.

Devemos ter em mente que uma variável é apenas um *lugar* para guardar dados, assim como uma caixa ou uma gaveta, desprovido de qualquer técnica de magia ou adivinhação. O valor de uma variável só muda quando você o fizer explicitamente. Por exemplo:

```
int a, b, c;
a = 5;
b = a;
c = 5*a;
a = 7;
```

Se eu escrever isso, ao final desse trecho, o valor de a será 7, mas o valor de b continuará sendo 5 e o de c continuará sendo 25. As variáveis são sempre independentes, não há como criar vínculos do tipo "c sempre vale 5a".

Também vale reforçar que uma variável é um lugar, de certa maneira, *temporário*. Quando lhe atribuímos um valor, qualquer valor que nela houvesse anteriormente é completamente esquecido.

1.3 Entrada e saída de dados

Um programa de computador é praticamente inútil se não tiver nenhum tipo de interação com o usuário. Por exemplo, quando aprendermos a fazer cálculos, precisaremos aprender também alguma maneira de mostrar os resultados ao usuário para que isso tenha alguma utilidade. A forma mais simples de fazer isso é de mostrá-los na tela — isso é chamado de *saída* de dados.

Já fizemos um exemplo bem simples de saída de dados, que foi o programa inicial que imprime uma string pré-fixada; mas, em geral, um programa deve ser capaz de imprimir valores que mudam conforme a execução do programa e/ou que dependem, por exemplo, dos dados que o usuário forneceu. Com o que vimos, você talvez fique tentado a escrever isso para imprimir o valor de uma variável inteira:

```
int num;
num = 17;
printf(num);
```

Infelizmente, isso está **errado**! No entanto, ainda é possível usar a função printf para esse propósito; só precisamos mudar um pouco a maneira de usá-la. Observe o seguinte exemplo:

```c
#include <stdio.h>

int main()
{
   int dia, mes, ano;

   dia = 22;
   mes = 7;
   ano = 2008;

   printf("Hoje é dia %d/%d/%d.\n", dia, mes, ano);
   return 0;
}
```

Este exemplo mostra como podemos imprimir uma string que contenha o valor de uma variável: colocamos o código `%d` onde queremos que a variável apareça; a variável em si é especificada à direita da string. Neste exemplo, já colocamos três variáveis de uma vez (com o código repetido em três lugares diferentes), e o computador as substitui na ordem em que elas aparecem após a mensagem que você escreveu. Há uma ilustração disso na figura 1.1.

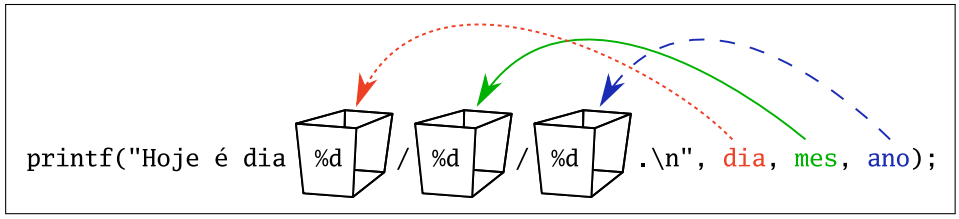


Figura 1.1: Ilustração do funcionamento dos "guardadores de lugar" da função printf.

O código `%d` é um "guardador de lugar" para as variáveis que se seguem. Mais especificamente, o computador lê a string, trechinho por trechinho; quando encontra um código desses, ele pega a primeira variável que encontrar depois da string; a cada novo código encontrado, ela pega a próxima variável da lista, até que todas se esgotem.

Na verdade, o código `%d` só serve se a variável que você quer imprimir é um inteiro; se for um número de ponto flutuante, por exemplo, usamos outro código. Na verdade, existem diversos desses códigos, que podem ser utilizados para fazer algumas alterações no formato da saída do programa.

É bom acostumar-se com a nomenclatura: dentro dos parênteses que seguem o nome da função printf, cada um dos 4 itens separados por vírgulas é um **parâmetro** ou **argumento**. Em outras palavras, quando imprimimos uma mensagem na tela com a função printf, o "esqueleto" da mensagem é o primeiro parâmetro, e os demais parâmetros devem corresponder às expressões que serão inseridas dentro do esqueleto (nos "guardadores de lugar").

Não menos importante que a saída é a *entrada* de dados, ou seja, o ato de ler informações fornecidas pelo usuário. Novamente, uma das maneiras mais simples de fazer isso é ler dados que o usuário digita pelo teclado. Para isso, usamos uma função parente da printf, a função **scanf**. O que ela faz é passar o controle para o teclado, esperar que o usuário digite algo

e termine com a tecla Enter, interpretar o que o usuário escreveu e salvar em uma ou mais variáveis.

Para cada dado que você quiser ler, você deve fornecer à função scanf duas informações: o tipo de dado (inteiro, número de ponto flutuante, etc.) e o lugar onde o dado deve ser armazenado (uma variável). Um exemplo simples:

```
#include <stdio.h>

int main()
{
   int idade;

   printf("Qual é a sua idade? ");
   scanf("%d", &idade);
   return 0;
}
```

Veja que aqui apareceu novamente o código `%d`, que significa o mesmo que na função printf, mas funciona ao contrário: ele lê um inteiro do teclado e guarda seu valor na próxima variável da lista de parâmetros. Você também poderia ler vários valores e guardar em várias variáveis num comando só:

```
#include <stdio.h>

int main()
{
   int dia, mes, ano;

   printf("Qual é a sua data de nascimento (no formato dd mm aaaa)? ")↩
       ;
   scanf("%d %d %d", &dia, &mes, &ano);
   return 0;
}
```

Mas aqui ainda tem uma coisa esquisita: tem um E comercial (`&`) antes do nome de cada variável. Pra que isso serve? Ele é uma maneira de dizer que estamos nos referindo ao *lugar* em que está a variável (um lugar na memória), e não ao seu valor. Isso é necessário justamente para que a função possa colocar o valor lido no lugar que queremos — veja que não estamos atribuindo explicitamente nenhum valor às nossas variáveis. (Veremos como isso funciona no Capítulo **??**, *Ponteiros*.) Você não pode deixar esse E comercial de lado; se você o omitir, o valor lido não irá para a variável que você pediu, e seu programa poderá terminar com um erro do tipo *Falha de segmentação*.

1.4 Matemática básica

A palavra “computador” vem do latim *computus*, que significa ‘cálculo’, ‘conta’. Era justamente esse o objetivo do computador na sua primeira concepção: fazer contas. Então, nada mais razoável que dedicar um tempo para aprender a fazer contas em C.

As operações matemáticas básicas não são nenhum mistério em C. A linguagem possui os mesmos operadores matemáticos com os quais estamos acostumados (com algumas adaptações que foram outrora necessárias), de acordo com a seguinte tabela:

Operador	Significado
`+`	adição
`-`	subtração
`*`	multiplicação
`/`	divisão
`%`	resto da divisão inteira

Os operadores em C funcionam da maneira usual — para realizar uma operação com dois números, basta colocar o operador entre eles. Esses números podem ser tanto constantes numéricas quanto variáveis (ou qualquer coisa que em C for considerada uma expressão). Formalmente, os números ou as expressões às quais se aplica um operador são chamados de *operandos*.

Podemos também montar expressões aritméticas mais complexas, e nelas as operações seguem também a ordem usual: multiplicações e divisões[3] são realizadas antes de adições e subtrações, a menos que você use parênteses para alterar a ordem. Observe que para esse fim você *não* pode usar os colchetes e chaves; em compensação, você pode usar vários níveis de parênteses, como em `3 * (5 * (7 - 1))`.

O operador - (o sinal de menos) também pode ser usado para obter o valor de uma variável com o sinal trocado. Ele também é usado como na matemática: `-num` corresponde ao valor de `num` com o sinal trocado. Note que esse valor não muda o valor da variável, apenas obtém uma "cópia" dele com o sinal trocado. Para alterar o valor da variável, é necessário dizer explicitamente que você quer guardar o valor com sinal trocado de volta na variável.

Veja alguns exemplos de uso dos operadores aritméticos, com os resultados anotados ao lado:

```
x = 14 / 7;                 /* 2 */
y = 12 % 7;                 /* 5 (resto da divisão) */
x = 12 + 5 * 7;             /* 47 */
x = (12 + 5) * 7;           /* 119 */
x = ((12 + 5) * 7 + 2) * 3; /* 363 */
fat = 1 * 2 * 3 * 4 * 5;    /* 120 */

x = y + 2*z;                /* usando os valores atualmente
                               guardados em outras variáveis ⟵
                                   */

x = -x;                     /* troca o sinal de x e
                               guarda de volta em x */

x = x + 1;                  /* aumenta em uma unidade o valor
                               de x, guardando de volta em x ⟵
                                   */
```

[3] Aqui está incluído também o uso do operador `%`, já que o resto também é resultado de uma operação de divisão.

Note (pelos dois últimos exemplos) que podemos usar a mesma variável nos dois lados da operação de atribuição. Isso não tem nenhuma ambigüidade para o computador; ele sabe reconhecer que, no lado direito, você está fazendo algum cálculo que envolve o valor da variável, e que o lado esquerdo representa o lugar onde você quer guardar o resultado desse cálculo. O computador primeiro faz o cálculo com o valor atual da variável, e depois guarda de volta na variável.

Você pode usar expressões aritméticas diretamente na função printf: os parâmetros não precisam ser nomes de variáveis. Por exemplo:

```
int x;
printf("Digite um número inteiro: ");
scanf("%d", &x);
printf("O dobro de %d é %d.\n"
       "O quadrado de %d é %d.\n", x, 2*x, x, x*x);
```

Na função scanf, é lógico que o lugar de armazenamento tem de ser, obrigatoriamente, uma variável; não faz sentido nenhum escrever algo como `&(2*x)`, pois a expressão `2*x` não representa um lugar bem definido onde um valor pode ser guardado.

1.4.1 Contas com inteiros

O exemplo a seguir converte uma temperatura (inteira) dada na escala Celsius para a escala Fahrenheit. Para quem não se lembra, a relação entre dois valores T_C (em Celsius) e T_F (em Fahrenheit) correspondentes à mesma temperatura é

$$T_F - 32 = \frac{9}{5} T_C.$$

```
#include <stdio.h>

int main()
{
   int celsius, fahrenheit;

   printf("Digite a temperatura de hoje: ");
   scanf("%d", &celsius);
   fahrenheit = 9 * celsius / 5 + 32;

   printf("Hoje está fazendo %d graus Fahrenheit!\n", fahrenheit);

   return 0;
}
```

Você, que conhece a propriedade comutativa da multiplicação, deve imaginar que a conta principal do programa poderia ser escrita de várias maneiras:

```
fahrenheit = 9 * celsius / 5 + 32;
fahrenheit = celsius * 9 / 5 + 32;
fahrenheit = celsius / 5 * 9 + 32;
fahrenheit = 9 / 5 * celsius + 32;
```

Em teoria, isso estaria certo. Agora tente executar o programa com as quatro variações, testando algumas temperaturas diferentes. Você deve reparar que as duas primeiras variações dão os valores mais próximos do valor correto, enquanto que a terceira dá um erro maior e a quarta dá um erro realmente grosseiro. Por que isso ocorre? Para o computador, faz sentido que uma operação entre inteiros deve dar um resultado inteiro; por isso, quando você divide dois inteiros, você obtém apenas a parte inteira do resultado. Dessa maneira, quando fazemos no programa a divisão por 5, obtemos a versão arredondada do valor que esperaríamos obter, por exemplo, numa calculadora.

Os resultados foram diferentemente incorretos porque os arredondamentos são realizados em diferentes etapas. Sabendo que o C calcula expressões matemáticas sempre da esquerda para a direita, podemos prever, por exemplo, que, na última variação da nossa conta, o primeiro cálculo realizado é `9 / 5`, que dá 1 (com resto 4). Assim, a temperatura em graus Celsius é simplesmente somada com 32, o que dá um erro muito grosseiro. Na terceira conta, o problema é na divisão da temperatura por 5 — qualquer temperatura que não seja múltipla de 5 sofrerá um arredondamento para baixo, de modo que 29 °C seriam convertidos para 77 °F, que na verdade correspondem a 25 °C.

Por isso, neste caso, a melhor opção a usar é a primeira ou a segunda, já que o arredondamento na divisão é feito sobre um número maior e ocorre numa das últimas etapas, de modo que o erro não se propaga mais para outras contas — ou melhor, ele só se propaga para a adição, na qual não há perigo de haver outros erros, pois a adição de inteiros é sempre exata.

Mais adiante veremos como trabalhar com números de ponto flutuante, o que resolverá o problema das divisões — mas também traz alguns outros problemas, como também vamos estudar.

1.5 Boas maneiras em C

1.5.1 Comentários

Talvez você tenha reparado que num certo ponto usei os símbolos `/*` e `*/` para anotar resultados de contas. Eles também são parte da linguagem; eles são usados para fazer **comentários** no meio do código. Você pode escrever o que quiser entre esses dois símbolos, pois tudo que ali estiver será ignorado pelo compilador. A única coisa que você não pode fazer é colocar um comentário dentro do outro:

```
/* um comentário /* isto é um erro */ dentro do outro */
```

Existe também uma outra maneira de fazer comentários, que foi copiada da linguagem C++: usam-se duas barras `//`, mas o comentário só vale até o final da linha. Essa segunda maneira só foi fixada no padrão mais recente da linguagem (C99); se seu compilador estiver operando no modo ANSI, ele não gostará desse segundo tipo de comentário.

```
/* Comentário estilo C
 * ===================
 *   Começa com barra e asterisco,
 *   termina com asterisco e barra.
 */
```

```
// Comentário estilo C++
// ====================
//   Começa com duas barras,
//   vale apenas até o final da linha.
```

Tenho algumas palavrinhas a proferir sobre o uso dos comentários, mas postergarei esse assunto para o próximo capítulo, quando tivermos visto um pouco mais de teoria.

1.5.2 Estrutura e estilo

Em C, praticamente todos os espaços, tabulações e quebras de linha supérfluos — de agora em diante, sempre que eu falar de espaços,[4] entenda que tabulações e quebras de linha também estão incluídas — são ignorados pelo compilador (exceto, é claro, em uma string). Os programas abaixo são, perante as regras da linguagem, absolutamente equivalentes:

```
#include <stdio.h>
int main(){printf("Olá, maravilhoso mundo da programação!\n");return ↩
    0;}
```

```
#include <stdio.h>

int main()
{
   printf("Olá, maravilhoso mundo da programação!\n");
   return 0;
}
```

```
#include <stdio.h>

int          main(

){
   printf(

        "Olá, maravilhoso mundo da programação!\n"


       );
   return 0;
          }
```

Mas qual deles é mais fácil de ler? O segundo, não é? Embora o estilo específico que você irá usar possa variar de acordo com sua preferência, há alguns princípios gerais que você deve procurar seguir ao escrever um programa:

[4] Em inglês, há a expressão *whitespace*, que indica qualquer sequência de espaços, tabulações e quebras de linha. Não conheço nenhuma expressão equivalente em português, então teremos de conviver com essa ambiguidade.

- Escreva uma instrução (um comando) por linha. Quando for necessário, divida a instrução em mais de uma linha para melhorar a legibilidade (você pode quebrar a linha em qualquer lugar onde poderia colocar um espaço). Em algumas situações também pode ser aceitável juntar duas instruções em uma linha só, mas não abuse disso.

- Sempre que você tiver um *bloco* de comandos (algo delimitado por chaves), o conteúdo do bloco deve estar mais afastado da margem que o exterior do bloco. É isso que chamamos de **indentação**; ela torna muito mais fácil a visualização da estrutura do código e de onde os blocos se encaixam. Você também deve deixar com o mesmo recuo as linhas que pertencem ao mesmo bloco, senão acaba aparecendo uma falsa noção de hierarquia (ou uma verdadeira noção de bagunça).

Controle de fluxo 2

Com o pouco que aprendemos até agora, só é possível construir programas *lineares*, ou seja, que só sabem fazer uma determinada sequência de operações, quaisquer que sejam os dados fornecidos (ou outras condições). Na maioria dos programas, isso não é suficiente; é preciso que o programa saiba fazer *decisões*. A capacidade do computador de fazer decisões é conhecida como **controle de fluxo**.

2.1 Desvios condicionais: if

Uma das estruturas mais simples de controle de fluxo é a construção **if** (*se*), que verifica se uma certa condição é satisfeita, e executa um conjunto de comandos em caso afirmativo. Em C, isso se escreve assim:

```
if (condição)
{
   /* comandos aqui */
}
```

O funcionamento dessa estrutura é simples. Quando a execução do programa atinge o if, a condição é avaliada, e há duas saídas possíveis:

- se a condição for satisfeita, o programa executa os comandos colocados dentro do par de chaves, e depois continua a executar os comandos fora do par de chaves;
- se a condição não for satisfeita, o programa simplesmente pula o que está entre chaves e continua a executar os comandos após as chaves.

Para que isso tenha alguma utilidade, precisamos aprender como são escritas as tais condições. Dentre as condições mais simples que podemos escrever estão as comparações entre números. Elas são escritas com os operadores a seguir, denominados *operadores relacionais*:

Tabela 2.1: Operadores relacionais, para comparação entre números.

Operador	Significado
`<`	menor que
`<=`	menor que ou igual a
`>`	maior que
`>=`	maior que ou igual a
`==`	igual a (cuidado! são DOIS iguais!)
`!=`	diferente de

Do mesmo jeito que ocorre para os operadores matemáticos, não há nenhum grande mistério no uso dos operadores relacionais. Por exemplo, a condição "a é menor do que b" é escrita como `a < b`; a igualdade entre a e b pode ser verificada pela expressão `a == b`, com DOIS iguais. Porém, uma condição *composta* como "x está entre a e b" (matematicamente, $a < x < b$) *não* pode ser escrita como `a < x < b`; a expressão até é sintaticamente válida, mas seu resultado é completamente diferente do que você esperaria. O jeito certo de fazer essa comparação será esclarecido daqui a pouco.

```
if (a == b)
{
   printf("Os números são iguais!\n");
}
```

Existe uma abreviação muito útil que pode ser usada com a estrutura if (e também pode ser usada com outros tipos de estruturas que veremos a seguir). Quando houver apenas um comando dentro do bloco delimitado pelas chaves, você pode omitir as chaves. Assim, poderíamos escrever o trecho acima de forma um pouco mais compacta:

```
if (a == b)
   printf("Os números são iguais!\n");
```

Reforço que você só pode fazer essa abreviação quando dentro do bloco houver *apenas um comando*. Se você fizer isso quando houver mais de um comando, mesmo que estejam todos na mesma linha ou com a mesma indentação, todos os comandos a partir do segundo serão dados como *externos* ao bloco. Por exemplo, observe o código a seguir:

```
if (a < 0)
   printf("O valor de A não pode ser negativo!\n");
   a = 0;  /* indentação enganosa! */
```

A terceira linha foi indentada de modo que pareça fazer parte do if; na verdade ela está fora dele, e será executada em qualquer caso, seja a negativo ou não. Esse código é equivalente a este outro a seguir; este sim dá a correta impressão da organização do programa.

```
if (a < b) {
   printf("A é menor que B!\n");
}
a = b;
```

Outro erro comum é confundir o operador de comparação de igualdade, `==`, com o operador de atribuição, `=`. O código a seguir é "gramaticalmente correto", mas não faz o que ele aparenta fazer:

```
if (a = b)
   printf("A é igual a B!\n");
```

O que está acontecendo aqui? Copiamos o valor de *b* para a variável *a*; o comando de atribuição `a = b` é por si mesmo uma expressão, cujo valor é igual ao valor que foi atribuído. Esse valor é, então, interpretado como se fosse o valor de uma condição — cobriremos isso mais adiante, na seção 2.6; mas veja que, independentemente de quem ele é, esse valor condicional não tem a ver com a relação entre *a* e *b*, pois ele só depende do valor da variável *b*! Portanto, a conclusão de que *a* é igual a *b* a partir desse `if` está errada.

Esse é um dos vários erros que podem não ser detectados pelo compilador, pois o código, ainda que semanticamente incorreto, é sintaticamente válido; por causa disso, você conseguirá compilar e executar seu programa, mas alguma parte dele exibirá algum comportamento inesperado. Portanto, muito cuidado ao fazer comparações de igualdade.

2.1.1 Condições alternativas: else

Suponha que você quer ler um número do teclado e imprimir uma mensagem dizendo se o número é par ou ímpar. Para verificar se é par, podemos simplesmente ver se ele é divísivel por 2, ou seja, ver se o resto da divisão por 2 é zero. Isso poderia ser feito assim:

```
#include <stdio.h>

int main()
{
   int num;
   printf("Digite um número: ");
   scanf("%d", &num);

   if (num % 2 == 0)
      printf("O número é par.\n");

   if (num % 2 != 0)
      printf("O número é ímpar.\n");

   return 0;
}
```

Mas pense bem: sendo o número digitado inteiro, ou ele é par ou é ímpar; então, se já verificamos que ele não é par, não precisamos fazer outra conta para ver que ele é ímpar. Esse tipo de padrão — associar um procedimento ao caso em que a condição é satisfeita, e outro procedimento ao caso em que ela é falsa — é extremamente comum em programação. Por isso, a cada condição você pode associar também um outro bloco de comandos, a ser executado caso ela seja *falsa*. Para isso, você usa a palavra-chave **else** (que significa *senão*), da seguinte maneira:

```
if (condição) {
```

```
        /* SE a condição for satisfeita, executa estes comandos */
    }
    else {
        /* SENÃO, executa estes outros comandos */
    }
```

(Continua existindo a abreviação do um-comando: se houver apenas um comando dentro do else, você pode tirar as chaves.)

Assim, nosso programa acima pode ser reescrito da seguinte maneira, que também deixa bem mais claro o fato de que as duas situações são opostas, complementares:

```
#include <stdio.h>

int main()
{
    int num;
    printf("Digite um número: ");
    scanf("%d", &num);

    if (num % 2 == 0)
        printf("O número é par.\n");
    else
        printf("O número é ímpar.\n");

    return 0;
}
```

2.2 Repetindo operações: laços (*loops*)

O computador é útil para *automatizar* tarefas; por exemplo, tarefas *repetitivas* que seriam muito trabalhosas se realizadas de outra maneira. Por isso, é interessante criar, numa linguagem de programação, uma estrutura que permita a execução repetida de uma tarefa ou de várias tarefas parecidas.

Em C, existem três tipos de estruturas de repetição, mas o princípio de funcionamento de todas é o mesmo: repetir um certo conjunto de comandos até que uma certa condição de parada seja satisfeita. Esse tipo de estrutura costuma ser chamado de *laço* ou, equivalentemente em inglês, *loop*.

O tipo de laço mais simples é o **while** (*enquanto*). O funcionamento dele é simples — uma condição é avaliada; se ela for falsa, continua a execução do resto do programa; se ela for verdadeira, um bloco de comandos é executado, e volta-se ao começo do processo (avaliando a condição novamente). Em outras palavras, o bloco de comandos é executado *enquanto a condição for satisfeita*. Isso é representado no esquema da figura 2.1.

A sintaxe desse laço, bem parecida com a do if, é a seguinte:

```
while (condição)
{
    /* comandos aqui */
}
```

Cada execução do bloco de comandos é chamada de **iteração** — não só para o while, mas também para os outros laços que veremos posteriormente. Assim, uma observação importante a fazer é que, se a condição já for falsa quando o while for atingido, nenhuma iteração será executada — o bloco de comandos será ignorado.

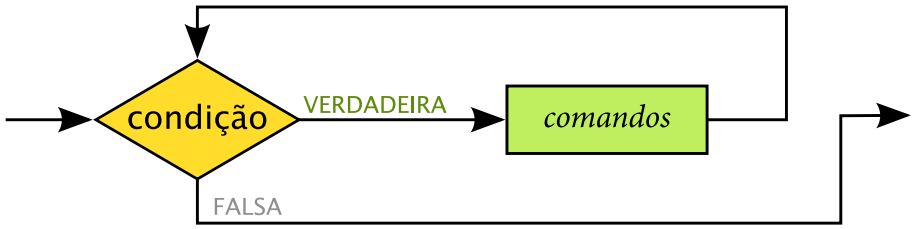


Figura 2.1: Esquema de execução da estrutura **while**.

Vamos criar um exemplo bem básico: um programa que imprime na tela, em ordem crescente, os números inteiros de 1 a 10. Como fazer isso usando while?

Uma idéia é usar uma variável que tem o valor inicial 1, e aumentar esse valor de 1 em 1 *até que* chegue a 10 (aqui entra o while!), imprimindo o valor atual antes de realizar cada incremento. Ficou confuso? Vamos ver um esquema disso, usando no lugar do C um esboço em uma linguagem de programação simplificada e genérica:

```
num ←1
repita  enquanto num ≤ 10:
    imprima num
    num ←num + 1
```

Para ter certeza de que ficou tudo claro, vamos simular a execução desse pseudocódigo. Veja que inicialmente a variável *num* vale 1. Como $1 \leq 10$, entramos no bloco do repita; assim, imprimimos o número 1 e a variável passa a valer 2. Voltamos para a avaliação da condição e obtemos novamente um resultado positivo, $2 \leq 10$; o programa imprime o número 2 e a variável recebe o valor 3. Isso continua até que o valor de *num* seja 9; esse número é impresso e o valor 10 é colocado na variável. Aí ocorre a última iteração do laço: imprimimos o número 10, e *num* passa a valer 11. Com isso, a condição do laço deixa de ser verdadeira, pois ela será avaliada como $11 \leq 10$.

Pseudocódigo é o nome dado a esse tipo de "linguagem" de programação simplificada e genérica, e é geralmente usado para enfatizar o *algoritmo* de uma certa operação, sem depender das peculiaridades das diversas linguagens.

Algoritmo, por sua vez, é a seqüência de instruções usada para completar uma certa tarefa computacional. Essa definição é muito parecida com a de *programa*; de fato, um programa pode ser considerado uma grande e complexa associação de algoritmos para as diversas tarefas que ele tem de realizar. Em geral, o algoritmo para realizar uma tarefa não é único; pode haver vários, alguns mais eficientes ou práticos que outros.

Agora vamos passar isso de volta para o C. A tradução é quase literal, excluindo-se o "cabeçalho" e o "rodapé" do programa:

```c
#include <stdio.h>

int main()
{
   int num;
   num = 1;
   while (num <= 10) {
      printf("%d\n", num);
      num = num + 1;
   }

   return 0;
}
```

Esse tipo de código é muito importante em programação, e ilustra um padrão conhecido como **contador** — uma variável usada em um laço, cujo valor varia em uma unidade a cada iteração. Esse padrão é usado em diversas situações que envolvam a repetição de várias operações iguais ou bem parecidas, que possam ser diferenciadas apenas pelo valor do contador.

EXERCÍCIO 2.1 *Faça um programa que lê um número n do teclado e, em seguida, imprime todos os números inteiros de 1 a n junto com seus quadrados.*

Vamos construir um exemplo mais sofisticado: calcular o fatorial de um número inteiro não-negativo fornecido pelo usuário.

Definição. Dado um número natural n, definimos como **fatorial** de n, e denotamos $n!$, o produto de todos os números naturais menores que ou iguais a n. Observe que $1! = 1$. Definimos também $0! = 1$.

Por exemplo, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$.

Como podemos usar o while nessa tarefa? Suponhamos primeiro que n é maior que 1. Nessa situação, devemos fazer várias multiplicações, cada vez com fatores diferentes — afinal, a multiplicação com n fatores se define indutivamente a partir da multiplicação elementar de dois fatores. Se realizarmos as operações da esquerda pela direita, faremos as seguintes contas:

$$\begin{aligned} &1 \cdot 2 \\ &(1 \cdot 2) \cdot 3 \\ &\vdots \\ &(1 \cdot 2 \cdots (n-1)) \cdot n \end{aligned}$$

Tendo isso em vista, podemos montar um algoritmo no qual, em cada passo, o resultado do passo anterior (a) é multiplicado pelo valor atual de um contador (b), que vai de 2 até n:

$a \leftarrow 1$
$b \leftarrow 2$

```
repita enquanto b ≤ n:
    a ← a * b
    b ← b + 1
```

Ao final do algoritmo, a variável *a* conterá o fatorial do número n, conforme desejávamos. Agora pensemos nos casos $n = 1$ e $n = 0$. Se $n = 1$, a condição do laço ($2 \leq 1$) será, já de início, falsa, e o valor inicial de *a* garantirá o resultado correto, $1! = 1$. Caso $n = 0$, a condição também será falsa de início, e assim também teremos o resultado $0! = 1$. Agora vamos escrever tudo isso em C:

```c
/* Calcula o fatorial de um número inteiro não-negativo. */
#include <stdio.h>

int main()
{
   int n, a, b;

   printf("Digite um número: ");
   scanf("%d", &n);

   a = 1;
   b = 2;
   while (b <= n) {
      a = a * b;
      b = b + 1;
   }

   printf("%d! = %d\n", n, a);
   return 0;
}
```

Exercício 2.2 *Elabore um algoritmo (e construa um programa) que calcule o fatorial de um número n multiplicando os fatores da direita para esquerda (ou seja, começando de n e indo até 1). Quais problemas você encontra?*

Exercício 2.3 *Faça um programa que lê do teclado dois números, $a > 0$ e $b \geq 0$, e imprime o valor da potência a^b.*

Vamos agora ver outro tipo de programa que vale a pena ser ressaltado. Buscaremos resolver o seguinte problema: ler do teclado uma sequência de números, cujo tamanho não é inicialmente conhecido, mas que sabemos ser terminada por um zero (ou seja, os demais números da sequência são todos não-nulos), e somar os números lidos. Que tipo de algoritmo podemos usar para atacar esse problema?

Obviamente, se não sabemos *a priori* quantos elementos a sequência tem, não podemos armazenar todos os elementos para depois somar todos; precisamos acumular as somas parciais a cada número lido. Utilizaremos para isso uma variável chamada *soma*, que deverá ser inicializada com o valor zero, e à qual se somará cada número lido.

A idéia central é ler números do teclado indefinidamente até que o número lido seja zero. Sempre que o número for diferente de zero, devemos incluí-lo na soma e continuar

lendo números. Declarando uma variável *num* para armazenar os números lidos, podemos proceder assim:

```
while (num != 0) {
   soma = soma + num;

   printf("Digite outro número ou 0 para terminar: ");
   scanf("%d", &num);
}
```

Mas é preciso prestar atenção a um detalhe: `num` deverá ser lido pela primeira vez antes desse laço — caso contrário, a variável não estaria inicializada!. Feitas essas observações, nosso programa ficará com essa cara:

```
#include <stdio.h>

int main()
{
   int num, soma;

   soma = 0;

   printf("Digite um número ou 0 para terminar: ");
   scanf("%d", &num);

   while (num != 0) {
      soma = soma + num;

      printf("Digite um número ou 0 para terminar: ");
      scanf("%d", &num);
   }

   printf("Soma = %d\n", soma);
   return 0;
}
```

Vamos agora adicionar mais uma tarefa ao nosso problema: encontrar o elemento máximo da sequência digitada pelo usuário. Para ver como faremos isso, imagine que você dispõe de papel e caneta, e que alguém está ditando para você a tal sequência de números. Como você realizaria essa tarefa? Você anota o primeiro número e, caso ouça posteriormente um número maior que ele, anota-o logo abaixo; se ouvir outro número ainda maior, anota-o abaixo do anterior; e assim por diante — não é necessário tomar nota de todos os números. No final, o último número anotado será o maior número da sequência.

Um computador poderia fazer isso exatamente da mesma maneira. Em vez de 'anotar' os números, ele os guardaria em variáveis; além disso, ele não precisaria manter as 'anotações' anteriores; basta substituir o valor da variável. Assim, nosso programa completo poderia ser escrito assim:

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

```c
int main()
{
   int soma, /* somas parciais */
       num,  /* último número lido */
       max;  /* candidatos a máximo */

   soma = 0;

   printf("Digite um número ou 0 para terminar: ");
   scanf("%d", &num);
   max = num;

   while (num != 0) {
      soma = soma + num;
      if (num > max)      /* para procurar o máximo */
         max = num;

      printf("Digite um número ou 0 para terminar: ");
      scanf("%d", &num);  /* lê o próximo número */
   }

   printf("Soma = %d\n", soma);
   printf("Máximo = %d\n", max);
   return 0;
}
```

2.3 Contadores e laços for

Se você olhar com atenção para os exemplos do fatorial e do programa que conta de 1 a 10, que utilizam uma variável-contador, verá que os dois programas têm estruturas bem parecidas: primeiro era definido o valor inicial do contador; depois, escrevia-se um laço que definia o valor máximo desse contador; no final desse laço, o valor do contador era aumentado. Esquematicamente, tínhamos

```
inicialização do contador;
while (controle do valor máximo)
{
   comandos;
   incremento do contador;
}
```

Esse tipo de situação é uma das mais frequentes aplicações dos laços; por isso, essa estrutura comum foi abreviada em outro tipo de laço: a estrutura **for**. Com ela, a escrita desses laços fica mais compacta e de mais fácil visualização:

```
for (inicialização; controle; incremento)
{
   comandos;
```

```
}
```

Com isso, podemos reescrever os exemplos anteriores da seguinte maneira:

```c
/* Imprime todos os números inteiros de 1 até 10
 * junto com seus quadrados */
#include <stdio.h>

int main()
{
   int num;
   for (num = 1; num <= 10; num = num + 1)
      printf("%d\n", num);
   return 0;
}
```

```c
/* Calcula o fatorial de um número inteiro não-negativo. */
#include <stdio.h>

int main()
{
   int n, a, b;
   printf("Digite um número: ");
   scanf("%d", &n);

   a = 1;
   for (b = 2; b <= n; b = b + 1)
      a = a * b;

   printf("%d! = %d\n", n, a);
   return 0;
}
```

O funcionamento da estrutura for pode ser mais detalhado como a seguir (você pode ver isso de maneira mais esquemática na figura 2.2):

1. A variável do contador é inicializada com um valor especificado; isso ocorre apenas uma vez, antes do início do laço. A inicialização costuma ser uma instrução do tipo `i = 1`.

2. A condição de controle especifica que condição a variável do contador deve satisfazer para que os comandos sejam executados; em outras palavras, quando essa condição deixar de valer, o laço irá parar de se repetir. Por exemplo, uma condição do tipo `i <= n` diz que o laço será interrompido assim que o contador `i` ficar maior que `n`. Se você quiser que seu contador *decresça* até um valor mínimo, você usará uma condição do tipo `i >= n`.

3. Sempre que a condição de controle for satisfeita, é executada uma iteração do bloco de comandos fornecido, de cabo a rabo. Mesmo que a condição de controle deixe de ser satisfeita no meio do bloco, o laço só será abandonado quando chegarmos ao final do bloco. Essa última observação vale tanto para o for quanto para o while.

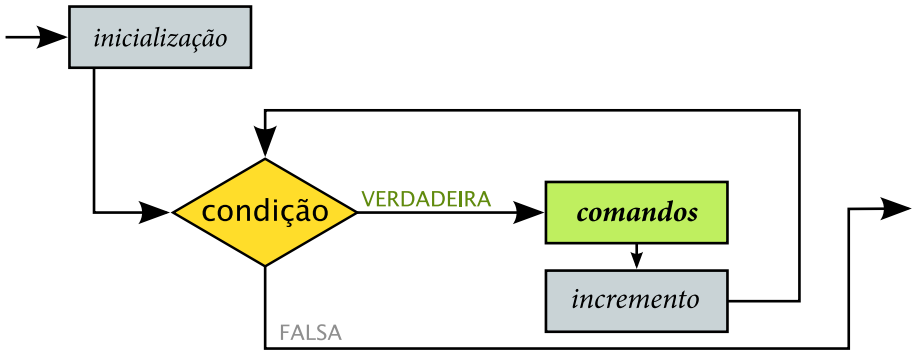


Figura 2.2: Esquema de execução da estrutura **for**.

4. Ao final de cada iteração, executa-se a instrução de incremento — ela pode ser qualquer instrução que altere o valor do contador, como `i = i + 1` ou `i = i - 2`. Apesar do nome 'incremento', você pode variar o valor do contador da maneira que desejar — diminuindo, por exemplo.

Apesar de termos vinculado nossa discussão do for a uma variável-contador, o for é uma estrutura bastante flexível. No entanto, isso não significa que você deve sair usando o for a torto e a direito e esquecer o while; deve existir uma boa razão para existirem os dois tipos de estrutura. E essa razão é semântica: você deve usar o for quando seu laço tiver um contador ou algo com funcionamento semelhante; nos outros casos, você deve, em geral, ater-se ao while.

2.3.1 Abreviações

Atribuições que envolvem o valor original da variável que está sendo alterada (como a nossa já conhecida `i = i + 1`) são muito comuns, já que são usadas o tempo todo nos laços com contadores; por isso, foram criadas algumas abreviações bastante úteis. Quando $\odot$ representa um dos cinco operadores aritméticos, podemos abreviar `a = a` $\odot$ `b` para `a` $\odot$`= b`. Assim, temos:

Abreviação	Significado
`var += expr`	`var = var + expr`
`var -= expr`	`var = var - expr`
`var *= expr`	`var = var * expr`
`var /= expr`	`var = var / expr`
`var %= expr`	`var = var % expr`

Existe ainda outro tipo de abreviação — quiçá o mais usado em C — que serve para aumentar ou diminuir em uma unidade o valor de uma variável. São os operadores `++` e `--`, que podem ser usados tanto *antes* quanto *depois* do nome das variáveis a serem alteradas. Ou seja, para aumentar o valor de uma variável, podemos escrever

```
var++;
```

```
++var;          /* equivalem a: var = var + 1 */
```

e, para diminuir, usamos

```
var--;
--var;          /* equivalem a: var = var - 1 */
```

No contexto atual, o mais comum é usar os operadores *depois* dos nomes das variáveis. Porém, mais tarde veremos que existe uma diferença importante entre as duas possibilidades de posição.

Dessa maneira, a "cara" de um laço com contador passa a ficar parecida com isso (aqui foi reescrito o laço do último exemplo):

```
for (b = 2; b <= n; b++)
   a *= b;
```

Exercícios

2.4. Crie um programa que lê um número natural n do teclado e imprime todos os divisores desse número. Ao final, imprima também a soma dos divisores encontrados.

2.5. Aproveite o programa do exercício anterior para verificar se um número n dado é primo.

2.6. Crie um programa que calcula o *fatorial duplo* de um número natural n (lido do teclado), que é definido como a seguir:

$$n!! = \begin{cases} n(n-2)\cdots 4 \cdot 2, & \text{se } n \text{ é par;} \\ n(n-2)\cdots 3 \cdot 1, & \text{se } n \text{ é ímpar.} \end{cases}$$

2.7. Faça um programa que lê dois números inteiros a e b, sendo $a > 0$ e $b \geq 0$, e calcula a potência a^b. Veja se seu algoritmo funciona para o caso $b = 0$; tente elaborá-lo de modo que não seja preciso um teste especial para esse caso.

2.8. Faça um programa que lê uma sequência de números naturais do teclado (terminada por zero) e imprime a quantidade de números pares e ímpares da sequência. Imprima também o maior e o menor ímpar e o maior e o menor par encontrados na sequência.

2.9. *Variação sobre o mesmo tema:* Mesmo problema que o anterior; mas agora o tamanho n da sequência é conhecido previamente. Antes de tudo você deve ler n, e depois ler os n números.

2.10. Faça um programa que lê do teclado uma sequência de números inteiros não-nulos (terminada por zero) e os soma e subtrai alternadamente — por exemplo, para a sequência $[5, 7, 1, -4, 9]$, seu programa deverá calcular $5 - 7 + 1 - (-4) + 9$.

2.11. [Problema] Crie um programa que recebe um número n e a seguir lê uma sequência de n algarismos de 0 a 9; você deverá calcular e imprimir o número que corresponde àqueles algarismos, na mesma ordem (não vale imprimir dígito por dígito! O objetivo é construir o número a partir da sua sequência de dígitos). Por exemplo, para a sequência $[5, 4, 2, 0, 7, 0]$, seu programa deverá imprimir o número 542070.

2.12. [Idem] Crie um programa que lê um número natural `n` e imprime a soma de seus dígitos (na base decimal).

2.4 Condições compostas

Em muitos casos, você precisará verificar condições mais complexas do que as simples expressões com operadores relacionais, como `x > 10` ou `cod == 15`. Haverá muitas vezes em que você precisará fazer alguma coisa somente quando duas (ou mais) condições forem satisfeitas ao mesmo tempo; em outras, você precisará fazer algo quando qualquer uma, dentre várias condições, for satisfeita.

Por exemplo, suponha que você queira que o usuário digite números entre 0 e 10 e que você precise verificar se os números realmente estão entre 0 e 10. Já falamos anteriormente que, em C, isso *não* se escreve como `0 <= x <= 10` — esse código é válido, mas não faz o que gostaríamos. Mas podemos contornar isso usando ifs encadeados, pois $(0 \leq x \leq 10)$ é equivalente a $((x \geq 0)$ e $(x \leq 10))$:

```
if (x >= 0) {
   if (x <= 10)
      printf("Ok!\n");
   else
      printf("O número precisa estar entre 0 e 10!\n");
}
else
   printf("O número precisa estar entre 0 e 10!\n");
```

No entanto, esse método tem duas desvantagens: (1) ele não deixa tão evidente o fato de que queremos que as duas condições sejam satisfeitas ao mesmo tempo; e (2) tivemos mais trabalho para verificar quando as condições *não* são satisfeitas — foi necessário usar *dois* blocos else, e acabamos usando o mesmo código nos dois.

Quando precisamos de critérios compostos como esse dentro de um laço, a situação torna-se ainda mais complicada; com o que aprendemos até agora, isso é simplesmente impossível de se implementar — mesmo com outros recursos da linguagem, a solução não seria nem um pouco prática ou clara. Por isso, precisamos introduzir dois operadores que permitem a criação de *condições compostas*:

`&&`	*e*	`||`	*ou*

O operador `&&`, chamado de *'E' lógico*, serve para verificar se duas condições são satisfeitas *simultaneamente*. O operador `||`, o *'OU' lógico*, verifica se, dentre duas condições, *pelo menos* uma é satisfeita. (Os nomes desses operadores serão grafados em maiúsculas para evitar confusões no texto.)

Note que, coloquialmente, quando usamos a palavra "ou" para unir duas orações, geralmente imaginamos que apenas uma delas é verdadeira (por exemplo, *"Independência ou morte"* — cada uma das possibilidades exclui o acontecimento da outra). Nesse caso, dizemos que trata-se de um **'ou' exclusivo**. Em outras palavras, esse 'ou' significa que, dentre as duas sentenças, *uma e apenas uma é verdadeira*.

Já em computação, quando dizemos *"A ou B"*, geralmente queremos dizer que, das duas sentenças (condições), *pelo menos uma é verdadeira*. Esse é conhecido como **'ou' inclusivo**. Daqui para a frente, a menos que indiquemos o contrário, usaremos sempre o **'ou' inclusivo**.

Com esses operadores, podemos reescrever aquele exemplo assim:

```
if (x >= 0 && x <= 10)
   printf("Ok!\n");
else
   printf("O número precisa estar entre 0 e 10!\n");
```

Bem mais claro, não? Veja que também podemos inverter as coisas para escrever nosso exemplo usando um operador OU — se um número não satisfaz a condição de estar *entre* 0 e 10, então necessariamente ele é ou menor que zero ou maior que 10:

```
if (x < 0 || x > 10)
   printf("O número precisa estar entre 0 e 10!\n");
else
   printf("Ok!\n");
```

2.5 Repetições encaixadas

Uma grande classe de problemas computacionais comuns requer, para sua solução, o uso de *repetições encaixadas*. Isso, conceitualmente, não é nenhuma novidade; trata-se simplesmente de colocar um laço dentro do outro: para cada iteração do laço externo, o laço interno será executado tantas vezes quanto for necessário para que ele termine.

Podemos encaixar os laços da maneira que quisermos, ou seja, qualquer mistura de for e while é permitida. O único cuidado que devemos ter é o de não misturar as variáveis de controle dos vários laços — por exemplo, se encaixamos dois laços for, devemos dar nomes diferentes aos contadores (por exemplo, i para o laço externo e j para o interno). No entanto, os contadores não precisam ser completamente independentes: os limites dos contadores dos laços internos podem depender da posição atual no laço principal.

Vamos estudar um exemplo. Veremos um programa que encontra os primeiros N números primos, onde N é um número dado. Para isso, passamos por todos os números naturais (enquanto não tivermos selecionado os N primeiros primos) e verificamos se cada um deles é primo. Não é difícil testar se um número n é primo: devemos verificar se ele tem um divisor não-trivial, isto é, um que não seja igual a 1 ou n. Assim, verificamos entre todos os candidatos (a princípio, $\{2, 3, \ldots, n-1\}$, mas essa lista ainda pode ser bem refinada) se há algum divisor de n. Se não acharmos nenhum, é porque n é primo.

Agora vejamos como escrever a solução para nosso problema. Se você fez o exercício 2.5 da página 30, já tem o problema quase inteiro resolvido. (Se não fez, volte e tente resolver!) Vamos ver uma possível implementação da verificação de primalidade de um número:

```
int divisores, /* conta o número de divisores */
    k, /* candidato a divisor */
    n; /* número a ser testado */

divisores = 0;

/* testa todos os candidatos */
for (k = 2; k < n; k++)
   if (n % k == 0)
      divisores++;

if (divisores == 0) {
   /* não achamos nenhum divisor dentre os possíveis
      candidatos, então é primo */
}
```

Essa é uma das implementações mais ineficientes possíveis, pois vários dos candidatos da lista $\{2, 3, \ldots, n-1\}$ podem ser sumariamente descartados. Por exemplo, nenhum número maior que $\frac{n}{2}$ pode ser divisor de n, pois isso significaria que n também tem um divisor entre 1 e 2. Ainda mais, se só precisamos verificar se n é primo (e não achar todos os seus divisores), basta verificar os possíveis divisores $d \leq \sqrt{n}$ (pois, se d for realmente divisor, $n/d \geq \sqrt{n}$ também será divisor). Mas, no momento, para o propósito atual, vamos nos contentar com a maneira ineficiente.

Agora tudo o que precisamos fazer é usar esse código várias vezes (usando um laço) para encontrar N números primos:

```
int N, /* quantidade de primos a procurar */
    i, /* primos já encontrados */
    n, /* candidato atual */
    divisores; /* conta o número de divisores */

for (n = 2, i = 0; i < N; n++) {
   divisores = 0;

   /* testa todos os candidatos a divisor */
   for (k = 2; k < n; k++)
      if (n % k == 0)
         divisores++;

   if (divisores == 0) {
      /* não achamos nenhum divisor dentre os possíveis
         candidatos, então é primo */
      printf("%d\n", n);
      i++;
   }
}
```

Observe que cada iteração do laço `for` externo verifica se *um* número `n` é primo, e só aumenta o contador `i` em caso afirmativo. Veja também que o limite do contador `k`, no `for`

interno, depende do contador `n` do laço externo.

Mais um detalhe que foi usado no programa mas ainda não havia sido mencionado: na inicialização e no incremento de um laço for, é possível escrever comandos compostos. Podemos separar várias atribuições por **vírgulas** (lembre-se que o ponto-e-vírgula separa a inicialização da condição e esta do incremento), como foi feito no comando de inicialização do laço externo:

```
for (n = 2, i = 0; i < N; n++) {
```

Da mesma maneira, poderíamos criar um incremento composto, como `i++, j++`.

2.6 Variáveis booleanas

As estruturas if, while e for dependem da avaliação de expressões condicionais para executarem seu trabalho. Para que essas expressões realmente façam sentido como condições, elas só podem assumir dois valores: *verdadeiro* ou *falso* (que indicam se o bloco correspondente será ou não executado). Variáveis que só indicam esses dois estados são chamadas **booleanas** ou **lógicas**.

Em C, não há nenhum tipo primitivo especificamente para lidar com variáveis booleanas; por isso, convenciona-se que qualquer valor diferente de zero é interpretado como verdadeiro, e o zero é interpretado como falso. Ou seja, qualquer variável ou expressão, independente de seu tipo, pode ter um *valor booleano* que é ou 'verdadeiro' ou 'falso'.[5]

Os operadores que devolvem um valor lógico usam os valores 0 e 1 (além de serem a escolha mais simples, esses dois valores são sempre representáveis em qualquer tipo numérico) para representar os resultados "falso" e "verdadeiro", respectivamente; assim, quando fazemos alguma comparação do tipo `a < 10` ou `x == y`, o valor da expressão comparativa é 0 ou 1. No entanto, em geral não devemos nos preocupar com essa representação, pois essas expressões já têm um valor lógico bem definido.

A partir desse conhecimento, podemos analisar o que acontece em algumas das situações capciosas que mencionamos anteriormente. Primeiro temos as comparações compostas do tipo `a < b < c`; quando escrevemos em C exatamente dessa maneira, o que ocorre é que a expressão é avaliada da esquerda para a direita, dois operandos por vez. Assim, primeiramente é avaliada a expressão `a < b`, cujo resultado poderá ser 0 ou 1 — chamemo-no genericamente de *R*. Então é avaliada a expressão `R < c`, que nada tem a ver com a relação entre *b* e *c*.

Olhemos agora para as comparações errôneas que usam o operador de atribuição em vez do operador de comparação de igualdade, como `if (a = b)`. O valor de uma expressão como `a = b` corresponde ao valor que foi atribuído, ou seja, `b` (ou `a` após a atribuição). Assim, o código `if (a = b)` é equivalente a

```
a = b;
if (b)
```

No entanto, como a variável *b* está num contexto lógico, seu valor será interpretado como um valor booleano, ou seja, *falso* se `b == 0` e *verdadeiro* se `b != 0`. Assim, finalmente, a comparação `if (a = b)` na verdade é equivalente a

[5] Nota para o futuro: na verdade, só podem ser interpretados como booleanos os tipos escalares, ou seja, os tipos numéricos (inteiro e ponto flutuante), além dos ponteiros, que só veremos no capítulo 5.

```
a = b;
if (b != 0)
```

que, claramente, não tem nada a ver com a igualdade (prévia) entre *a* e *b*.

REVISADO ATÉ AQUI — 05/02/2011

Funções 3

3.1 Introdução

Em C, uma **função** é um pedaço de código, dentro de um programa maior, que realiza uma certa tarefa com uma certa independência do resto do programa. Funções podem ser executadas várias vezes, e uma grande vantagem disso é a *reutilização de código*: em vez de repetir várias vezes o código para executar certa tarefa, podemos simplesmente chamar várias vezes a função que executa essa tarefa. Além de economizar linhas de código, isso permite que você mude facilmente o código associado a essa tarefa — se não fosse pelas funções, você teria de buscar em seu programa por todos os locais em que você executou essa tarefa e alterar o código em cada um. Mais ainda, ao organizarmos o código em várias funções, podemos focar cada parte do código em uma só tarefa, deixando o programa mais claro e limpo.

Em C, uma função deve ter as seguintes características:

- Um *nome* pela qual ela possa ser chamada. Os nomes possíveis seguem as mesmas restrições que os nomes de variáveis: devem começar com uma letra ou com um sublinhado (_), e podem conter qualquer combinação desses e dos algarismos 0–9. Lembre-se de que há distinção entre maiúsculas e minúsculas.

- Valores de *entrada* ou *parâmetros* — são os valores sobre os quais a função deve operar. Os parâmetros das funções (também chamados de *argumentos*) atuam de maneira análoga às *variáveis* das funções matemáticas.

  Também é possível criar funções sem argumentos — por exemplo, se você quiser criar uma função que calcula o valor (aproximado, é claro) do número π, não precisa de nenhum parâmetro (a princípio; você poderia introduzir parâmetros se quisesse — por exemplo, a precisão desejada —, mas eles não são necessários se você quer executar a operação de uma maneira pré-determinada).

- Um valor de *saída*, que corresponde ao resultado da função para o conjunto dos parâmetros de entrada fornecidos. Também é possível criar funções que não devolvem nenhum valor de saída. Por exemplo, uma função que simplesmente exibe uma mensagem na tela não precisa devolver nenhum valor — embora a função *tenha* um resultado, ele é mostrado na tela, e não devolvido internamente para o programa.

Burocraticamente, ao se definir uma função, precisamos sempre especificar todas essas características: o nome da função, a quantidade de parâmetros e o tipo de cada um, além do tipo do valor de saída (caso haja valor de saída). E, é claro, você deve definir o que a função vai fazer.

Para definir uma função, usamos a seguinte estrutura:

```
tipo_da_saída nome_da_função (parâmetros)
{
    conteúdo da função;
}
```

Essa definição deve ser colocada no "nível superior" do arquivo, ou seja, não deve estar dentro de outra função como o main. Todas as funções dentro de um arquivo devem ficar no mesmo nível, cada uma após o final da anterior. Na verdade, enquanto eu não falar de uma outra coisa (ainda neste capítulo), as demais funções devem ficar **antes** da função main.

O tipo do valor de saída pode ser qualquer um dos tipos usados para variáveis (por enquanto, você só conhece o int). No caso em que não há valor de saída, você deve usar no lugar do tipo a palavra `void` (*vazio*, em inglês). Ela não é um tipo de variável; ela apenas indica a *ausência* de um valor. (Muitos falam do "tipo void", mas isso é apenas um abuso de linguagem.)

A definição dos parâmetros é semelhante à declaração de variáveis. Cada parâmetro deve ter um nome (seguindo, novamente, as mesmas restrições válidas para os nomes de variáveis) e um tipo. Para especificar esses parâmetros, você deve usar o formato

tipo_1 *nome_1*, **tipo_2** *nome_2*, …, **tipo_n** *nome_n*

Note que, nesse caso, não existe nenhum tipo de abreviação para vários parâmetros do mesmo tipo (como ocorria na declaração de variáveis). No caso de não haver parâmetros, você deve usar a palavra **void** sozinha dentro dos parênteses:

tipo_da_saída **nome_da_função** (**void**)

Atenção: segundo muitos textos, em caso de ausência de parâmetros bastaria deixar os parênteses "vazios", sem nada no meio. Segundo o padrão da linguagem, nesse caso o compilador apenas entenderia que *não há informação* sobre os parâmetros; mesmo assim, isso costuma ser aceito pela maioria dos compiladores.

O conjunto dessas três definições — do nome, do tipo de saída e da lista de parâmetros da função — é chamado de *cabeçalho* da função. Vamos construir alguns exemplos de cabeçalhos:

- Uma função que calcula a soma dos divisores de um número inteiro `n`. Como entrada, teremos obviamente o número `n`, que será uma variável do tipo int. Como saída, teremos outro valor do tipo int, que corresponderá á soma dos divisores de `n`. Com isso, o cabeçalho fica

  ```
  int soma_divisores(int n)
  ```

- Uma função que recebe dois números inteiros, `a` e `b`, e devolve o valor da potência a^b. Novamente, todos os valores envolvidos são do tipo int, e nosso cabeçalho vem a ser

```
int potencia(int a, int b)
```

- Você está criando uma função que recebe um mês e um ano e imprime na tela o calendário desse mês. Nesse caso, não há nenhum valor de saída (os dados são enviados diretamente para a tela, com a função printf), o que indica que usaremos a palavra void no lugar do tipo da saída; mas há dois parâmetros do tipo int, então o cabeçalho fica assim:

```
void imprime_calendario(int mes, int ano)
```

- Suponha agora que você quer fazer uma função que lê um inteiro do teclado (usando a função scanf como intermediária). Para isso, você não precisa de nenhum parâmetro de entrada; você simplesmente devolverá o número lido.

```
int le_inteiro(void)
```

Agora, o prato principal: como escrever o *conteúdo* da função? Isso é o menor dos mistérios — tudo que você precisaria saber sobre isso já foi feito na função main, que é uma função (quase) como outra qualquer. Basta colocar um par de chaves após o cabeçalho e colocar no meio das chaves tudo o que você souber fazer em C: declarar variáveis, fazer contas, chamar as funções scanf e printf, usar laços e controles, etc.

Antes de poder criar exemplos concretos, precisamos ver um pouco mais de teoria.

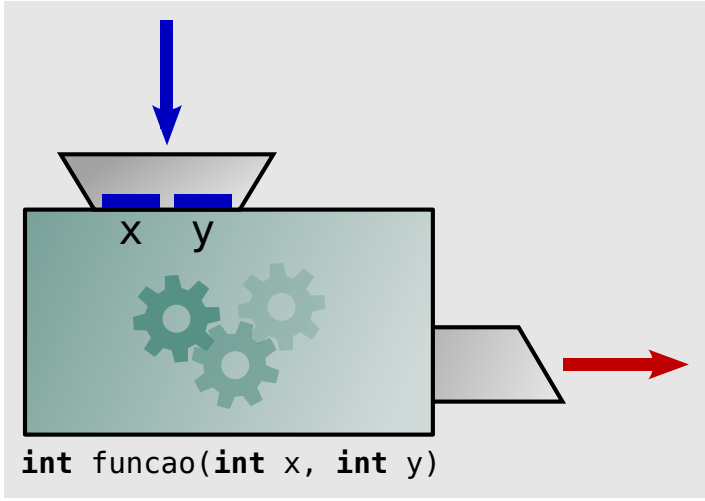


Figura 3.1: Representação ilustrativa de uma função.

3.2 Os parâmetros e o valor de saída

Já vimos como definir as entradas e saídas de uma função; agora precisamos saber lidar com elas: como acessar os parâmetros de entrada e como devolver o valor de saída. A primeira

tarefa é muito simples: uma vez definidos os parâmetros no cabeçalho da função, você pode acessá-los como se fossem variáveis normais. Por exemplo, se quiséssemos criar uma função que recebe dois inteiros e imprime sua soma na tela, poderíamos escrever:

```
void imprime_soma(int a, int b)
{
   int soma;
   soma = a + b;
   printf("%d\n", soma);
}
```

Veja que nesse caso a função não tem nenhum resultado a devolver para o programa, então usamos a palavra void para o "tipo" de saída. Lembre-se de que não é necessário criar uma variável intermediária para fazer essa conta. Poderíamos ter escrito apenas

```
void imprime_soma(int a, int b)
{
   printf("%d\n", a + b);
}
```

Para devolver o valor de saída, usamos a instrução **return** seguida do valor de saída (terminando com um ponto-e-vírgula). O valor pode ser qualquer expressão que seja legítima de se colocar no lado direito de uma atribuição: o valor de uma variável, uma constante numérica, uma expressão aritmética, etc. Por exemplo:

```
return 0;
return x*x;
return y + 1;
```

Vamos ver um exemplo mais concreto. A função a seguir devolve para o programa a soma dos dois números recebidos como parâmetros:

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

É importante ressaltar que a instrução return também *encerra a execução da função*, ou seja, você só pode executá-la quando não houver mais nada a fazer dentro da função. Se você colocar uma instrução return no meio da função, ela devolverá o valor indicado e ignorará todo o resto da função.

Vale também salientar que uma função void *não pode* devolver nenhum valor. Seria um erro escrever algo do tipo `return 0` numa função void — da mesma maneira, é errado usar uma instrução return sem valor numa função que não é void. No entanto, você pode usar a instrução return (sem nenhum valor) para terminar uma função void no meio. Por exemplo:

```
void imprime_numero(int n)
{
   if (n < 0) {
      printf("Não quero imprimir números negativos!\n");
      return;
```

```
    }

    printf("%d\n", n);
}
```

Vamos criar um exemplo mais elaborado: uma função que calcula a potência a^b, dados dois inteiros a e b. (Para deixar o código mais claro, chamamos a de *base* e b de *expoente*.) Veja que o código da função é essencialmente o mesmo que havíamos criado antes.

```
int potencia(int base, int expoente)
{
    int pot, i;

    pot = 1;
    for (i = 1; i <= expoente; i++)
        pot *= base;
    return pot;
}
```

3.3 Usando funções

Já sabemos como criar funções; mas como fazemos para usá-las? Tendo em mãos o nome da função e a lista de valores que desejamos mandar como parâmetros de entrada, a "fórmula" para chamar uma função é simples, e não depende de a função ter ou não um valor de saída:

nome_da_função (*parâmetro_1*, *parâmetro_2*, …)

Caso a função não tenha nenhum parâmetro, simplesmente deixe os parênteses sozinhos sem nada no meio:

nome_da_função ()

Esse tipo de comando é uma **chamada de função**; ele simplesmente faz com que o computador pule para a função chamada, execute-a por inteiro e depois volte para o mesmo ponto de onde saiu.

Se a função tiver um valor de saída, provavelmente vamos querer aproveitá-lo — nesse caso, basta colocar essa chamada de função no meio de uma expressão qualquer; por exemplo, podemos guardá-lo numa variável, colocá-lo no meio de uma expressão aritmética ou mesmo mandá-lo como parâmetro para outra função. Um exemplo, utilizando duas funções que já escrevemos acima:

```
int a, b, c;
a = potencia(2, 3);
b = soma(a, 8);
c = potencia(3, soma(b, a) + b);
```

Se não houver valor de saída, simplesmente colocamos a chamada de função com o tradicional ponto-e-vírgula no final. Por exemplo:

```
    int a, b;
    a = potencia(2, 3);
    b = soma(a, 8);
    imprime_soma(a, b + potencia(a, b));
```

Vamos ver um exemplo de programa completo usando funções. O funcionamento dele será bem simples: leremos um par de inteiros do teclado e calcularemos o primeiro elevado ao segundo (o segundo deve ser positivo!), usando a função potencia já escrita anteriormente:

```
#include <stdio.h>

int potencia(int base, int expoente)
{
   int pot, i;

   pot = 1;
   for (i = 1; i <= expoente; i++)
      pot *= base;
   return pot;
}

int main()
{
   int base, expoente;

   printf("Digite a base: ");
   scanf("%d", &base);

   printf("Digite o expoente (inteiro positivo): ");
   scanf("%d", &expoente);

   printf("Resultado: %d\n", potencia(base, expoente));

   return 0;
}
```

Veja que a função potencia foi colocada antes da main, como já observamos que seria necessário. Se você trocasse a ordem das funções, receberia uma mensagem de erro do compilador; veremos a seguir por que isso ocorre e uma outra maneira de definir as funções que, de certa maneira, elimina essa limitação.

3.4 Trabalhando com várias funções

A linguagem C é bastante rígida quanto aos tipos dos objetos (variáveis e funções) usados nos programas: toda variável deve ser declarada antes de ser utilizada, para que o computador saiba como organizar a memória para o acesso a essa variável. Da mesma maneira, para que o computador saiba como organizar o "trânsito" dos valores de entrada e saída, *toda função deve ser declarada* antes de ser chamada.

A princípio, isso não é um grande problema, pois, quando fazemos a definição de uma função (isto é, escrevemos seu cabeçalho e logo em seguida seu conteúdo), a declaração é feita implicitamente. No entanto, essa declaração só vale para o que vier depois dela; então uma função não pode ser usada dentro de outra função que vem antes dela. Por isso que eu disse que todas as funções deveriam ser definidas antes da main; se chamamos a função potencia dentro da função main e a definição de potencia só vem depois de main, o compilador não tem nenhuma informação sobre quem é potencia, então não tem como chamar essa função corretamente.

Por esse motivo, uma função pode ser declarada explicitamente antes de ser propriamente definida. A declaração explícita de uma função é bastante simples: basta reescrever o cabeçalho da função, seguido de um *ponto-e-vírgula*:

tipo_da_saída **nome_da_função** (*parâmetros*);

Por exemplo, para declarar a função potencia que criamos anteriormente, escreveríamos simplesmente

```
int potencia(int base, int expoente);
```

Já vimos que não é estritamente necessário declarar explicitamente as funções em todos os casos: em várias situações é possível contornar o problema com uma simples reordenação das funções. No entanto, com programas maiores e mais complexos, é sempre uma boa idéia declarar todas as funções no começo do arquivo, e você é encorajado a sempre fazê-lo — isso nos permite uma melhor organização do código, sem precisar nos prender à ordem de dependência das funções; além disso, a lista de declarações pode servir como um pequeno "índice" das funções que foram definidas num certo arquivo.

3.5 Escopo de variáveis

Uma característica bastante útil (e importante) da linguagem C é que as funções são totalmente independentes quanto às variáveis declaradas dentro delas. Se você declara uma variável dentro de uma função — essas são chamadas de **variáveis locais** —, ela só existe dentro da própria função; quando a função termina sua execução, as variáveis locais são, de certa maneira, perdidas. Nenhuma operação feita com variáveis locais poderá afetar outras funções. Ainda mais, você pode declarar variáveis locais com o mesmo nome em funções diferentes, sem o menor problema; cada função saberá qual é a sua variável e não mexerá nas variáveis das outras funções.

É bom ressaltar o papel dos parâmetros de funções nessa história: eles funcionam exatamente como as variáveis locais, com a diferença de que seus valores são atribuídos de forma implícita. Cada vez que você chama uma função com um certo conjunto de parâmetros, os valores desses parâmetros são *copiados* para a função chamada. Assim, você também pode modificar os valores dos parâmetros de uma função, e nada mudará na função que a chamou.

Um exemplo clássico para ilustrar esse comportamento é uma tentativa de criar uma função que troca o valor de duas variáveis:

```
#include <stdio.h>

void troca(int a, int b)
{
   int temp;
```

```
    printf("função troca - antes:  a = %d, b = %d\n", a, b);
    temp = a;
    a = b;
    b = temp;
    printf("função troca - depois: a = %d, b = %d\n", a, b);
}

int main()
{
    int x, y;
    x = 10;
    y = 5;
    printf("main - antes:  x = %d, y = %d\n", x, y);
    troca(x, y);
    printf("main - depois: x = %d, y = %d\n", x, y);
    return 0;
}
```

Se você executar esse exemplo, verá que o valor das variáveis *x* e *y* dentro da função main não se altera. A saída desse programa será:

```
main         - antes:  x = 10,  y = 5
função troca - antes:  a = 10,  b = 5
função troca - depois: a = 5,   b = 10
main         - depois: x = 10,  y = 5
```

Você deve entender por que isso acontece: quando trocamos as variáveis *a* e *b*, estamos nada mais que trocando de lugar as *cópias* das variáveis originais *x* e *y*. Quando a função troca é encerrada, essas cópias são inutilizadas e a função main volta a ser executada. Não fizemos nada explicitamente com as variáveis da função main; não há motivo para que elas mudem.

Existe, sim, uma maneira de criar uma função que troque o valor de duas variáveis — ela envolve o uso de *ponteiros*, conforme veremos no Capítulo 5.

As variáveis locais podem ser declaradas não apenas dentro de funções, mas dentro de laços ou estruturas de controle (if, while, for, etc.). Isso quer dizer que podemos criar variáveis que só existem dentro de um bloco if (por exemplo), e não podem ser acessadas fora dele, mesmo dentro da mesma função. O contexto em que uma variável existe e pode ser acessada é denominado **escopo**. Então dizemos que o escopo de uma certa variável é um certo bloco ou uma certa função.

Mesmo podendo declarar variáveis dentro de qualquer bloco (**bloco** é como se costuma designar genericamente qualquer estrutura de comandos delimitada por { }, seja uma função ou uma estrutura de controle), continua valendo a restrição de que as declarações de variáveis devem vir no *começo* do bloco correspondente, não podendo haver nenhum outro tipo de comando antes dessas declarações.

Mais sobre números 4

4.1 Bases de numeração

Ao criar uma máquina que faz cálculos, nada é mais crucial que a escolha da maneira de representação dos números ou do formato em que eles serão armazenados. Para nós, o sistema decimal parece ser o mais natural — contamos com os 10 dedos das mãos (uma possível explicação para o uso da base 10 e não de outra); usamos as potências de 10 (dez, cem, mil) como referência ao falar os números por extenso.

No entanto, os componentes de um computador funcionam com base em correntes elétricas, e com eles é muito mais simples implementar um sistema de numeração **binária** (ou **base 2**), no qual existem apenas dois dígitos (0 e 1), correspondendo, por exemplo, a um nível baixo ou nulo (0) e a um nível alto (1) de corrente ou voltagem.

Se tomarmos um instante para compreender como a nossa usual base 10 funciona, será mais fácil entender como a base 2 funciona. Tomemos um número; por exemplo, 41 502. Ele pode ser também escrito como

$$\begin{aligned} 41502 &= 40000 + 1000 + 500 + 2 \\ &= 4 \cdot 10000 + 1 \cdot 1000 + 5 \cdot 100 + 0 \cdot 100 + 2 \cdot 1 \\ &= 4 \cdot 10^4 + 1 \cdot 10^3 + 5 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0 \end{aligned}$$

Veja, então, que cada algarismo funciona como *um multiplicador de uma potência de 10*. Os expoentes são contados a partir da direita, começando pelo zero. Assim, o primeiro algarismo (a partir da direita) é multiplicado por 10^0, o segundo por 10^1, e assim por diante. Perceba a regrinha implícita do sistema: o valor de cada algarismo é sempre menor que a base (o maior algarismo, 9, é menor que a base, 10) e maior ou igual a zero. Isso é necessário para que cada número tenha uma única representação nessa base.

Para lembrar como são as regras para se contar, pense em um contador analógico (por exemplo, o odômetro dos carros mais antigos, ou os medidores de consumo de água e energia elétrica nas casas), daqueles que mostram cada algarismo em uma rodinha. Cada rodinha marca um algarismo de 0 a 9 e corresponde a uma ordem de grandeza (1, 10, 100, …). Sempre que uma das rodinhas dá uma volta completa (passando do 9 para o 0), a rodinha à sua esquerda aumenta seu dígito. Se esse dígito já era 9, o mesmo processo é repetido sucessivamente para as outras rodinhas da esquerda.

Sabendo isso, podemos, por analogia, proceder à base 2. As regras são, pois, as seguintes:

- Como cada algarismo deve ter um valor menor que o da base e maior e igual a zero, só podemos ter os algarismos 0 e 1, como você provavelmente já sabia. Por isso, contar em binário é bastante simples — pense num contador binário, no qual cada rodinha só tem duas posições: 0 e 1. Quando uma rodinha passa do 1 para o 0, a próxima rodinha (a da esquerda) se movimenta: se estiver no 0, passa para o 1; se estiver no 1, passa para o 0, "propagando" o movimento para a esquerda.

  Os números de um a dez escrevem-se assim em binário:

  $$1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010$$

  Os algarismos binários, como já disse, costumam corresponder ao estado de um componente quanto à passagem de corrente. Esses algarismos são geralmente chamados de **bits** — abreviação de *binary digit*.

- O valor do número é obtido multiplicando-se cada algarismo por uma potência de 2 (e não mais de 10) de acordo com sua posição. Assim, o número 1101101 poderia ser "traduzido" para o sistema decimal da seguinte maneira:

  $$(1101101)_2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (109)_{10}$$

  Veja que usamos a notação $(\cdots)_2$ ou $(\cdots)_{10}$ para especificar a base em que estamos escrevendo. (Quando não houver indicação, ficará entendido que o número está na base decimal.)

4.1.1 Uma base genérica

Podemos generalizar esse raciocínio para qualquer base de numeração b; outras bases comumente usadas em computação são a octal ($b = 8$) e hexadecimal ($b = 16$). Se um número é representado como uma sequência de $n + 1$ dígitos, $a_n a_{n-1} \cdots a_1 a_0$, seu valor é dado por

$$a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0 = \sum_{k=0}^{n} a_k b^k$$

Para que a representação de um número seja única, é necessário que os valores de todos os seus algarismos sejam menores do que b, ou seja, no máximo $b-1$. Caso contrário, o número b^2, por exemplo, poderia ser representado pelas duas sequências $(b, 0)$ e $(1, 0, 0)$. Os valores dos dígitos também não podem ser negativos — senão surgiria mais uma ambiguidade na representação.

É importante nesse ponto não confundir os *valores* dos dígitos com as suas *representações* (que costumam ser os algarismos arábicos sempre que possível). Aqui, os a_j simbolizam os *valores* dos dígitos (como objetos matemáticos abstratos), independentemente de sua representação. Quando a base não ultrapassa 10, os valores confundem-se com suas representações (usamos os algarismos de 0 a $b - 1$ para representar os valores de 0 a $b - 1$). Quando a base é maior que 10, precisamos de outra convenção — para a base hexadecimal, por exemplo, os dígitos com valores de 10 a 15 são representados pelas seis primeiras letras do alfabeto, enquanto os dígitos até 9 continuam sendo representados da maneira usual.

A fórmula acima nos permite, dados os valores dos dígitos de um número, achar o valor do número. Podemos também querer realizar o processo inverso: dado um número, achar

os valores dos seus dígitos na representação em base b. Olhando para a fórmula acima, podemos ver que, ao dividir (com resto) o valor do número por b, obtemos como resto o valor de a_0; o quociente é o novo número

$$a_n b^{n-1} + a_{n-1} b^{n-2} + \cdots + a_1$$

E, ao dividi-lo por b, obteremos como resto o próximo dígito, a_1 (que pode muito bem ser zero). Não é difícil concluir que, se realizarmos $n+1$ divisões por b, os restos serão, na ordem, os dígitos

$$(a_0, a_1, a_2, \ldots, a_{n-1}, a_n).$$

Ou seja, dividindo repetidamente por b, obteremos os dígitos *da direita para a esquerda*. Note que, na última divisão, obtemos a_n como resto e zero como quociente.

Para ilustrar esse algoritmo, vamos fazer uma função que recebe um número inteiro e imprime seus dígitos na representação binária.

```
void imprime_binario(int numero)
{
   while (numero > 0) {
      printf("%d", numero % 2);
      numero /= 2;
   }
   printf("\n");
}
```

Notemos, primeiro, que essa função fornece os dígitos na ordem inversa (conforme foi observado acima); por exemplo, o número $12 = (1100)_2$ seria impresso como `0011`.

Veja também que, em vez de fixar o número de divisões, estabelecemos como critério de parada o anulamento do quociente. Com isso, não precisamos saber de antemão quantos dígitos binários tem o número; vimos que o quociente será zero quando só sobrar o dígito mais significativo.

4.2 O armazenamento dos dados

Ao armazenar um número na memória do computador, precisamos estabelecer um tamanho padronizado (uma espécie de *número de algarismos*) que um número ocupará. Por que isso? Imagine que a memória do computador é uma enorme aglomeração de rodinhas numeradas. Se não tivesse um tamanho padrão, como saberíamos onde começa e onde termina cada número? Precisaríamos de algum *outro número* para saber onde cada número começaria, e essa história não terminaria nunca.

Num computador, a menor unidade possível de informação é o bit; mas a menor unidade de informação que pode ser de fato acessada é o **byte**, que, nos microcomputadores modernos, equivale a 8 bits. No entanto, um byte sozinho não serve para muita coisa; números inteiros costumam ser armazenados em espaços de 1, 2, 4 ou 8 bytes (ou seja, 8, 16, 32 ou 64 bits, respectivamente).

Apesar de o tamanho de um inteiro ser padronizado, ainda é possível que existam inteiros de mais de um tamanho. Por exemplo, em C existem pelo menos três tipos inteiros, de tamanhos diferentes. O que ocorre é que cada processador trabalha naturalmente com um certo tamanho de inteiro (usualmente, 32 ou 64 bits) — todos os seus espaços internos de

armazenamento (eles se chamam *registradores*) têm esse mesmo tamanho. Quando queremos usar inteiros de tamanhos diferentes, o processador simplesmente trabalha com pedaços desses tais registradores ou com vários registradores juntos.

Agora voltemos um pouco ao C. Já conhecemos os dois tipos básicos de inteiros: `int` e, embora não o tenhamos usado ainda, `char` (como vimos no começo do primeiro capítulo, os caracteres são armazenados como se fossem inteiros). O tipo `char` tem um tamanho único: um byte — ou seja, 8 bits. O tipo `int` geralmente tem um tamanho padrão de 32 bits, mas ele também possui alguns "subtipos" com tamanhos diferentes. São eles:

- `short int`, um "inteiro curto", que costuma ter 16 bits;
- `long int`, um "inteiro longo", que costuma ter 32 bits (não há nada de errado aqui; o tipo `int` realmente costuma ser igual ao `long`).
- `long long int`, que geralmente tem 64 bits. (Somente no padrão C99.)

Nota: No padrão da linguagem C, os tamanhos desses tipos de dados são definidos de uma maneira mais formal (e mais vaga), devido à grande diferença entre os vários tipos de sistemas em que o C pode ser usado. Eu os defini de uma maneira mais prática, segundo os tamanhos que esses tipos têm na maioria dos computadores pessoais de hoje em dia; uma definição mais correta pode ser encontrada na especificação da linguagem.

Todos esses subtipos podem (e costumam) ser abreviados, tirando deles a palavra `int`. Assim, `short int` costuma ser escrito como simplesmente `short` (e assim por diante).

Quão grandes são esses tamanhos? Que valores cabem num inteiro de 32 ou 16 bits (ou de qualquer outro número n)? Para isso, podemos pensar no menor e no maior número que pode ser representado com n bits. O resultado é análogo ao caso decimal: se temos k algarismos, o maior número que podemos representar é uma sequência de k algarismos 9, que é igual a $10^k - 1$. Da mesma maneira, com n bits, o maior número representável (em binário) é $2^n - 1$, que corresponde a uma sequência de n algarismos 1.[6]

Assim, com 32 bits, por exemplo, podemos representar todos os números de 0 a $2^{32} - 1$, que vale 4 294 967 295. Na tabela a seguir você pode ver as capacidades dos tamanhos de inteiros mais comuns.

Tabela 4.1: Os tamanhos mais comuns de inteiros nos microcomputadores atuais, e os maiores números armazenáveis com tais tamanhos.

Tipo	Bits	Bytes	Maior número
`char`	8	1	255
`short`	16	2	65 535
`int`, `long`	32	4	4 294 967 295
`long long`	64	8	18 446 744 073 709 551 615

4.2.1 Números negativos

Talvez no meio disso tudo você tenha se perguntado se existe alguma maneira de escrever números negativos no sistema binário. Numa escrita "humana", o natural seria simplesmente

[6] Você também pode chegar a esses números pensando em termos das nossas somas $\sum a_k b^k$, com todos os $a_k = b - 1$. Lembre da fórmula de soma de uma progressão geométrica e verifique que, de fato, o resultado é o mesmo.

escrever o sinal de menos para os números negativos. Nos computadores, a codificação mais comum de números negativos tem como princípio reservar um dos bits do número para o sinal — assim, um número de 32 bits fica com 31 bits para guardar o módulo do número e 1 bit para guardar o sinal. Geralmente é usado o bit mais significativo (o que estiver mais à esquerda na nossa representação usual), e ele vale 1 para números negativos ou 0 para números positivos. Sabendo isso, talvez você poderia imaginar que, por exemplo, se o número 14 é armazenado em 8 bits como 0000 1110, o número -14 seria armazenado como 1000 1110. Não é bem assim.

O que acontece é que essa representação não é muito eficiente quando o computador vai fazer cálculos. Existe uma outra representação, conhecida como representação do *complemento de 2*, que permite que números negativos sejam operados exatamente da mesma maneira que os números positivos, isto é, sem que o computador precise verificar se um número é negativo para saber como fazer a conta.

"Complemento de 2" não é o melhor nome (seria mais adequado *complemento de* 2^n); mas o método consiste em guardar cada número negativo $-k$ como se fosse o número positivo $2^n - k$, sendo n é o número de bits reservado para o inteiro. Por exemplo, ao trabalhar com 8 bits, o número -14 seria guardado como se fosse $256 - 14 = 242$ (em binário, isso seria 1111 0010).

Agora, se olharmos para o número 128, veremos que ele é o seu próprio complementar quando usamos $n = 8$ bits: $256 - 128 = 128$. Mas lembre-se que a representação binária de 128 é 1000 0000 — como o bit de sinal é 1, faz muito mais sentido convencionar que essa será a representação do número negativo -128. Então, nessa representação, o 128 "não existe" (precisaríamos usar mais que 8 bits); os números positivos param no 127. Os números negativos vão até o -128.

Generalizando essas conclusões, num espaço de n bits é possível guardar, *com sinal*, todos os números inteiros de -2^{n-1} até $2^{n-1} - 1$. (Veja como isso é equivalente ao nosso intervalo de -128 a 127 para a representação de 8 bits.)

Um algoritmo prático para calcular o complemento de 2 de um número, já na representação binária, é o seguinte: preencha com zeros à esquerda para completar a quantidade de bits que está sendo usada; então inverta todos os bits da representação (trocar os 0 por 1 e vice-versa) e some 1. Lembre-se de inverter também os zeros à esquerda que você incluiu. [7]

Em C é possível armazenar tanto números negativos quanto positivos, como já mencionamos no começo. Mais ainda, ao declarar uma variável você pode escolher se quer guardar números negativos e positivos (com sinal) ou só positivos (sem sinal) — como acabamos de ver, isso faz diferença no intervalo de números que pode ser guardado. Assim, uma variável de 8 bits pode suportar números de 0 a 255 ou números de -128 a 127. Para indicar isso, você pode modificar os tipos numéricos inteiros (`char` e `int`) com mais dois "adjetivos": **signed** (com sinal) e **unsigned** (sem sinal). Quando você não diz nada, o computador assume que você quer guardar os números com sinal (signed).

[7] Se você quiser verificar por que os dois métodos são equivalentes, pense na soma de um número com o que tem todos os seus bits invertidos.

Tabela 4.2: As faixas de números armazenáveis nos tipos inteiros com sinal.

Tipo	Região
`char` (8 bits)	−128 a 127
`short` (16 bits)	−32 768 a 32 767
`int`, `long` (32 bits)	−2 147 483 648 a 2 147 483 647
`long long` (64 bits)	−9 223 372 036 854 775 808 a 9 223 372 036 854 775 807

4.3 Números reais

É possível fazer muita coisa só com números inteiros, mas em muitas situações somos obrigados a usar números fracionários ou reais ("números com vírgula"). Em C, os números reais estão encarnados em dois tipos de dados: **float** e **double**. Os dois funcionam mais ou menos do mesmo jeito; a diferença entre eles é a precisão de cada um — dizemos que o *float* tem precisão simples e o *double* tem precisão dupla. Na maioria das linguagens de programação (inclusive C), esses tipos de números são conhecidos como números de **ponto flutuante** — mais adiante veremos o porquê desse nome.

Falamos em precisão porque, assim como no caso dos inteiros, é necessário impor um tamanho para cada número, o que limita a representação dos números — veremos mais adiante como exatamente isso se dá.

Antes de entrar na parte mais teórica, vamos ver como se usam os tais números de ponto flutuante. A declaração de variáveis funciona da mesma maneira que para os inteiros:

```
double x, y;
float z, w;
```

Para escrever números fracionários em C, usamos o ponto (`.`) para separar a parte inteira da fracionária — por exemplo, `3.14159` ou `-0.001`. Podemos também usar uma espécie de notação científica para trabalhar com números muito grandes ou muito pequenos — escrevemos o valor "principal" do número da maneira que acabei de descrever, e usamos a letra `e` ou `E` para indicar a potência de 10 pela qual o número deve ser multiplicado. Alguns exemplos:

`3.14159e-7` $\left(3{,}14159 \times 10^{-7}\right)$ `1.234E+26` $\left(1{,}234 \times 10^{26}\right)$ `4.56e5` $\left(4{,}56 \times 10^{5}\right)$

Veja que não faz diferença se o caractere `E` é maiúsculo ou minúsculo, nem se colocamos o sinal de + nos expoentes positivos.

Tão importante quanto saber escrever esses números é saber imprimi-los na tela ou lê-los do teclado. Usando a função printf, usamos o código `%f` (da mesma maneira que o nosso conhecido `%d`), tanto para o double quanto para o float, como no seguinte exemplo:

```c
#include <stdio.h>

int main()
{
   double x = 5.0;
   float  y = 0.1;
   int    z = 27;

   printf("x = %f\n", x);
```

```
    printf("y = %f, z = %d\n", y, z);

    return 0;
}
```

Para ler números reais do teclado, precisamos dizer se a variável onde vamos guardar é do tipo double ou float. No primeiro caso, usamos o código `%lf`; no segundo, apenas `%f` (como no printf)

```
#include <stdio.h>

int main()
{
    double x, z;
    float  y;

    printf("Digite dois números de ponto flutuante: ");
    scanf("%lf %f", &x, &y);

    z = x + y;
    printf("%f\n", z);

    return 0;
}
```

Veja que podemos fazer contas entre números de ponto flutuante da mesma maneira que fazíamos com os inteiros; podemos até misturar doubles com floats. Mais adiante faremos algumas observações sobre isso.

4.3.1 Mais sobre printf

Quando trabalhamos com números de ponto flutuante, pode ser útil exibir números em notação científica. Para isso, podemos trocar nosso código `%f` por `%e` ou `%E`: o resultado será impresso com a notação que estabelecemos anteriormente, usando um `E` maiúsculo ou minúsculo dependendo do que for especificado. Por exemplo,

```
printf("%e %E\n", 6.0, 0.05);
/* resultado:  6.000000e+00  5.000000E-02 */
```

Mas o printf também tem uma opção muito inteligente, que é a `%g` — ela escolhe o melhor formato entre `%f` e `%e` de acordo com a magnitude do número. Números inteiros não muito grandes são impressos sem casas decimais (e sem ponto); números muito grandes ou muito pequenos são impressos com notação científica. Também podemos usar um `G` maiúsculo se quisermos que o `E` da notação científica saia maiúsculo. Por exemplo:

```
printf("%g %g %G\n", 6.0, 0.00005, 4000000.0);
/* resultado:  6  5e-05  4E+06 */
```

Outra coisa que podemos mudar é o número de casas decimais exibidas após o ponto. O padrão para o formato `%f` é de 6 casas; isso pode ser demais em vários casos — por exemplo,

se você for imprimir preços de produtos. Para que o número de casas decimais exibidas seja n, trocamos o código `%f` por

`%.`*`n`*`f`

Para imprimirmos um número com 2 casas, por exemplo, podemos escrever assim:

```
printf("Preço = %.2f\n", 26.5);
/* resultado:  26.50 */
```

Se colocarmos $n = 0$, o ponto decimal também será apagado. Veja que os números serão sempre arredondados conforme necessário. Por exemplo:

```
printf("%.2f %.0f\n", 26.575, 26.575);
/* resultado:  26.58  27 */
```

4.3.2 A representação

Falamos um pouco acima que tanto o float quanto o double são tipos de *ponto flutuante*. Isso diz respeito à representação dos números reais na memória do computador — trata-se de uma espécie de notação científica. Nessa representação, um número tem duas partes: a *mantissa*, que são os algarismos significativos, e o *expoente*, que indica a posição da vírgula decimal em relação aos algarismos significativos.

Pensando na base decimal, um número como 0,000395 teria a representação $3{,}95 \times 10^{-4}$, cuja mantissa é 3,95 e cujo expoente é -4 (indica que a vírgula deve ser deslocada 4 dígitos para a esquerda). Na notação científica, estabelecemos que a mantissa deve ser maior que ou igual a 1, mas não pode ter mais de um dígito à esquerda da vírgula — por exemplo, a mantissa pode ser 1,00 ou 9,99, mas não 15,07 ou 0,03. Assim, estamos de fato "dividindo" a informação do número em duas partes: o expoente dá a ordem de grandeza do número, e a mantissa dá o valor real dentro dessa ordem de grandeza.

Nem todo número tem uma representação decimal finita — em outras palavras, nem todo número tem uma mantissa finita. Por exemplo, a fração $5/3$ pode ser expandida infinitamente como $1{,}6666\ldots$; se quisermos expressá-la com um número finito de dígitos (como é necessário em um computador), devemos parar em algum dígito e arredondar conforme necessário — por exemplo, 1,6667, se precisarmos trabalhar com 5 dígitos.

Assim, para colocar um número nessa representação, é necessário primeiro normalizar a mantissa para que fique entre 1 (inclusive) e 10, e depois arredondá-la para um número de dígitos pré-estabelecido.

A representação usual de ponto flutuante no computador é praticamente igual a essa; antes de ver como ela realmente funciona, vamos ver como funciona a representação dos números fracionários na base binária.

O que é a representação decimal de um número? Considere um número que pode ser escrito como $a_n a_{n-1} \cdots a_1 a_0{,}b_1 b_2 \cdots$ (veja que a parte inteira, à esquerda da vírgula, tem um número finito de dígitos; a parte fracionária, à direita da vírgula, pode ter infinitos dígitos). Já vimos como funciona a representação da parte inteira; falta analisarmos a parte fracionária.

Vejamos primeiro o caso de um número cuja parte fracionária é finita e tem m dígitos — não vamos nos importar com a parte inteira; suponha que é zero. Esse número pode ser escrito como $0{,}b_1 b_2 \cdots b_m$. Então, se multiplicarmos esse número por 10^m, ele voltará

a ser inteiro, e será escrito como $b_1 b_2 \cdots b_m$. Já sabemos como funciona a representação inteira de um número! Esse número vale

$$b_1 10^{m-1} + b_2 10^{m-2} + \cdots + b_{m-1} 10^1 + b_m 10^0 = \sum_{k=1}^{m} b_k 10^{m-k}$$

Como esse é o número original multiplicado por 10^m, o número original vale

$$b_1 10^{-1} + b_2 10^{-2} + \cdots + b_{m-1} 10^{-(m-1)} + b_m 10^{-m} = \sum_{k=1}^{m} b_k 10^{-k}$$

Não é difícil estender essas contas para os casos em que a parte fracionária não é finita (ou em que a parte inteira não é zero), mas seria muito tedioso reproduzi-las aqui; deixá-las-ei como exercício se você quiser pensar um pouco, e direi apenas que aquela representação (possivelmente infinita) $a_n a_{n-1} \cdots a_1 a_0, b_1 b_2 \cdots$ corresponde ao número

$$\sum_{k=0}^{n} a_k 10^k + \sum_{k=1}^{\infty} b_k 10^{-k}$$

Como no caso dos inteiros, é claro que todos os dígitos a_k e b_k só podem assumir os valores $\{0, 1, \ldots, 9\}$, para que a representação de cada número seja única. (Na verdade isso não é suficiente para que a representação seja única; precisamos, para isso, exigir que haja um número *infinito* de dígitos diferentes de 9, evitando assim as representações do tipo $0{,}999999\ldots \equiv 1$.)

Veja que essa representação é uma extensão da representação dos inteiros; poderíamos trocar os b's por a's com índices negativos ($b_k = a_{-k}$) e estender a somatória para os índices negativos:

$$\sum_{k=-\infty}^{n} a_k 10^k$$

Com isso, a transição para a base binária está muito bem encaminhada. O que mudará na base binária é que os dígitos são multiplicados por potências de 2 (o dígito a_k é multiplicado por 2^k), e só podem assumir os valores 0 e 1.

Por exemplo, o número $3/8 = 1/8 + 1/4 = 2^{-2} + 2^{-3}$ poderá ser representado como $(0{,}011)_2$. E o número $1/10$, que é aparentemente muito simples? Na base 2, ele não tem uma representação finita! É fácil ver o porquê: se ele tivesse representação finita, bastaria multiplicar por uma potência de 2 para torná-lo inteiro; sabemos que isso não é verdade (nenhuma potência de 2 é divisível por 10!).

Como descobrir a representação binária de $1/10$? Vou falar sobre dois jeitos de fazer isso. O primeiro envolve alguns truquezinhos algébricos. Vamos escrever

$$\frac{1}{10} = \frac{1}{2} \cdot \frac{1}{5} = \frac{1}{2} \cdot \frac{3}{15} = \frac{3}{2} \cdot \frac{1}{2^4 - 1} = \frac{3}{2} \cdot 2^{-4} \cdot \frac{1}{1 - 2^{-4}}$$

Agora a última fração é a soma da série geométrica de razão 2^{-4}. Podemos então escrever

$$\begin{aligned}
\frac{1}{10} &= 3 \cdot 2^{-5} \cdot \left(1 + 2^{-4} + 2^{-8} + \cdots\right) \\
&= (1+2) \cdot 2^{-5} \cdot \left(1 + 2^{-4} + 2^{-8} + \cdots\right) \\
&= \left(2^{-4} + 2^{-5}\right) \cdot \left(1 + 2^{-4} + 2^{-8} + \cdots\right) \\
\frac{1}{10} &= \left(2^{-4} + 2^{-8} + 2^{-12} + \cdots\right) + \left(2^{-5} + 2^{-9} + 2^{-13} + \cdots\right)
\end{aligned}$$

Assim, identificando esse resultado com a "fórmula" da representação de um número, temos $b_k = 1$ se $k = 4, 8, 12, \ldots$ ou se $k = 5, 9, 13, \ldots$, e $b_k = 0$ caso contrário. Ou seja, os dígitos nas posições $4, 5, 8, 9, 12, 13, \ldots$ são 1, e os outros são 0. Logo, a representação binária de $1/10$ é

$$\frac{1}{10} = (0{,}0\,0011\,0011\,0011\ldots)_2$$

A partir dessas contas, podemos ver alguns padrões interessantes (em analogia com a base 10). Um deles é que multiplicar e dividir por 2 tem, na base binária, o mesmo efeito que tinham as multiplicações/divisões por 10 na base decimal. Ou seja, ao multiplicar por 2 um número, a vírgula vai uma posição para a direita na representação binária (ou acrescenta-se um zero caso o número seja inteiro).

Outro padrão é que frações do tipo $\frac{a}{2^n-1}$, em que $a < 2^n - 1$ é um inteiro, são ótimas geradoras de dízimas periódicas (binárias): podemos escrevê-las como

$$\frac{a}{2^n-1} = \frac{a}{2^n}\frac{1}{1-2^{-n}} = \frac{a}{2^n}\left(1 + 2^{-n} + 2^{-2n} + \cdots\right)$$

Como a representação de a tem no máximo n dígitos, e a série geométrica à direita é uma dízima de período n, a representação da nossa fração será uma dízima cujo período é a representação de a (com zeros à esquerda para completar os n dígitos se necessário).

O outro método de achar a representação decimal de uma fração você já conhece: é o algoritmo da divisão. Para aplicá-lo ao caso das frações, basta escrever o denominador e o numerador em binário e fazer o processo de divisão, lembrando que as regras do jogo mudam um pouco — por exemplo, você só pode multiplicar o divisor por 0 ou 1. Não vou ensinar os detalhes aqui — não é o propósito deste livro. Mas é basicamente este o algoritmo usado pelo computador para converter números fracionários para a representação binária; é assim que ele pode controlar quantos dígitos serão mantidos na representação — basta parar o algoritmo na casa desejada.

Fizemos uma longa digressão sobre representação binária de números. Para que isso serviu? Agora poderemos entender melhor como funciona a representação de ponto flutuante. Como dissemos acima, na representação de ponto flutuante um número tem duas partes: a mantissa e o expoente. Para a mantissa M devemos impor uma condição de normalização como no caso da notação científica decimal: devemos ter $1 \leq M < 2$ para que à esquerda da vírgula haja um e apenas um dígito (não-nulo). Assim, uma fração como $21/4$ seria normalizada da seguinte maneira:

$$\frac{21}{4} = \frac{21}{16} \times 4 = \frac{(10101)_2}{2^4} \times 2^2 = (1{,}0101)_2 \times 2^2$$

Devemos considerar que a mantissa deve ser representada com um tamanho fixo (e finito) de dígitos. No caso da precisão simples do float, esse número costuma ser de 24 dígitos. Assim, muitas frações precisam ser arredondadas — tanto as que têm representações infinitas quanto as que têm representações finitas porém muito longas. Por exemplo, a fração $1/10$ (representação infinita) seria arredondada para

$$(1{,}100\,1100\,1100\,1100\,1100\,1101)_2 \times 2^{-4} = 0{,}100000001490116119384765625$$

Agora vamos chegar mais perto da representação que é realmente usada pelo computador. Um número de ponto flutuante é representado com a seguinte estrutura:

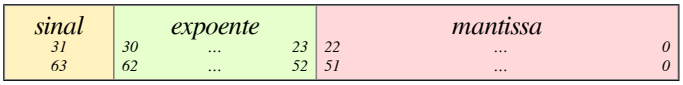


Figura 4.1: Esboço da estrutura de representação de ponto flutuante. Em cada parte foram indicados os números dos bits utilizados nas representações de precisão simples e dupla, respectivamente.

Eu ainda não havia falado do sinal: a representação de ponto flutuante usa um bit para o sinal, assim como na representação de inteiros. Mas, ao contrário desta, em ponto flutuante não se usa nada parecido com o "complemento de 2"; a única diferença entre as representações de dois números de mesma magnitude e sinais opostos é o bit de sinal.

O expoente (relacionado a uma potência de 2, não de 10!) é representado (quase) como um número inteiro comum, e por isso há uma limitação nos valores que ele pode assumir. Na precisão simples, são reservados 8 bits para o expoente, o que permite até 256 valores possíveis — não são exatamente de -128 a 127; na verdade, os dois extremos são reservados para situações especiais; os expoentes representáveis são de -127 até 126. Na precisão dupla, usam-se 11 bits, e a gama de expoentes é muito maior: de -1023 até 1022.

Sobre a mantissa não resta muito a falar; os bits da mantissa são armazenados sequencialmente, como na representação binária que construímos acima. Em precisão simples, são reservados 23 bits para a mantissa. O leitor atento notará que eu falei em 24 dígitos alguns parágrafos atrás. De fato, a representação binária só reserva 23 dígitos. Mas, como a mantissa está sempre entre 1 e 2, à esquerda da vírgula temos sempre um algarismo 1 sozinho; então, convenciona-se que esse algarismo não será escrito (o que nos deixa com 1 bit a mais!), e assim temos uma mantissa de 24 dígitos em um espaço de apenas 23 bits. Na precisão dupla, são reservados 52 bits (que na verdade representam 53).

Na verdade, há vários outros detalhes sobre a representação de ponto flutuante; poderia gastar mais algumas páginas com essa descrição, mas não vem ao caso. Meu objetivo era dar uma idéia geral do funcionamento dessa representação.

Até agora, só explorei a parte teórica dessa representação. Precisamos conhecer também as características práticas da representação de ponto flutuante. Por exemplo, em termos da representação decimal, como se traduzem os limites de representação do expoente e da mantissa? Não vou fazer contas aqui, vou apenas mostrar os resultados práticos. Descreverei primeiro que resultados são esses:

- Devido aos limites de expoentes, existe um número máximo e um número mínimo que podem ser representados com uma dada precisão. O número máximo corresponde ao maior expoente possível com a maior mantissa possível; o número mínimo, analogamente, corresponde ao menor expoente possível com a menor mantissa possível. Na tabela, são apresentadas as faixas aproximadas.

  Os mesmos limites se aplicam para os módulos dos números negativos, já que números negativos e positivos são tratados de maneira simétrica na representação de ponto flutuante.

- Como vários números precisam ser arredondados para serem representados, a representação tem um certo número de algarismos significativos de precisão — quantas casas decimais estão corretas na representação de um número. Como os números não são representados em base 10, a quantidade de casas corretas pode variar de número para número; os valores apresentados na tabela seguir correspondem à quantidade mínima.

Tabela 4.3: Comparação dos diferentes tipos de número de ponto flutuante em C

Tipo	Expoente	Mantissa	Intervalo	Precisão
float (32 bits)	8 bits	23 bits	$10^{-45} \sim 10^{38}$	6 decimais
double (64 bits)	11 bits	52 bits	$10^{-324} \sim 10^{308}$	15 decimais
long double (80 bits)	15 bits	64 bits	$10^{-4950} \sim 10^{4932}$	19 decimais

Nota: A representação de ponto flutuante pode não ser a mesma em todos os computadores. As informações aqui descritas estão de acordo com a representação conhecida como *IEEE 754*, que é a mais usada nos computadores pessoais hoje em dia.

O tipo long double ainda não tinha sido mencionado. Ele é um tipo que, em geral, tem precisão maior que o double (embora certos compiladores não sigam essa norma), mas seu tamanho não é tão padronizado como os outros dois tipos (precisão simples e dupla). Além de 80 bits, é possível encontrar tamanhos de 96 ou 128 bits.

De maneira similar ao double, é necessário utilizar o código `%Lf` em vez de apenas `%f` quando queremos ler com scanf uma variável long double. Nesse caso também é necessário utilizar esse mesmo código para o printf (em contraste com o double que continua usando o código `%f`).

4.3.3 Problemas e limitações

Como já vimos, os tipos de ponto flutuante não conseguem representar com exatidão todos os números fracionários. Devido aos arredondamentos, acontecem alguns problemas que, segundo a matemática que conhecemos, são inesperados. Contas que, matematicamente, dão o mesmo resultado, podem ter resultados diferentes. Até uma simples troca de ordem das operações pode alterar o resultado.

Vamos ilustrar isso com um exemplo. Pediremos ao usuário que digite os coeficientes reais de uma equação quadrática

$$ax^2 + bx + c = 0,$$

e analisaremos os três tipos de solução que podem ocorrer dependendo do valor de $\Delta = b^2 - 4ac$ (ainda não vimos como calcular raiz quadrada, então vamos apenas analisar os casos):

- Se $\Delta > 0$, a equação tem duas raízes reais distintas.
- Se $\Delta = 0$, a equação tem uma raiz real dupla.
- Se $\Delta < 0$, a equação não tem raízes reais.

O programa ficaria assim:

```
#include <stdio.h>

int main()
{
   float a, b, c, delta;
   printf("Dê os coeficientes da equação ax^2 + bx + c = 0: ");
   scanf("%f %f %f", &a, &b, &c);
```

```
    delta = b*b - 4*a*c;
    printf("Delta = %g\n", delta);

    if (delta > 0)
       printf("A equação tem duas raízes reais distintas.\n");
    else if (delta == 0)
       printf("A equação tem uma raiz real dupla.\n");
    else
       printf("A equação não tem raízes reais.\n");

    return 0;
}
```

Esse programa, apesar de estar logicamente correto, apresenta um problema. Considere as equações que têm uma raiz dupla λ, ou seja, equações do tipo $a(x-\lambda)^2$: para elas, Δ é exatamente igual a 0. No entanto, se você testar o programa para várias dessas equações, descobrirá um comportamento estranho: para muitos valores de λ, o valor de Δ calculado pelo programa não é igual a zero. Por exemplo, para valores não inteiros e próximos de 1, obtemos um Δ da ordem de 10^{-7}.

Por que isso acontece? O problema são os arredondamentos que já vimos serem necessários para representar a maioria dos números; por conta deles, nossa conta não dá exatamente zero no final.

Neste momento, considere isso como um alerta — evite fazer comparações de igualdade com números de ponto flutuante; um pouco mais adiante, veremos o que é possível fazer para contornar (ou conviver melhor com) essas limitações.

4.3.4 Ponto flutuante *vs.* inteiros

As operações entre números de ponto flutuante funcionam basicamente da mesma maneira que operações entre inteiros; você pode até realizar operações "mistas" entre números inteiros e de ponto flutuante — o inteiro é convertido automaticamente para ponto flutuante e a operação é realizada como se os dois números fossem de ponto flutuante. Em geral, sempre que você fornece um inteiro num lugar onde era esperado um número de ponto flutuante, o inteiro é convertido para ponto flutuante.

Por outro lado, a coerência matemática da linguagem exige que o resultado de uma operação entre dois inteiros seja um inteiro. Agora lembremos que a divisão entre dois números inteiros nem sempre é exata, e daí surgem os números racionais — a divisão entre dois números inteiros deve ser, em geral, um número racional. Como resolver esse conflito?

Em C, a primeira regra (fechamento das operações entre inteiros) é a que prevalece: quando dividimos dois inteiros, obtemos apenas o quociente da divisão inteira entre eles. Para obter o resultado racional da divisão entre dois inteiros, precisamos converter um deles em ponto flutuante.

Suponha que você deseja imprimir a expansão decimal da fração $1/7$. Uma possível primeira tentativa seria a seguinte:

```
    float x;
    x = 1/7;
    printf("1/7 = %f\n", x);
```

Para sua frustração, você obteria o número `0.000000` como resultado. Claro, se você fez uma divisão entre inteiros, o resultado foi a divisão inteira entre eles; ao guardar o resultado numa variável de ponto flutuante, apenas o resultado é convertido — quem decide que tipo de operação será realizada são os operandos em si.

Para fazer o que queríamos, pelo menos um dos operandos deverá ser de ponto flutuante. Ou seja, devemos escolher uma das alternativas:

```
x = 1.0 / 7;
x = 1 / 7.0;
x = 1.0 / 7.0;
```

Mas e se quiséssemos calcular a divisão entre dois números que não conhecemos *a priori*? Devemos usar a

4.3.5 Conversão explícita de tipos (*casting*)

Muitas conversões de tipo são feitas automaticamente em C, mas em alguns casos, como na motivação que antecede esta seção, é preciso fazer uma conversão "forçada". Esse tipo de *conversão explícita* também é conhecido pelo nome técnico (em inglês) de *casting*. Ela é feita da seguinte maneira:

(novo_tipo) variável_ou_valor

Por exemplo, para calcular a razão (fracionária) entre dois inteiros fornecidos pelo usuário, poderíamos converter um deles para o tipo float e realizar a divisão:

```
int a, b;
float x;
/* (...) leitura dos números */
x = (float)a / b;
printf("%f\n", x);
```

Note, porém, que o operador de conversão de tipos só atua sobre a variável que vem imediatamente depois dele — por exemplo, se quiséssemos converter uma expressão inteira, precisaríamos de parênteses:

```
x = (float)(a / b) / c;
```

Nesse caso, seria realizada a divisão inteira entre *a* e *b*, e depois seu quociente seria dividido por *c*, numa divisão racional.

4.4 Funções matemáticas

Em diversas áreas, as quatro operações básicas não cobrem todas as contas que precisam ser feitas em um programa. É muito comum, mesmo que não estejamos lidando diretamente com matemática, precisar das funções trigonométricas, raiz quadrada, exponencial, entre outras. Todas essas funções de uso comum constam no padrão da linguagem C e fazem parte da biblioteca matemática padrão. Para usá-las em um programa, precisamos de mais uma instrução no começo do arquivo de código:

```
#include <math.h>
```

Ao compilar com o GCC, também é necessário incluir a biblioteca matemática na linha de comando da compilação; isso é feito com a opção `-lm`. (Na versão do MinGW para Windows, devido à diferença de organização do sistema, isso não é necessário.)

Uma lista completa das funções disponíveis nesse arquivo pode ser facilmente encontrada nas referências sobre a biblioteca padrão; na tabela 4.4 são listadas apenas as principais. A maioria das funções aqui listadas recebe apenas um parâmetro, exatamente como esperado; casos excepcionais serão indicados.

Tabela 4.4: Principais funções matemáticas da biblioteca padrão

Função	Significado
`sin`, `cos`, `tan`	Funções trigonométricas: seno, cosseno e tangente. Os ângulos são sempre expressos em radianos.
`asin`, `acos`, `atan`	Funções trigonométricas inversas. `asin` e `atan` devolvem um ângulo no intervalo $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$; `acos` devolve um ângulo no intervalo $[0, \pi]$.
`sinh`, `cosh`, `tanh`	Funções hiperbólicas (seno, cosseno e tangente)
`sqrt`	Raiz quadrada (*square root*)
`exp`	Função exponencial (e^x)
`log`	Logaritmo natural, base e (ln)
`log10`	Logaritmo na base 10
`abs`, `fabs`	Módulo (valor absoluto) de um número. Use `abs` para inteiros e `fabs` para números de ponto flutuante.
`pow(x, y)`	Potenciação: x^y (x e y podem ser números de ponto flutuante)

4.4.1 Cálculo de séries

Várias das funções implementadas na biblioteca matemática podem ser calculadas por meio de expansões em série (somatórias infinitas); por exemplo, a função exponencial:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

É claro que, em um computador, é impossível calcular todos os termos dessa série para chegar ao resultado; após um certo ponto, devido à convergência das séries, o valor de cada termo é muito pequeno, de modo que, frente à precisão do computador, somá-los não faz mais diferença. Dessa maneira, devemos somar um número finito N de termos dessa série. Mas como escolher N? Seguem alguns dos critérios possíveis:

- Escolher um N fixo. Esse critério não é muito bom, porque a precisão até o N-ésimo termo costuma depender do argumento x; em vários casos, para valores grandes de x é necessário somar um número maior de parcelas, enquanto, para x bem pequeno, poucas parcelas já dão uma aproximação muito boa.
- Limitar a magnitude das parcelas a serem somadas — interromper a soma quando o módulo da parcela for menor que um dado ε (por exemplo, $\varepsilon = 10^{-9}$). Esse critério é melhor que o anterior, mas não leva em conta a magnitude do resultado; por exemplo, se o valor da função for da ordem de 10^9, uma parcela de 10^{-9} realmente não faz

diferença; se o valor da função for da ordem de 10^{-6}, a mesma parcela de 10^{-9} ainda é significativa.

- Podemos modificar um pouco esse critério levando isso em conta: em vez de limitar o tamanho de cada parcela, limitaremos o tamanho da parcela *dividido pelo valor acumulado até então*. Se dividirmos pela soma do passo anterior, que não inclui essa parcela, obtemos a quantidade conhecida como *variação relativa* — se S_n indica a soma das primeiras n parcelas, a $(n+1)$-ésima parcela corresponde a $S_{n+1} - S_n$, e então o critério de limitação na variação relativa traduz-se matematicamente em

  $$\left| \frac{S_{n+1} - S_n}{S_n} \right| < \varepsilon,$$

  parando no passo $n+1$ se essa condição for satisfeita. Esse critério é bastante usado em diversos métodos numéricos.

- Devido à precisão do computador, em algum momento as parcelas ficarão tão pequenas que absolutamente não irão mais alterar a soma — por exemplo, se estamos trabalhando com 5 algarismos significativos, somar 0,0001 no número 100 não faz a menor diferença. Assim, podemos comparar a soma de cada passo com a soma anterior, e interromper o processo quando as somas forem iguais.

  No entanto, é necessário tomar cuidado com possíveis otimizações do compilador — como você somou uma parcela não-nula, logicamente a soma deve ter-se alterado; portanto, com certas opções de otimização ativadas, o compilador "adianta" a resposta da comparação entre as duas somas e diz que elas são diferentes, sem verificar se a variação realizada estava realmente dentro da precisão do ponto flutuante. O resultado disso é um laço infinito.

4.4.2 Definindo constantes

A linguagem C permite que você crie "apelidos" para constantes que você usa no seu programa. Isso ajuda muito a manter o código organizado, pois evita que o código fique cheio de números "anônimos" difíceis de entender. Além disso, quando utilizamos uma constante pelo nome, ganhamos uma grande flexibilidade: se precisamos alterar o valor dela, podemos alterar simplesmente a sua definição, e a alteração refletir-se-á em todos os lugares onde a constante foi utilizada.

Para definir constantes, utilizamos o comando `#define`, da seguinte maneira:

#define *CONSTANTE valor*

As restrições sobre os nomes das constantes são as mesmas que para nomes de variáveis e funções. Para utilizar essas constantes,

Um exemplo de aplicação disso: suponha que você criou um programa que mostra o cardápio de uma pizzaria, recebe um pedido e calcula o valor total. Agora a pizzaria quer que você refaça o programa pois os preços foram reajustados. Se você tiver escrito os preços diretamente no meio do código do programa, você terá bastante trabalho procurando os preços e modificando em diversos lugares. No entanto, se você definir constantes como `PRECO_MARGHERITA` e `PRECO_PORTUGUESA` e usá-las no código, você só precisará alterar a definição das constantes.

```c
#include <stdio.h>

#define PRECO_MUSSARELA  15.00
#define PRECO_MARGHERITA 16.00
#define PRECO_PORTUGUESA 18.50

void imprime_cardapio()
{
  printf (
    "Mussarela      %5.2f\n"
    "Margherita     %5.2f\n"
    "Portuguesa     %5.2f\n",
    PRECO_MUSSARELA, PRECO_MARGHERITA, PRECO_PORTUGUESA);
}

/* Deixarei como exercício para você pensar a parte de registrar um
 * pedido.  Na verdade, por enquanto você pode se preocupar apenas em
 * calcular o preço total.
 * Sugestão: use um terminador para poder saber quando o pedido acaba.
 */
```

4.5 O tipo char

Você já conheceu três dos quatro tipos fundamentais da linguagem C. Falta apenas um, o tipo **char** — que já mencionamos, mas não tivemos a oportunidade de usar. Como o nome sugere, ele é designado para guardar caracteres (como `a Y ^ { @ # %`). No entanto, ele é apenas mais um tipo inteiro, com um intervalo de valores permitidos mais modesto que o de um inteiro comum. Isso ocorre assim porque, para um computador, um caractere é apenas um número; caracteres são codificados como números de acordo com uma tabela de correspondência — por exemplo, a letra `A` maiúscula é armazenada como 65, o cifrão `$` como 36, etc. Esses exemplos são os códigos da conhecida tabela ASCII, que é a base do padrão atual para a codificação de caracteres.

Uma variável do tipo `char` pode conter um caractere (apenas um!), e caracteres são representados entre aspas *simples* (não confunda com as aspas duplas como as que você usa nas funções `printf` e `scanf`). Alguns exemplos: `'a'`, `'2'`, `'@'`. Um exemplo de variável do tipo `char` seria:

```c
char opcao;
opcao = 'b';
```

Há alguns caracteres que não são representados literalmente, mas por meio de códigos especiais. Um deles já lhe foi apresentado, a saber, o caractere de quebra de linha, representado pela sequência `'\n'`. Alguns outros exemplos são apresentados na tabela 4.5, junto com os códigos numéricos correspondentes na tabela ASCII.

Tabela 4.5: Algumas das sequências utilizadas para representar caracteres especiais.

Sequência	ASCII	Significado
`\t`	9	Tabulação. Avança o cursor para posições pré-definidas ao longo da linha; usualmente, essas posições são definidas de 8 em 8 caracteres a partir da primeira posição da linha.
`\n`	10	Quebra de linha. Esse caractere é comumente conhecido pela sigla NL, *new line*, ou por LF, *line feed*.
`\r`	13	"Retorno de carro" (CR, *carriage return*): retorna o cursor para o início da margem esquerda. (Ver observação adiante.)
`\"`	34	Aspa dupla.
`\'`	39	Aspa simples.
`\\`	92	Barra invertida.
`\`*nnn*		Permite especificar qualquer caractere pelo seu código em base octal (de 000 a 377). Cada *n* é um dígito de 0 a 7.
`\x`*nn*		Idem, em base hexadecimal (de 00 a FF). Cada *n* é um dígito de 0 a 9 ou uma letra de A a F (maiúscula ou minúscula).

Os caracteres CR e LF (além de outros que não foram indicados aqui) são uma herança da época das máquinas de escrever. Para começar a escrever texto na linha seguinte, eram necessárias duas ações: retroceder o carro de impressão para o começo da margem (CR) e alimentar mais uma linha de papel (LF). Em sistemas Windows, o final de uma linha de texto é usualmente indicado pela sequência CR + LF, ao invés de simplesmente LF, como é de praxe nos sistemas Unix (enquadram-se aqui o Linux e o Mac OS X). Nos sistemas Mac antigos usava-se apenas CR, sem LF.

Veja que os caracteres `" ' \` precisam de uma representação alternativa: as duas aspas porque podemos querer representá-las como caracteres sem que sejam interpretadas como o final da sequência de caracteres; a barra invertida porque ela própria é usada em outras sequências de caracteres especiais.

4.5.1 Entrada e saída

Para ler e imprimir variáveis do tipo char, você pode usar o código de formato `%c` nas funções printf e scanf. Por exemplo:

```
char opcao;
scanf("%c", &opcao);
printf("Você escolheu a opção (%c)!\n", opcao);
```

```
int opcao;
opcao = getchar();
putchar('a');
```

É importante notar que, tanto com a função scanf quanto com a getchar, os caracteres digitados não são recebidos pelo programa em tempo real. Cada vez que você digita um caractere, ele é armazenado temporariamente numa região da memória chamada *buffer de teclado*, e só após o final da linha é que o conteúdo do buffer é liberado para o nosso programa, através dessas funções.

Ponteiros e vetores 5

5.1 Prolegômenos

Num computador, cada variável é guardada em uma certa posição da memória. Essas posições de memória são numeradas, de modo que cada uma tem um *endereço* numérico — é como se cada uma fosse uma casa em uma rua.

Em C, um **ponteiro** (também chamado de *apontador*) é uma variável que guarda uma referência a outra variável — seu valor é o endereço de uma outra variável. Se o valor de um ponteiro p é o endereço de uma variável X, então dizemos que *p aponta para X*. Veja uma ilustração disso na figura 5.1. Se um ponteiro aponta para uma variável, você pode acessar essa variável (ler ou alterar seu valor) através do ponteiro — logo veremos como.

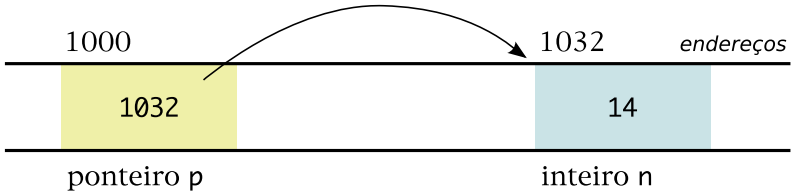


Figura 5.1: Esquema do funcionamento de um ponteiro. O ponteiro p contém o endereço da variável n (1032), ou seja, p aponta para n.

Isso pode parecer apenas uma maneira de complicar as coisas, mas na realidade tem diversas utilidades, das quais citamos algumas:

- Quando precisamos transmitir uma grande quantidade de dados a outra parte do programa, podemos passar apenas um ponteiro para esses dados em vez de fazer uma cópia dos dados e transmitir a cópia. Isso economiza tempo — o processo de duplicar os dados gasta tempo de processamento — e, obviamente, espaço na memória.

- Uma função em C só pode devolver um valor com a instrução `return`. No entanto, se uma função recebe como parâmetros ponteiros para outras variáveis, você poderá gravar valores nessas variáveis, e com isso uma função pode gerar vários valores de saída.

- O conceito de ponteiros pode ser estendido para *funções* — é possível passar um ponteiro para uma *função* como parâmetro. Por exemplo, podemos criar uma função chamada `acha_raiz` com um algoritmo numérico que acha raízes de uma função matemática `f`; a função `f` poderia ser passada (na forma de ponteiro) como parâmetro da função `acha_raiz`.

5.1.1 Declaração de ponteiros

Em C, para declarar uma variável que funciona como ponteiro, colocamos um *asterisco* (`*`) antes do seu nome. Um ponteiro só pode apontar para um tipo de variável, já que a maneira de armazenar o valor de uma variável na memória depende do seu tipo. Por exemplo, um ponteiro que aponta para uma variável inteira não pode ser usado para apontar para uma variável de ponto flutuante. Assim, um ponteiro também precisa de um tipo, que deve ser igual ao tipo de variável para a qual ele irá apontar.

Por exemplo, se queremos criar um ponteiro *p* que irá apontar para uma variável inteira, declaramo-no da seguinte maneira:

```
int *p;
```

Essa instrução apenas declara um ponteiro, sem apontá-lo para nenhuma variável.

Se quisermos declarar vários ponteiros com uma única instrução, devemos colocar o asterisco *em cada um deles*. Se você escrever o asterisco apenas no primeiro nome, apenas a primeira variável será um ponteiro!

```
int *p1, *p2, *p3;    /* ok, os três são ponteiros */
double *p4, p5, p6;   /* problema! só p4 será um ponteiro */
```

Para fazer um ponteiro apontar para uma variável, devemos atribuir-lhe como valor o endereço de outra variável, e não um "número comum". Para isso, usamos o operador `&` (E comercial), que fornece o endereço de uma variável — aqui ele será conhecido como o operador *endereço-de*. Se temos uma variável *var*, seu endereço é representado por `&var`. Por exemplo:

```
int n;
int *p;
p = &n;
```

Aqui criamos uma variável inteira chamada *n*, e em seguida criamos um ponteiro *p*, que é apontado para a variável *n*.

5.1.2 Acesso indireto por ponteiros

Para acessar a variável que é apontada por um ponteiro, usamos o operador `*` (o mesmo asterisco usado na declaração), chamado *operador de indireção* ou *operador de desreferenciação*.[8] Esse operador faz a "volta" do processo que leva da variável ao ponteiro (a *referenciação* da variável); por isso o chamamos de operador de desreferenciação. O nome "indireção" é usado simplesmente porque isso é um acesso indireto à variável.

[8] Esse é um termo difícil de se traduzir. Em inglês, diz-se *dereference*, que seria uma combinação do prefixo *de-* (equivalente, nesse caso, ao nosso des-) e da palavra que significa "referência", no sentido de "desfazer uma referência". Muitas pessoas tentam traduzir isso como *de-referência*, mas essa palavra não consta nos dicionários. (Tudo bem, "desreferenciação" também não, mas eu achei melhor.)

Nomenclaturas à parte, se `p` é um ponteiro, podemos acessar a variável para a qual ele aponta com `*p`. Essa expressão pode ser usada tanto para ler o conteúdo da variável quando para alterá-lo.

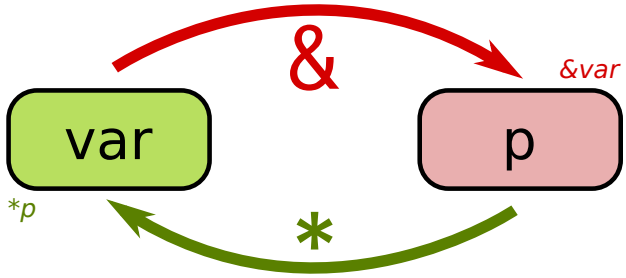


Figura 5.2: Ilustração da relação entre ponteiros e variáveis.

Por exemplo,

```
int n, *p;                /* veja que podemos declarar os dois */
p = &n;                   /* num mesmo comando */

*p = 5;
printf("n = %d\n", n);    /* imprime 5  */

n = 10;
printf("*p = %d\n", *p);  /* imprime 10 */
```

Vemos, então, que acessar um ponteiro para uma variável é, de certa forma, equivalente a acessar a variável apontada. Podemos também mudar a variável para a qual um ponteiro aponta, e a partir daí as operações com o ponteiro só afetarão a variável nova:

```
int a, b, *p;

p = &a;
*p = 5;
printf("a = %d\n", a);    /* imprime 5  */

p = &b;
*p = 10;
printf("a = %d\n", a);    /* ainda imprime 5  */
printf("b = %d\n", b);    /* imprime 10 */
```

Desreferenciação ou multiplicação? O leitor atento deve ter-se perguntado se o fato de o asterisco dos ponteiros ser o mesmo asterisco da multiplicação não gera nenhum problema. E de fato não há nenhum problema; podemos inclusive multiplicar os valores apontados por dois ponteiros, por exemplo:

```
c = *p1 * *p2;
```

Esse código teria o efeito de obter os valores apontados por *p1* e *p2*, multiplicá-los e guardar o resultado em *c*, como poderíamos esperar.

Isso ocorre porque os operadores de desreferenciação são sempre interpretados antes dos de multiplicação. Após a interpretação das expressões `*p1` e `*p2`, o que sobra são os dois valores apontados pelos ponteiros, com um asterisco entre eles; isso só pode significar uma multiplicação — se o asterisco do meio fosse atuar como operador de desreferenciação, a expressão ficaria inválida, além de ele estar agindo sobre algo que já não seria um ponteiro!

Apesar do código mostrado acima ser válido e inambíguo (para um computador), é recomendável que você use parênteses quando tiver de fazer esse tipo de coisa — isso deixa o código bem mais legível:

```
c = (*p1) * (*p2);
```

5.2 Ponteiros como parâmetros de funções

Uma grande utilidade dos ponteiros é a possibilidade de se modificar as variáveis que foram passadas como parâmetros para uma função. Anteriormente vimos um exemplo de uma função que tenta (sem sucesso) trocar o valor de duas variáveis; sem usar ponteiros, a tarefa era impossível, pois os valores dos parâmetros de uma função são sempre *copiados*, de modo que ela não tem acesso às variáveis originais.

Agora que conhecemos ponteiros, a tarefa fica fácil. Se queremos que uma função modifique uma variável, basta passar a ela um ponteiro para a variável, em vez de passar o valor da variável (que é o comportamento padrão). Para de fato modificar a variável dentro da função, devemos usar o operador de indireção para trocar os valores apontados pelos ponteiros (senão, mudaríamos apenas o lugar para o qual o ponteiro aponta, o que não nos interessa).

Vamos começar com um exemplo bem simples, que apenas dobra o valor de uma variável. No cabeçalho da função, o ponteiro é especificado da mesma maneira que nas declarações de variável — com um asterisco entre o tipo e o nome do parâmetro.

```
void dobra_variavel(int *var)
{
   *var = (*var) * 2;
}
```

Para chamar essa função, usamos o operador `&` para passar o endereço de uma variável. Veja que você não pode passar um número (uma constante) diretamente para essa função, pois ele não tem um endereço! Outro detalhe que deve ser levado em conta é que a variável apontada deve ser inicializada antes de chamarmos a função, já que a função se baseia no valor que havia na variável.

```
int main()
{
   int num;

   num = 10;
   dobra_variavel(&num);
   printf("%d\n", num);    /* 20 */

   return 0;
```

```
}
```

Com esse exemplo em mente, não devemos ter dificuldades para montar uma função que de fato troca o valor de duas variáveis (essa função, na prática, não é incrivelmente útil, mas é boa para ilustrar esse conceito):

```
void troca(int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

Naturalmente, também será necessário alterar a chamada à função troca, já que agora precisamos passar os *endereços* das variáveis e não mais os seus valores. Para chamá-la, escreveremos algo como o seguinte:

```
   int m, n;
   m = 10;
   n = 20;
   troca(&m, &n);
```

Em computação, é comum usar os nomes *chamada por valor* e *chamada por referência*; eles indicam se a função chamada recebe apenas uma cópia do *valor* da variável, ou se recebe uma *referência* (um ponteiro) para a variável. Em C, as chamadas de funções são intrinsecamente por valor, mas podemos usar ponteiros para obter o comportamento das chamadas por referência.

5.2.1 Usando ponteiros para devolver valores

Em muitos casos, uma função pode ter mais de uma saída. Como uma função só pode devolver um único valor através da instrução return, estamos um pouco limitados. No entanto, se passarmos a uma função o *endereço* de uma variável, a função pode gravar diretamente sua saída naquela variável.

Vamos rever o exemplo da equação de segundo grau $ax^2 + bx + c = 0$, com coeficientes reais. Sabemos que, de acordo com o sinal do discriminante $\Delta = b^2 - 4ac$, pode haver duas, uma ou nenhuma raiz real. Vamos escrever uma função que distingue entre esses três casos, como já fizemos na página 56, e calcula as raízes reais da equação quando for o caso. A função deverá devolver o número de raízes reais (distintas), e gravar as raízes encontradas (quando for o caso) em variáveis fornecidas pelo usuário. Veja nossa solução:

```
int equacao_quadratica (double a, double b, double c, double *raiz1,
                        double *raiz2)
{
   double delta;

   delta = b*b - 4*a*c;
   if (delta < 0) {
      return 0;
   }
```

```
    else if (delta == 0) {
        *raiz1 = -b/(2*a);
        return 1;
    }
    else {
        *raiz1 = (-b - sqrt(delta)) / (2*a);
        *raiz2 = (-b + sqrt(delta)) / (2*a);
        return 2;
    }
}
```

5.3 Cuidado com os ponteiros!

Um grande poder exige uma grande responsabilidade. Os ponteiros são uma ferramenta muito poderosa da linguagem C, mas devem ser usados com muita cautela. Se um ponteiro não estiver apontando para o lugar que você espera, coisas terríveis podem ocorrer. É comum que apareçam bugs ou erros esquisitos no seu programa simplesmente porque você está acessando um ponteiro inválido. Dizemos que um ponteiro é *inválido*, em geral, quando ele aponta para uma posição de memória que não é uma variável do seu programa. Isso costuma ocorrer em situações como as seguintes:

- Você não inicializou o ponteiro (isto é, não atribuiu nenhum valor a ele), de modo que não se tem idéia do lugar da memória para o qual ele aponta.
- Você apontou o ponteiro para um endereço "arbitrário", que não pertence ao seu programa. Por exemplo, se você tentar atribuir o valor 300 ao seu ponteiro, ele apontará para a posição 300 da memória do seu computador — não temos idéia do que pode haver nesse lugar.
- Seu ponteiro "caiu" num lugar inválido da memória, apesar de ter sido inicializado corretamente. Isso ocorre mais frequentemente quando você está variando através de um laço (for, while) a posição apontada pelo ponteiro (veremos isso mais adiante) e não pára na hora certa — alguma hora, seu ponteiro acaba apontando para um lugar que não deveria.

Por exemplo, você **não** deve fazer nada parecido com isso:

```
int *p;
*p = 5;      // erro terrível!!
```

O que ocorreu aqui é que você criou um ponteiro mas não definiu para onde ele aponta. Seu valor inicial será aleatório, e portanto, ao tentar atribuir um valor à variável apontada pelo ponteiro, você estará acessando uma posição aleatória da memória. Repito: *não faça isso!*

Geralmente, quando você tentar acessar ponteiros inválidos, o programa irá dar um erro durante a execução e fechar. No Linux, você verá algo do tipo "Segmentation fault" ou "Falha de segmentação" (isso quer dizer que você acessou um *segmento* de memória inválido; tem a ver com a organização interna da memória). No Windows, você deverá ver uma das famosas e genéricas mensagens "Este programa executou uma operação ilegal e será fechado".

Às vezes também pode acontecer de o programa continuar funcionando mesmo com um erro desses; os sintomas serão mais sutis, como valores inesperados nas suas variáveis, e é aí que esses erros são mais difíceis de rastrear. Portanto, olho vivo ao usar ponteiros!

5.4 Vetores

Vetores são uma maneira prática de guardar um grande conjunto de variáveis relacionadas — por exemplo, sequências de números. Em C, um vetor é uma série de variáveis *indexadas* — que podem ser acessadas por meio de um índice inteiro, por exemplo `vetor[4]`. Há uma pequena restrição: um vetor só guarda variáveis do mesmo tipo — ou seja, você pode fazer um vetor de inteiros e um vetor de números de ponto flutuante, mas não um vetor que contenha ambos.

> Há vários termos diferentes para se referir aos vetores. É muito comum encontrar **array**, que é o termo original em inglês; também vemos **matrizes**, termo que prefiro reservar apenas aos "vetores multidimensionais", como veremos mais tarde. O termo **lista** também é possível, mas pouco usado em C (ele pode ser usado em outro contexto semelhante, sendo mais comum em outras linguagens).

Essas variáveis são todas guardadas sequencialmente (sem buracos) na memória e, em um vetor de n elementos, são identificadas por índices de 0 a $n-1$ (veja a figura 5.3). Em C, podemos nos referir ao elemento de índice i de um vetor V pela expressão `V[i]`.

Figura 5.3: Ilustração do modo de armazenamento de um vetor com n elementos.

Para usar um vetor, precisamos primeiro declará-lo, como era feito para qualquer variável normal. A declaração de um vetor é feita da seguinte maneira:

tipo_de_dados nome_vetor[*tamanho*];

O compilador entende essa "frase" como: *reserve na memória um espaço para* `tamanho` *variáveis do tipo* `tipo_de_dados`*, e chame esse espaço de* `nome_vetor`. Veja dois exemplos desse tipo de declaração:

```
int sequencia[40];
double notas[100];
```

É importante ressaltar que o compilador apenas *reserva* o espaço de memória pedido, sem colocar nenhum valor especial nele. Isso significa que o vetor conterá inicialmente uma seleção "aleatória" de valores (que sobraram da execução de algum programa que usou aquele espaço), exatamente como ocorria para as variáveis comuns.

Você deve prestar atenção a alguns detalhes do funcionamento dos vetores em C:

- Os elementos de um vetor são numerados a partir de zero.[9] Dessa maneira, num vetor V que tem 5 elementos, os elementos são: $V[0]$, $V[1]$, $V[2]$, $V[3]$ e $V[4]$. Não se

[9] Isso não é apenas uma extravagância do C; muitas linguagens funcionam dessa maneira, que é a mais natural para um computador — logo mais veremos por quê.

confunda com a declaração! Um tal vetor seria declarado com uma instrução do tipo `int V[5]`, mas o elemento `V[5]` não existiria!

É realmente necessário tomar bastante cuidado com a numeração dos elementos. Se você tentar acessar um elemento "inválido", como $V[5]$ ou $V[100]$ (para este caso com 5 elementos), o compilador não o avisará disso, e erros estranhos começarão a ocorrer no seu programa — o mesmo tipo de erro que pode ocorrer com os ponteiros.

- O tamanho deve ser um valor **constante** (não pode depender de valores de variáveis). Ou seja, você não pode perguntar ao usuário o tamanho desejado, guardar numa variável `n` e depois declarar um vetor do tipo `int V[n]`. Em outras palavras, o tamanho do vetor deve ser um valor que possa ser estabelecido *na hora da compilação do programa*.

  Por isso, quando for necessário ler uma quantidade de dados que só é estipulada na execução do programa, a princípio teremos de estabelecer um teto no número de dados a serem lidos, usando um vetor de tamanho fixo. Caso o teto não seja atingido, algumas entradas do vetor ficarão sem ser utilizadas.

  É possível, sim, criar vetores cujo tamanho só é conhecido *a posteriori*; no padrão C99 é possível fazer declarações do tipo `int V[n]` com algumas restrições. Há outro recurso, de certa maneira mais flexível, que permite criar vetores cujo tamanho só é conhecido na execução do programa: a *alocação dinâmica de memória*, que será vista no Capítulo 7.

- Além de constante, o tamanho dos vetores é **imutável**, ou seja, se eu declarei um vetor de 5 entradas, eu não posso aumentá-lo para que caibam 10 entradas. Se eu quero que caibam 10 entradas, eu preciso reservar espaço para 10 entradas logo no começo.

  (Novamente, a alocação dinâmica de memória salva a pátria nesse aspecto; no capítulo 7, você verá o que é possível fazer quanto a isso.)

Podemos acessar os elementos de um vetor, em geral, tratando-os como se fossem variáveis normais. O operador de endereço também pode ser usado com os elementos individuais do vetor. Por exemplo,

```
int lista[3];

scanf("%d", &lista[0]);
lista[1] = 37;
lista[2] = lista[1] - 3*lista[0];

printf("%d %d %d\n", lista[0], lista[1], lista[2]);
```

Dito isso, vamos ver algumas aplicações simples do uso de vetores.

Exemplo 5.1. Vamos fazer um programa que lê uma lista de n números ($n \leq 100$) e os imprime na ordem inversa (em relação à ordem em que foram lidos). Para isso, é necessário armazenar cada um dos números lidos antes de começar a imprimi-los — isso não é possível (a não ser com um código extremamente pedestre e propenso a erros) com o que tínhamos aprendido antes.

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

```c
int main()
{
   int lista[100];
   int i, n;

   printf("Digite o número de elementos (no máximo 100): ");
   scanf("%d", &n);
   if (n > 100) {
      printf("n não pode ser maior que 100! Só lerei os "
             "100 primeiros números.\n");
      n = 100;
   }

   printf("Digite a seguir os %d elementos:\n", n);
   /* leitura dos elementos */
   for (i = 0; i < n; i++)
      scanf("%d", &lista[i]);

   /* impressão dos elementos, na ordem inversa */
   for (i = n-1; i >= 0; i--)
      printf("%d ", lista[i]);
   printf("\n");

   return 0;
}
```

Um aspecto que ainda não reforçamos foi a validação da entrada do usuário. Você pode pedir encarecidamente que o usuário digite um número até 100, mas nada garante que o usuário não será desonesto ou distraído e digite um número fora dos limites pedidos. Nessas horas, você *não deve confiar no usuário*, e **deve** verificar se o valor digitado é válido, para evitar que aconteçam coisas más em seu programa — nesse caso, se o usuário digitasse um número maior que 100 e não fizéssemos essa verificação, acabaríamos acessando posições inválidas do vetor (acima do índice 99).

5.4.1 Inicialização de vetores

Em algumas situações você precisará usar vetores cujo conteúdo seja determinado inicialmente por você, e não lido do teclado ou de algum arquivo. Obviamente seria muito cansativo ter de inicializar elemento por elemento, como a seguir:

```c
int lista[100];

lista[0] = 9;
lista[1] = 35;
⋮
lista[99] = -1;
```

Felizmente, o C permite que você inicialize os valores de um vetor junto com a declaração (assim como de uma variável escalar comum). Isso é feito da seguinte maneira:

tipo_de_dados nome_vetor[*tamanho*] = { *lista de valores* };

Na realidade, você não precisa escrever o tamanho explicitamente se especificar todos os elementos — o compilador simplesmente contará quantos elementos você digitou (é necessário, no entanto, manter os colchetes, para que o compilador saiba que isso é um vetor). Assim, o nosso árduo exemplo poderia ser escrito da seguinte maneira (vamos reduzir um pouco o tamanho do vetor, só para não precisarmos digitar tantos elementos):

```
int lista[] = {9, 35, -17, 9, -4, 29, -2, 10, -1};
```

Note que isso só é possível na hora da declaração. Não é possível, fora desse contexto, definir de uma só vez todos os elementos do vetor, como a seguir (é necessário atribuir elemento por elemento):

```
/* isto está errado! */
lista = {7, 42, 0, -1, 3, 110, 57, -43, -11};
```

Também é possível declarar um vetor de um certo tamanho mas só inicializar parte dele (desde que seja a parte do começo). Por exemplo, suponha que queremos espaço para uma sequência de 100 inteiros, mas que vamos começar inicializando apenas os 10 primeiros. Nesse caso, escrevemos explicitamente o tamanho do vetor, mas só especificamos os elementos desejados:

```
int lista[100] = {9, 35, -17, 9, -4, 29, -2, 10, -1, 47};
```

Nesse caso, os elementos que não foram especificados serão automaticamente inicializados com o valor `0`.

5.5 Vetores como argumentos de funções

Não há, a princípio, problema nenhum em passar vetores como argumentos para uma função. O que acontece geralmente é que vetores podem ser bem grandes, e portanto não seria muito prático copiar todos os valores para a função; por isso, vetores são *naturalmente passados por referência* para as funções — ou seja, quando você passa um vetor para uma função, ela na verdade recebe apenas o endereço dos dados; quando sua função for acessar os dados do vetor, ela será apontada diretamente para a posição deles no vetor original. Com isso, qualquer modificação num vetor passado como parâmetro é refletida na função original que passou o vetor.

Uma consequência da passagem por referência dos vetores é que não é possível passar para uma função um vetor "livre", ou seja, uma lista de valores *ad hoc* que não está vinculada a uma variável — na prática, isso quer dizer que não é possível escrever coisas do tipo

```
funcao({2, 3, 4, 5});
```

pois o vetor "livre" não possui um endereço a ser passado para a função.

Vamos primeiro ver na prática como podemos passar vetores como argumentos de funções. Quanto ao cabeçalho da função, basta imitar a declaração de vetores, exceto por um

detalhe: *não devemos fornecer o tamanho do vetor* — os colchetes devem ser deixados 'sozinhos', sem nada no meio.[10] A função a seguir recebe um vetor de inteiros como parâmetro e imprime seu primeiro elemento:

```
void imprime_primeiro(int v[])
{
   printf("%d\n", v[0]);
}
```

Para mandar um vetor como parâmetro de uma função, você simplesmente deve escrever o nome dele, sem colchetes ou qualquer outro símbolo:

```
   int vetor[] = {1, 2, 3, 4};
   imprime_primeiro(vetor);   /* 1 */
```

Agora, se a função não impõe nenhum tamanho para o vetor, como vamos descobrir qual o tamanho do vetor que a função recebeu? A resposta é que a função não tem como descobrir isso sozinha; em geral, é *você* quem deve cuidar disso. Por exemplo, você pode passar para a função *dois* parâmetros: o vetor e o tamanho do vetor. Uma ilustração disso é uma função que imprime todos os elementos de um vetor:

```
void imprime_vetor(int v[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      printf("%d\n", v[i]);
}
```

Agora, para chamá-la, devemos incluir o tamanho do nosso vetor original:

```
   int vetor[] = {1, 2, 3, 4};
   imprime_vetor(vetor, 4);
```

Da mesma maneira, poderíamos fazer uma rotina que lê do teclado uma série de números inteiros e armazena todos eles num vetor. Uma implementação simples seria como a seguir:

```
void le_vetor(int v[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      scanf("%d", &v[i]);
}
```

Para chamá-la, faríamos da mesma maneira que no exemplo anterior:

```
   int vetor[20];
   le_vetor(vetor, 20);
```

[10] Se o tamanho do vetor for explicitamente fornecido, o compilador não deve reclamar, mas o programa não será forçado de maneira alguma a respeitar esse tamanho.

Fornecer o tamanho correto do vetor à função é de suma importância! Se a função achar que seu vetor é maior do que realmente é, ela tentará acessar elementos inválidos (depois do fim do vetor), o que já vimos que é um erro bem grave — especialmente se a função tentar gravar dados nessas áreas!

5.6 Ponteiros e vetores

Vimos que vetores são naturalmente passados por referência como parâmetros de funções. Na verdade, vetores em C têm uma estreita relação com ponteiros. O 'nome' de um vetor funciona, na prática, como um ponteiro para seu primeiro elemento. Isso dá conta de explicar a passagem por referência natural dos vetores: sabendo o endereço do primeiro elemento, podemos encontrar todos os outros elementos do vetor.

Como fazer isso? Como os elementos são guardados juntinhos na memória, sem buracos, basta saber o tamanho T de cada elemento (por isso que é importante que o vetor tenha valores de um tipo só; sabendo o tipo, sabemos o tamanho), por exemplo, em bytes. Daí, partindo da posição do primeiro elemento, fazemos um salto de tamanho T tantas vezes quanto for necessário — para chegar ao elemento de índice n, devemos saltar n vezes. É por isso, também, que os índices começam de 0 e não de 1: para o computador é mais intuitiva a noção de *deslocamento* (em relação ao começo do vetor) do que a noção de *posição*.

Dado um ponteiro para o primeiro elemento de um vetor, é fácil realizar (em C) esse salto: basta *somar* ao ponteiro o número de *elementos* desejado. Note que você não precisa se preocupar em saber o número de bytes, apenas o número de elementos! Por exemplo, se `p` aponta para o começo de um vetor, `p + 1` aponta para o segundo elemento, independentemente de quantos bytes ocupa cada elemento. Da mesma maneira podemos usar os operadores como `++` e `+=` (e, analogamente, os de subtração):

```
int v[10];
int *p;
p = v;   /* faz o ponteiro apontar para o começo do vetor */
         /*  (equivale a:  p = &v[0])  */
p++;     /* aponta o ponteiro para o segundo elemento */
p = p+5; /* faz mais um salto e aponta para o 7º elemento */
```

Essas operações com ponteiros são conhecidas como **aritmética de ponteiros**. Veja que elas apenas alteram o *local* de destino dos ponteiros; elas não mexem com os valores apontados. Note também que não existem, nem fariam sentido, as operações de multiplicação e divisão de ponteiros.

Agora como usamos isso para acessar o i-ésimo elemento do vetor `v`? A expressão `v + i` (ou `p + i` neste exemplo acima) é certamente um ponteiro para ele, pelo que acabamos de ver; então podemos usar `*(v+i)` para acessá-lo. Como isso é usado com muita frequência, existe uma abreviação sintática, que é nada mais que… `v[i]`. Assim, ao acessar elementos de vetores, estamos o tempo todo utilizando a aritmética de ponteiros!

Isso nos dá uma nova maneira de caminhar pelos vetores: criar um ponteiro que aponta inicialmente para o começo do vetor e aumentá-lo de uma unidade num laço `for`, por exemplo:

```
int v[10];
int *p;

for (p = v; /* condição */; p++) {
```

```
    /* o elemento atual do vetor pode ser acessado
     * simplesmente pela expressão *p. */
}
```

Aparentemente há um problema: onde parar o laço se a posição não é controlada por um índice? Veremos em seguida um tipo de vetor para o qual isso não é um problema — é uma solução.

5.7 Strings

Uma das principais utilidades do tipo char é usar vetores para formar *sequências de caracteres*, conhecidas em computação como **strings** (esse termo tornou-se tão difundido que não costuma ser traduzido; possíveis traduções são *sequências* ou *cadeias de caracteres*). Lembre-se de que caracteres nada mais são que inteiros com um significado especial, codificados através de uma tabela (em geral, a tabela ASCII e suas extensões). Em C, uma string é simplesmente um vetor de caracteres, com uma convenção especial: como o valor zero não é utilizado por nenhum caractere, ele é utilizado para marcar o final de uma string, ou seja, ele é o **terminador** da string. O "caractere" de código zero é comumente chamado de *caractere nulo*, e é representado pela sequência especial `'\0'` ou simplesmente pela constante inteira `0`.

Sabendo disso, podemos declarar uma string da mesma maneira como declaramos vetores:

```
char string[] = {'O', 'l', 'á', '\0'};
```

Certamente não é nada prático escrever assim! Felizmente essa notação pode ser bastante abreviada: em vez de escrever explicitamente um vetor de caracteres com um caractere nulo, podemos deixar mais clara a 'cara' de string, colocando os caracteres desejados entre *aspas duplas*, sem separação entre eles, e o resto (inclusive o caractere nulo) ficará por conta do compilador. (Veja que já usamos essa sintaxe o tempo todo quando usamos as funções scanf e printf!) O código acima poderia ser reescrito como a seguir, e as duas formas são absolutamente equivalentes:

```
char string[] = "Olá";
```

Uma das grandes vantagens de usar o terminador `'\0'` (explicita ou implicitamente) é que você não precisa se preocupar com o tamanho das strings: ao caminharmos pelos caracteres de uma string, não precisamos registrar a posição em que estamos para saber quando parar; é só parar quando encontrarmos o caractere nulo. Com isso, em vez de percorrer os caracteres de uma string usando um índice de um vetor, podemos usar apenas um ponteiro. Veja um uso disso neste exemplo, que imprime os caracteres da string indicada, um por linha:

```
char string[] = "Bom dia!";
char *p;

for (p = string; *p != 0; p++)
   printf("Caractere: '%c'\n", *p);
```

A condição de parada do loop é simplesmente o caractere nulo, que não depende de forma alguma do tamanho da string! Podemos usar isso, por exemplo, para achar o tamanho de uma string:

```
int n = 0;
char string[] = "Bom dia!";
char *p;

for (p = string; *p != 0; p++)
   n++;
printf("A string tem comprimento %d\n", n);
```

5.7.1 Alterando strings

Lembre-se de que, se quisermos modificar o valor dos elementos de um vetor comum, é necessário atribuir os valores elemento por elemento; para as strings isso não haveria de ser diferente. Outro fator que complica nossa vida é o fato de vetores terem um tamanho imutável; se declararmos um vetor de um certo tamanho e precisarmos posteriormente de mais espaço, não há, a princípio, o que fazer. Tudo isso é muito relevante pois, se queremos trocar o conteúdo de uma string, é muito provável que seu tamanho precise mudar também. Como lidar com essas limitações?

A questão do tamanho pode ser contornada, para certos propósitos, de maneira fácil: como uma string contém informação sobre o seu próprio tamanho (indiretamente através do terminador `\0`), podemos declarar um vetor que tenha mais espaço do que o necessário para guardá-la, e ainda será fácil distinguir onde a 'frase' começa e termina. Por exemplo, podemos declarar um vetor de 100 entradas, para guardar uma frase de apenas 20 caracteres, sendo a 21ª entrada o terminador da string; as outras 79 entradas não serão usadas.

Essa solução nos permite alterar posteriormente o conteúdo do vetor para qualquer sequência de caracteres que não ultrapasse o tamanho que definimos inicialmente. O enfado de atribuir elemento por elemento pode ser eliminado graças à função strcpy, que copia o conteúdo de uma string para outra. Mas como isso nos ajuda? Quando precisamos passar um array comum como parâmetro para uma função, não é possível usar uma lista de valores *ad hoc*, mas apenas o nome de uma variável já declarada. Entretanto, as strings são, de certa forma, privilegiadas: é possível usar uma expressão de string com aspas duplas nesse tipo de situação (e em algumas outras também) — tanto que já as utilizamos diversas vezes, particularmente com as funções printf e scanf. Deste modo, usa-se a função strcpy (*string copy*) para copiar para o vetor já declarado uma string "pré-fabricada" — por exemplo:

```
char mensagem[20];
strcpy(mensagem, "Bom dia!");
```

Isso é equivalente a copiar individualmente cada caractere da nossa string pré-fabricada para a variável de destino:

```
char mensagem[20];
mensagem[0] = 'B';
mensagem[1] = 'o';
mensagem[2] = 'm';
mensagem[3] = ' ';
mensagem[4] = 'd';
mensagem[5] = 'i';
mensagem[6] = 'a';
```

```
mensagem[7] = '!';
mensagem[8] = 0;
```

Note que o terminador também é copiado! Assim, na string de destino, devemos garantir que o espaço seja suficiente para guardar todos os caracteres da string original *mais um* (o terminador). Se o destino tem tamanho 20, podemos copiar no máximo uma string de 19 caracteres.

Observe também que não utilizamos o operador de endereço com a nossa variável tipo string, pois ela é um vetor e, como já vimos, vetores são naturalmente passados por referência em parâmetros de funções.

Essa função também pode, naturalmente, ser usada para copiar o conteúdo de uma string para outra quando ambas estão contidas em vetores já declarados:

```
char origem[20] = "Boa noite!";
char destino[20];
strcpy(destino, origem);
```

Como sempre, devemos ter cuidado para não exceder o tamanho do vetor original. Se tentássemos copiar uma mensagem muito longa para o vetor, nosso programa iria tentar gravar dados fora da área reservada para o vetor, o que já vimos que não é uma boa ideia. Por isso, existe também uma outra função, strncpy, que copia o conteúdo de uma string para outra sem exceder um certo número máximo de caracteres (especificado por um parâmetro da função):

strncpy(*destino*, *origem*, *n máximo*);

Por exemplo:

```
char mensagem[20];
strncpy(mensagem, "Copiando uma mensagem muito longa", 20);
```

O resultado deste código será copiar a sequência `Copiando uma mensage`, com exatamente 20 caracteres, para a variável *mensagem*. O problema aqui é que, como a string de origem era mais longa do que o destino poderia suportar, não foi inserido nenhum terminador. Nesse caso, o que podemos fazer é inserir o terminador manualmente, e pedir para copiar no máximo 19 caracteres:

```
char mensagem[20];
strncpy(mensagem, "Copiando uma mensagem muito longa", 19);
mensagem[19] = 0;
```

Caso a origem tenha tamanho menor do que o tamanho máximo especificado, o terminador será copiado automaticamente.

Já vimos, na seção anterior, como é possível encontrar o tamanho de uma string. Como esse código é muito usado, existe uma função da biblioteca padrão que faz exatamente a mesma coisa: strlen (*string length*). Seu uso é bem simples:

```
char mensagem[50] = "Uma mensagem muito longa";
printf("A mensagem tem comprimento %d.\n", strlen(mensagem));
```

Note que a função devolve o comprimento da string (24), e não o comprimento do vetor que foi usado para guardá-la (que é 50).

Outra tarefa muito comum é descobrir se duas strings são iguais. Como strings são vetores, que são ponteiros, uma comparação do tipo `s1 == s2` compara os *endereços* em que estão guardadas as strings. Obviamente essa comparação só é verdadeira quando comparamos uma string com ela mesma; nosso objetivo é comparar duas strings distintas que, no entanto, possam apresentar o mesmo conteúdo.

Para isso, devemos comparar as strings elemento a elemento; convenientemente há uma função que já incorpora esse trabalho, strcmp (*string compare*). Dadas duas strings s_1 e s_2, `strcmp(s1, s2)` devolve o valor zero caso elas sejam iguais, e um valor diferente de zero caso sejam diferentes. Esse valor também serve para determinar a ordem lexicográfica (a "ordem do dicionário") das duas strings: um número negativo indica que $s_1 < s_2$, e um número positivo indica que $s_1 > s_2$. Por exemplo, dadas s_1 = "verde" e s_2 = "vermelho", temos $s_1 < s_2$ pois as duas palavras coincidem nas 3 primeiras letras, mas, na 4ª letra, 'd' vem antes de 'm'.

A ordem lexicográfica é estabelecida de acordo com a tabela ASCII — por exemplo, números vêm antes de letras maiúsculas, que vêm antes das minúsculas, de modo que "`verde`" é diferente de "`Verde`" (por exemplo, `Vermelho` < `verde` < `vermelho`). Se quisermos fazer uma comparação seguindo outra ordem, precisaremos construir nossa própria função, o que seremos capazes de fazer depois de estudar, no Capítulo 6, o funcionamento interno da função strcmp.

5.7.2 Entrada e saída

Não há uma maneira direta de trabalhar com vetores de números pelas funções de entrada/saída da biblioteca padrão (como scanf e printf); mas, como a entrada e a saída padrão são baseadas num fluxo de caracteres, é possível trabalhar diretamente com strings usando essas funções.

Para imprimir uma string, podemos usar o código `%s` na função printf. Por exemplo:

```
char cor[] = "vermelho";
printf("Meu carro é %s.\n", cor);
```

Para ler uma string, é possível também usar o código `%s` na função scanf; no entanto, há alguns cuidados que devem ser tomados, como explicaremos adiante. É agora o momento de introduzir uma nova função: fgets. A novidade dessa função é que ela também pode ser usada para ler uma string de um *arquivo*, e portanto precisamos de um parâmetro especial para dizer que o 'arquivo' de que vamos ler é a entrada do usuário pelo teclado — esse parâmetro é `stdin`, que é a abreviação em inglês para *entrada padrão* (*standard input*), que é o termo usado para denotar o canal de comunicação entre o programa e o terminal.[11]

Antes de dizer como se escreve isso em C, vamos observar que, para ler uma string, precisamos de um vetor grande o bastante para guardar os caracteres. Como não sabemos *a priori* quantos caracteres serão digitados, a estratégia que seguimos em geral é a seguinte: começamos reservando um espaço inicial (temporário) para os caracteres, e lemos os dados

[11] Se você se perguntou por que cargas d'água precisávamos usar esse termo técnico em vez de simplesmente dizer 'teclado', adianto que a entrada padrão pode ser *redirecionada* para pegar dados, por exemplo, de um arquivo em vez do teclado, e portanto é uma estrutura flexível que permite diversos tipos de entrada. Veremos um pouco sobre isso mais adiante.

disponíveis até encher esse espaço. Se não houver dados para preencher todo o espaço, terminamos nosso trabalho; se ainda sobrarem dados para ler, precisamos transferir os dados lidos para outro lugar, para poder continuar lendo o restante dos dados, repetindo o procedimento conforme for necessário. O nome que costuma ser dado a esse espaço temporário é **buffer**, que tem exatamente esse sentido — um espaço temporário para armazenar dados provenientes de algum dispositivo (que é o nosso caso, com o teclado), ou destinados a algum dispositivo.

A função fgets precisa, além do parâmetro especial `stdin`, de duas informações: quantos caracteres (no máximo) ler, e onde guardá-los. Isso é feito da seguinte maneira:

fgets(*destino*, *n*, `stdin`);

Um comentário importante faz-se necessário: o parâmetro *n* na verdade indica o número máximo de caracteres da string *menos um*, pois a função fgets sempre adiciona o terminador `\0` após o último caractere lido: o tamanho *n* refere-se ao comprimento da string quando se inclui o terminador. Nesse sentido é preciso prestar atenção à diferença em relação à função strncpy. Assim, se temos um vetor de tamanho 20, podemos chamar a função fgets com o parâmetro $n = 20$, e no máximo 19 caracteres serão lidos e armazenados.

Essa função irá ler *uma linha* da entrada padrão até o máximo de $n - 1$ caracteres. Se a linha tiver menos do que isso, a leitura acabará no final da linha; se a linha tiver mais do que isso, a leitura acabará no $(n - 1)$-ésimo caractere.

É possível ler strings usando a função scanf de duas maneiras diferentes, ambas um pouco diferentes do funcionamento de fgets. Para evitar problemas de acesso a lugares errados na memória, é sempre necessário especificar o número máximo *n* de caracteres que poderão ser armazenados (Nos itens abaixo, no meio dos códigos, entenda um *n* como esse número, e não como uma letra `n` a ser digitada literalmente.)

- Com o código `%`*n*`c`, serão lidos *n* caracteres da entrada padrão, incluindo espaços e quebras de linha. O terminador `\0` **não** é incluído.
- Usando o código `%`*n*`s`, são lidos no máximo *n* caracteres da entrada padrão, parando assim que encontrar algum espaço em branco[12] — que não será armazenado na string. Um terminador é incluído automaticamente ao final da string, mas o tamanho *n* não o leva em conta. Ou seja, o vetor deve ter espaço para $n + 1$ caracteres.

É necessário prestar atenção à diferença de comportamento das diferentes funções em relação à inclusão do terminador da string. Sempre que estiver em dúvida, consulte o manual da biblioteca padrão ou, melhor ainda, faça um programa simples para testar!

5.8 Mais sobre entrada e saída

[12] **Espaço em branco** é a denominação genérica para os caracteres que representam algum tipo de espaçamento, seja horizontal ou vertical — a saber, o espaço comum ‘ ’ (ASCII 32), a tabulação horizontal ‘\t’ (ASCII 9) e quebras de linha (‘\n’, ASCII 10, ou o retorno de carro ‘\r’, ASCII 13), entre outros de uso mais raro.

Tabela 5.1: Resumo das funções de manipulação de strings

Função	Descrição
`strcpy(`*dest*`, `*origem*`)`	Copia todo o conteúdo de *origem* para *dest*.
`strncpy(`*dest*`, `*origem*`, `*n*`)`	Copia no máximo *n* caracteres de *origem* para *dest*.
`strlen(`*str*`)`	Devolve o tamanho da string *str*.
`strcmp(`*s1*`, `*s2*`)`	Compara as strings *s1* e *s2*, e devolve 0 caso elas sejam iguais.
`fgets(`*dest*`, `*n*`, stdin)`	Lê uma linha da entrada padrão, com no máximo $n-1$ caracteres, armazenando em *dest*.

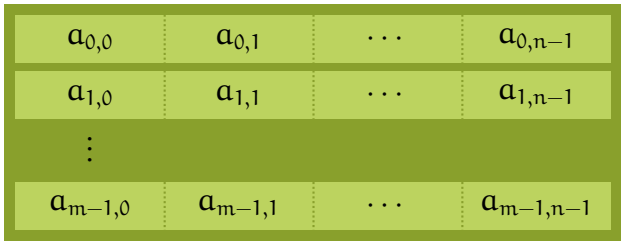
6 Algoritmos

Mais ponteiros 7

7.1 Matrizes

Os vetores que vimos até agora eram usados para guardar variáveis escalares; vamos explorar agora outra possibilidade: usar um vetor para guardar um conjunto de vetores. Por exemplo, se temos 3 vetores de 5 inteiros, podemos criar um vetor que contém esses 3 vetores, e podemos acessar os inteiros usando dois índices: primeiro o índice que identifica cada um dos três vetores, depois o índice que identifica cada inteiro dentro de cada vetor. Podemos interpretar isso como uma matriz: o primeiro índice indica a linha em que um elemento está, e o segundo indica a posição (coluna) desse elemento dentro da linha correspondente.

Em suma, nessa representação, cada linha de uma matriz é um vetor de n números, e a matriz é um vetor de m vetores-linha, formando assim uma matriz $m \times n$ (m linhas, n colunas). A seguir vê-se uma ilustração dessa representação, na qual as barras mais claras representam os vetores-linha, que estão contidos na caixa mais escura, que corresponde à matriz:



Poderíamos também inverter nessa representação o papel das linhas e colunas — isto é, o índice principal seria o da coluna e o secundário seria uma posição (linha) dentro dessa coluna. Preferimos manter a linha como índice principal para cooperar com a convenção dos matemáticos, mas não há nenhuma razão computacional que nos obrigue a escolher uma dessas interpretações em detrimento da outra. A estrutura computacional subjacente só determina que há uma hierarquia de índices, que podemos interpretar como quisermos.

Para declarar uma variável do tipo matriz, usamos a seguinte sintaxe, muito semelhante à sintaxe de vetores:

tipo_de_dados nome_matriz[*linhas*][*colunas*];

Aplicam-se as mesmas observações apontadas para os vetores: os números de linhas e colunas devem ser *constantes*, e os índices dos elementos são numerados a partir do zero.

Podemos inicializar matrizes de maneira similar à dos vetores, introduzindo dois níveis de chaves — os internos para agrupar os elementos de uma mesma linha, os externos para agrupar as linhas. No entanto, não é possível deixar os colchetes de tamanho vazios; é necessário preencher pelo menos o último:

```
int matriz[3][2] = {{2, 3}, {5, 7}, {9, 11}};  /* ok */
int matriz[][2] = {{2, 3}, {5, 7}, {9, 11}};   /* ok */
int matriz[][] = {{2, 3}, {5, 7}, {9, 11}};    /* inválido! ⟵
    */
```

As matrizes que criamos também são chamadas de *vetores de duas dimensões*; também é possível criar vetores com mais do que duas dimensões — ou seja, vetores com mais do que dois índices, como `vetor[2][3][1]` —, e não é difícil deduzir como se faz isso. Se um vetor tem d dimensões, cada índice tem um intervalo de valores possíveis — no caso das matrizes, o primeiro índice variava entre os números de linhas e o segundo entre os números de colunas. Dizemos que n_j é o comprimento do vetor ao longo da dimensão j ($j = 1, 2, \ldots, d$), o que equivale a dizer que o j-ésimo índice varia de 0 até $n_j - 1$. Para criar um vetor de d dimensões, com n_j entradas ao longo da dimensão j ($j = 1, 2, \cdots, d$), fazemos o seguinte:

tipo_de_dados nome_vetor$[n_1][n_2]\cdots[n_d]$;

7.2 Alocação dinâmica de memória

É comum a necessidade de lidar com uma quantidade de dados cujo tamanho é imprevisível no momento em que escrevemos nosso programa; por exemplo, suponha que queremos ler todo o conteúdo de um arquivo que está armazenado no computador, e armazenar numa variável. Essa variável deverá ser um vetor de caracteres, mas não sabemos, ao escrever nosso programa, o tamanho que isso poderá ocupar. Se inventarmos um tamanho máximo, reservando, por exemplo, espaço para um vetor de 100 000 entradas, corremos o risco de nos deparar com um arquivo maior do que isso, e dessa situação não há muita saída, pois não há como aumentar o tamanho desse vetor posteriormente.

Para lidar com esse tipo de situação de maneira bem mais elegante e prática, a biblioteca padrão do C inclui um conjunto de funções que permitem a **alocação dinâmica de memória**; são funções que, ao serem chamadas, pedem ao Guardião da Memória[13] um pedaço de memória de um certo tamanho, e, se o pedido for aceito (isto é, se houver memória disponível), devolvem o endereço de um pedaço de memória conforme pedido (isto é, um *ponteiro* para a região de memória). A declaração de um vetor como fazíamos anteriormente, em contrapartida, é chamada de *alocação estática de memória*, pois ela é pré-estabelecida na compilação do programa, ao contrário da alocação dinâmica que é de fato realizada durante a execução do programa.

Isso soluciona um dos nossos antigos problemas: a declaração de vetores ou matrizes lidos pelo usuário. Você deve se lembrar que não tínhamos como declarar um vetor cujo tamanho seja uma variável lida do usuário — éramos obrigados a estipular um tamanho limite e reservar um espaço desse tamanho, mesmo que fôssemos usar menos (e nos impedindo de usar mais). Usando a alocação dinâmica, podemos, sabendo o número n de dados a serem

[13] Vulgo sistema operacional.

lidos, pedir um pedaço de memória no qual caibam n inteiros (ou variáveis de qualquer outro tipo). O bom disso é que, como a porção de memória alocada *é contígua* (ou seja, sem buracos no meio), ela pode ser usada como se fosse um vetor comum.

7.2.1 Mãos à obra

Para pedir um bloco de memória, você usará a função malloc, passando como parâmetro o *número de bytes* de que você precisa:

```
ponteiro = malloc(num_bytes);
```

No entanto, na maioria das vezes, queremos alocar espaço para um certo *número de dados* ou de elementos de um vetor. Resta então saber o número de bytes que cada dado (ou elemento) ocupa. Para isso, usaremos o operador `sizeof`, que diz quantos bytes ocupa uma variável de um certo tipo. Veja como ele é usado a partir deste exemplo:

```
printf("O tamanho de um char é %d bytes\n", sizeof(char));
printf("O tamanho de um int é %d bytes\n", sizeof(int));
printf("O tamanho de um double é %d bytes\n", sizeof(double));
```

Você sempre deve usar o operador `sizeof` em vez de assumir, por exemplo, que o tamanho de um inteiro é de 4 bytes. Isso poderia causar grandes problemas quando o programa for transportado para um sistema em que o tamanho é diferente, além de não deixar claro o que significa aquele "4" no meio do código.

Dito isso, temos todo o material necessário para realizar a alocação dinâmica de um vetor. Notemos que, por exemplo, se um inteiro ocupa 4 bytes, então um vetor de n inteiros ocupará $4n$ bytes, podemos escrever a rotina de alocação desta maneira:

```
int    *v_int;
double *v_double;
char   *v_char;
v_int    = malloc(n * sizeof(int));
v_double = malloc(n * sizeof(double));
v_char   = malloc(n * sizeof(char));
```

Note que o endereço devolvido pela função malloc deve ser armazenado num ponteiro do tipo apropriado; ou seja, o espaço alocado para inteiros deve ser armazenado num ponteiro `int *`, e assim por diante.

Se a memória tiver sido alocada com sucesso, poderemos acessar esses ponteiros como vetores normais:

```
v_int[0] = -5;
v_int[1] = 4;
scanf("%d", &v_int[2]);
v_double[5] = (v_int[0] + v_int[1]) * 1.0/v_int[2];
```

Quando você terminar de usar um bloco de memória alocado dinamicamente, você **deve sempre** *liberá-lo* usando a função free com o ponteiro correspondente como argumento, como em `free(ponteiro)`. Ao *liberar* um pedaço de memória, o sistema operacional retoma a guarda dele, permitindo que ele seja posteriormente usado por outros programas.

Daí surge uma possível fonte de erros: se você tentar acessar um ponteiro depois que a memória tiver sido liberada, essa memória poderá estar sendo usada por outro programa, e portanto não é nada legal mexer nesse pedaço de memória! Ponteiros que apontam para um pedaço de memória que foi desalocado são chamados, em inglês, de *dangling pointers* (literalmente, "ponteiros pendentes"), que vou preferir traduzir como *ponteiros desapropriados*.

Assim, ao liberar um bloco de memória, o ponteiro que apontava pra ele torna-se "inválido", e você não deve tentar usá-lo novamente (a menos que reaponte o ponteiro para outro bloco de memória que você esteja permitido a usar). Uma solução que é usada com certa frequência é transformar o ponteiro em um ponteiro nulo após a liberação da memória — um ponteiro nulo é simplesmente um ponteiro que aponta para o endereço zero (a expressão `NULL` é simplesmente um "apelido" para o número zero que não causa problemas quando usada como endereço), que não é usado para nenhuma área válida da memória. Um exemplo disso é o seguinte:

```
free(ponteiro);
ponteiro = NULL;
```

O mérito dessa solução está no fato de ser muito mais fácil verificar se um ponteiro é nulo do que verificar se ele aponta para um lugar impróprio — tanto dentro do seu programa quanto pelo sistema operacional. Quando o programa tenta acessar um ponteiro nulo, o sistema detecta a tentativa e encerra a execução do programa; acessos a ponteiros desapropriados nem sempre podem ser detectados.

7.2.2 Falta de memória

Pode acontecer de haver algum erro e o sistema não conseguir alocar a memória que você pediu — geralmente porque não há mais memória disponível. Nessas situações, a função malloc devolverá um ponteiro nulo (`NULL`), que não aponta para região nenhuma da memória; coisas terríveis acontecerão se você tentar acessá-lo (desreferenciá-lo). Por isso, você sempre deve verificar se o ponteiro devolvido é válido, usando um código parecido com esse:

```
ponteiro = malloc(tamanho);
if (ponteiro == NULL)
{
   printf("Socorro! Não foi possível alocar memória!\n");
   /* executar alguma ação para sair deste imbróglio */
}
```

Sendo essa uma tarefa comum, que será executada várias vezes no programa, é útil escrever uma função que cuida desse trabalho repetitivo:[14]

```
void *mallocX(size_t tamanho)
{
   void *ponteiro;
   ponteiro = malloc(tamanho);
   if (ponteiro == NULL)
   {
      printf("Socorro! Não foi possível alocar memória!\n");
```

[14] Tanto a ideia quanto o nome da função foram inspirados nas notas do Prof. Paulo Feofiloff [1].

```
      exit(EXIT_FAILURE);
   }
}
```

A função exit que foi aqui usada serve para terminar a execução do programa imediatamente, como se tivesse terminado a função main. O argumento `EXIT_FAILURE` (uma constante definida pelo sistema) indica ao sistema operacional que houve algum erro na execução do programa. O efeito dessa instrução é equivalente a voltar à função main e escrever `return EXIT_FAILURE;`. No entanto, usar a função exit é mais prático pois assim não é necessário nos preocupar com a parte do programa em que estávamos.

7.3 Ponteiros duplos

Ainda não vimos como é possível alocar memória dinamicamente para uma matriz, ou qualquer vetor de mais de uma dimensão. Podemos pensar em duas maneiras de fazer isso; a primeira delas é uma não-solução: para criar uma matriz de m linhas e n colunas, alocamos um vetor de $m \times n$ elementos, convencionando que o primeiro grupo de n elementos corresponde à primeira linha, que o segundo corresponde à segunda linha, e assim por diante. Dessa maneira, para acessar um elemento (i, j) da matriz (usando $0 \leq i < m$ e $0 \leq j < n$), temos de fazer uma conta para descobrir em que posição k do vetor ele se encontra. Os elementos j da primeira linha ($i = 0$) podem ser acessados simplesmente pelo índice j; os elementos j da segunda linha ($i = 1$) podem ser acessados somando n ao índice j; prosseguindo assim, é fácil concluir que o "índice linearizado" do elemento (i, j) é $k = j + i \cdot n$.

Da mesma maneira podemos proceder para vetores de d dimensões, de comprimentos $(n_1, \ldots, n_d)$; os índices são denotados $(i_1, \ldots, i_d)$. O primeiro grupo de elementos corresponde aos elementos com o primeiro índice igual a zero ($i_1 = 0$); dentro desse grupo, haverá vários subgrupos, cada um correspondendo a um valor do segundo índice (i_2); e assim por diante. Não é difícil completar o raciocínio para encontrar a fórmula do índice linearizado do elemento $(i_1, \ldots, i_d)$. (Veja a resposta no rodapé.[15])

Essa maneira tem a vantagem de só necessitar de uma alocação de memória, com o (pequeno) custo de precisar fazer uma conta para encontrar as posições de cada elemento. Na verdade, ao alocar uma matriz estaticamente, é dessa maneira que o computador trabalha: com um grande bloco de memória, para o qual os índices multidimensionais são linearizados automaticamente pelo compilador.

Antes de prosseguir, vamos mostrar um exemplo simples de uso dessa técnica:

```
void exemplo_matriz(int m, int n)
{
   /* aloca a matriz de tamanho (m, n) */
   int *matriz = mallocX(m*n * sizeof(int));
   int i, j;

   /* preenche cada elemento com a posição linearizada
      correspondente */
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        matriz[j + i*n] = j + i*n;
```

[15] $k = i_d + n_d \left(i_{d-1} + n_{d-1} \left(\cdots (i_2 + n_2 i_1) \cdots \right) \right)$

```
}
```

A outra maneira de criar uma matriz dinamicamente permite que ela seja acessada com a mesma linguagem que as matrizes estáticas (`matriz[i][j]`), porém aumentando a carga computacional tanto do acesso aos elementos quanto da alocação da matriz. Para matrizes bidimensionais (m, n), ela consiste em alocar m vetores de n elementos, e alocar um vetor de m ponteiros no qual serão colocados os m vetores. É importante notar que os elementos desse último vetor são *ponteiros* para o tipo de dados que queremos guardar, o que gera uma pequena diferença no cálculo do tamanho de cada elemento:

```
void exemplo_matriz(int m, int n)
{
   /* aloca o vetor que guardará as m linhas */
   int **matriz = mallocX(m * sizeof(int*));
   int i, j;

   /* aloca cada linha com n elementos */
   for (i = 0; i < m; i++)
     matriz[i] = mallocX(n * sizeof(int));

   /* preenche cada elemento com a posição linearizada
      correspondente */
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        matriz[i][j] = j + i*n;
}
```

A carga computacional da alocação é evidente: em vez de fazer uma alocação, fizemos $m+1$ alocações. Já para acessar os elementos `matriz[i][j]`, devemos observar, em primeiro lugar, que essa expressão é equivalente a `(matriz[i])[j]`, ou seja, que primeiro devemos olhar para o vetor de linhas para achar a posição de memória em que está a linha `i` (não há nenhuma garantia de que as m linhas estejam guardadas em posições contíguas), e depois ir até essa posição e encontrar o `j`-ésimo elemento.

Dessa forma, é bom um pouco de cuidado ao escolher essa segunda técnica, pois, apesar de mais fácil quanto ao acesso dos elementos, ela pode ser menos eficiente. Isso não significa que ela nunca deve ser usada; em cada caso pode-se escolher o que for mais apropriado.

Na verdade, podemos melhorar um pouco a segunda técnica usando a ideia da primeira técnica de alocar um grande bloco de memória de mn elementos em vez de alocar m blocos de n elementos. Dessa maneira, as $m + 1$ alocações de memória reduzem-se a apenas duas, com o custo de ter de calcular as localizações de cada linha para inserir no vetor `matriz`.

7.4 Ponteiros para funções

Começo com um exemplo simples: você quer estimar a derivada de uma função f num ponto x_0, através de avaliações sucessivas do quociente de Newton,

$$\frac{f(x) - f(x_0)}{x - x_0},$$

para valores de x cada vez mais próximos de x_0. O algoritmo dessa operação é bastante simples, e é o mesmo para qualquer função f. Seria interessante, portanto, ter um mecanismo

de criar em C uma função genérica `calcula_derivada` que fizesse isso, dada uma função `f` e um ponto `x0`, bastando escrever um código que fosse, esquematicamente, como a seguir:

```
calcula_derivada(f, x0);
```

Em C, o que passaríamos para a função `calcula_derivada` seria não a função `f` em si, mas um **ponteiro** para ela. Dessa maneira, a função `calcula_derivada` poderia chamar a função `f` sem saber de antemão quem ela é: ela apenas chamaria a função que é apontada pelo ponteiro que foi dado.

Agora vamos ver como se faz isso.

7.4.1 Declarando um ponteiro para uma função

A declaração de um ponteiro para uma função pode parecer um pouco capciosa (de fato, não é das mais simples), mas não chega a ser um bicho-de-sete-cabeças. O caso mais simples seria o de um `ponteiro` para uma função sem parâmetros e que devolve um valor de um certo `tipo`. Sua declaração seria feita como:

```
tipo (*ponteiro)();
```

Os parênteses em torno de `*ponteiro` são absolutamente necessários; se não os usássemos, teríamos (verifique) uma declaração de uma *função que devolve um ponteiro* — algo totalmente diferente!

Se a função tiver parâmetros, você deve colocar seus *tipos*[16] dentro do segundo par de parênteses. Note bem: você só deve colocar os tipos dos parâmetros — não coloque os nomes. Por exemplo, se tivéssemos uma função que devolve um inteiro e recebe um inteiro e um número real, um ponteiro para ela seria declarado assim:

```
int (*ponteiro)(int, float);
```

Obviamente, como você já deve saber, um ponteiro não serve para nada se ele não aponta para algo que conhecemos. Pois bem, para que um ponteiro aponte para uma função, procedemos da mesma maneira que para os ponteiros para variáveis:

```
ponteiro = &funcao;
```

Muitos compiladores (o GCC inclusive) permitem que você omita o E comercial ao criar um ponteiro para uma função. No entanto, é recomendado que você o escreva explicitamente para garantir a máxima portabilidade do seu código.

Talvez você tenha achado estranho usarmos o *endereço* de uma função. Mas, exatamente como ocorre com as variáveis, as funções são guardadas em algum lugar da memória quando o programa é carregado, de modo que elas também têm um endereço.

7.4.2 Chamando uma função através de um ponteiro

Se você tem um `ponteiro` que aponta para uma função, você pode chamar a função desreferenciando o ponteiro desta maneira:

[16] Novamente, tudo isso é necessário porque o compilador precisa saber que parâmetros a função está esperando, para poder fazer as manipulações corretas dos dados na memória.

```
(*ponteiro)(parametro1, parametro2, ...);
```

Novamente, você não deve se esquecer dos parênteses em volta de `*ponteiro` — eles estão lá para indicar que a função a ser chamada é a função resultante da desreferenciação de `ponteiro`; se esquecêssemos dos parênteses, estaríamos *chamando a função `ponteiro` e desreferenciando o valor por ela devolvido*. Se você quer simplificar sua vida, você pode usar a sintaxe abreviada (apenas para a *chamada*), que é escrita exatamente como uma chamada de função comum:

```
ponteiro(parametro1, parametro2, \ldots);
```

Você também pode, naturalmente, "capturar" o valor da função, da maneira usual:

```
x = (*ponteiro)(parametro1, parametro2, ...);
x = ponteiro(parametro1, parametro2, ...);
```

7.4.3 Ponteiros para funções como parâmetros de funções

Uma das grandes utilidades dos ponteiros para funções reside na possibilidade de passá-los entre funções, como parâmetros. Uma vez compreendida a maneira de declarar e acessar ponteiros para funções, não é difícil usá-los como parâmetros. No cabeçalho da função, um ponteiro para função é especificado da mesma maneira que seria declarada uma variável do mesmo tipo. Veja o exemplo:

```
int operacao(int a, int b, int (*funcao)(int, int))
{
   return (*funcao)(a, b);
}
```

7.5 Escopo de variáveis

7.6 Funções recursivas

8 Estruturas

8.1 Structs

Em cada parte de um programa geralmente há várias variáveis associadas à realização de uma tarefa específica. Por causa disso, é conveniente ter um modo de agrupar um conjunto de variáveis relacionadas. Conhecemos anteriormente os vetores, que são agrupamentos de uma série de variáveis do mesmo tipo, cada uma identificada por um número.

Se, por outro lado, quisermos um tipo de agrupamento que englobe variáveis de tipos diferentes, ou no qual cada variável possa ser identificada por um nome específico, usamos um tipo de estrutura chamado de **registro** (mais conhecido por seu nome em inglês, **struct**, uma abreviação de *structure*, 'estrutura').

Esse recurso da linguagem C permite que o usuário defina seus próprios tipos de dados a partir dos tipos primitivos da linguagem. Esse tipo de estrutura é um exemplo de *tipo de dados composto* — o segundo a ser apresentado aqui, os vetores tendo sido o primeiro.

Um *struct* em C funciona de maneira similar a um registro (uma linha) em um banco de dados: ele contém um conjunto de variáveis, que têm tipos fixados e são identificadas por nomes (como as variáveis comuns). Por exemplo, um registro que representa um produto numa compra pode conter as seguintes variáveis:

```
char *descricao;
int quantidade;
double preco_unitario;
double desconto;
double preco_total;
```

Essas variáveis são denominadas **campos** ou **membros** do registro. O conjunto de nomes e tipos dos campos constitui um tipo de registro. Cada registro é em si uma variável, e tem boa parte dos privilégios de uma outra variável qualquer.

Um registro é declarado usando a palavra-chave **struct** seguida de um bloco (delimitado por chaves) contendo as declarações dos membros, como se fossem declarações de variáveis comuns. Um registro como exemplificado acima poderia ser declarado como no código a seguir, em que ele é armazenando numa variável chamada `produto`:

```
struct {
   char *descricao;
```

```
    int quantidade;
    double preco_unitario;
    double desconto;
    double preco_total;
} produto;
```

Podemos usar esse tipo de comando para declarar vários registros ao mesmo tempo, colocando vários nomes de variável em vez de apenas um (separando por vírgulas). Contudo, essa forma de declaração não permite reutilizar o mesmo tipo de registro em outro comando posterior.

Para resolver essa inconveniência, basta dar um nome ao tipo de registro: iniciamos sua declaração por `struct` *`nome`* (substituindo o nome escolhido, que segue as mesmas regras de sempre para nomes de coisas em C), em vez de apenas `struct` — assim, *declaramos um tipo de registro*, não apenas um registro. Daí em diante, é só digitar `struct` *`nome`* para se referir ao tipo já declarado.

Também é possível declarar o tipo sem declarar nenhuma variável desse tipo, bastando para isso não colocar nenhum nome de variável; é necessário manter o ponto-e-vírgula, no entanto. Com essa possibilidade, é interessante separar em comandos diferentes a declaração do tipo (que fica na parte exterior do programa) e as declarações de variáveis desse tipo (que ficam no lugar certo para cada variável), como no exemplo a seguir:

```
/* Declaração do tipo info_produto */
struct info_produto {
    char *descricao;
    int quantidade;
    double preco_unitario;
    double desconto;
    double preco_total;
};

/* Declaração de duas variáveis desse tipo */
struct info_produto produtoA, produtoB;
```

Para acessar campos de um registro, usamos o operador `.` (um ponto), colocando à esquerda dele o nome da variável que contém o registro, e à direita o nome do campo. Ainda no exemplo anterior, podemos acessar o preço do produto A usando a expressão `produtoA.preco_unitario`, como se fosse uma variável comum.

Podemos, por exemplo, tomar o endereço de um membro de um registro e com isso criar um apontador que aponte para esse membro. No exemplo a seguir, utilizamos uma função que, supostamente, calcula o preço total de um item (dadas as informações necessárias nos três primeiros argumentos) e o armazena na variável apontada pelo quarto argumento.

```
calcular_preco_total(produtoA.preco_unitario, produtoA.⟵
    quantidade, produtoA.desconto, &produtoA.preco_total);
```

8.1.1 Registros e ponteiros

Resumo

Declaração de um tipo de registro

```
struct nome_do_tipo {
  /* declarações dos membros */
};
```

Declaração de um registro

```
struct nome_do_tipo nome_da_variavel;
```

Acesso de membros de um registro

```
variavel.nome_do_membro
```

8.2 Listas ligadas

Compilação

Tabelas de referência

B

Referências Bibliográficas

[1] Paulo Feofiloff, *Projeto de Algoritmos*, http://www.ime.usp.br/~pf/algoritmos/

[2] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, second edition, 1988.