Construção de Algoritmos e Programação

Programação em C

Jander Moreira

24/08/2024

Apresentação

> **!** Importante
>
> Este material está **em produção** e, assim, é necessária atenção quanto à precisão do conteúdo e às possíveis alterações que serão promovidas ao longo do tempo.
> Versão 0.1-alfa

A disciplina de graduação *Construção de Algoritmos e Programação*, que é ofertada regularmente pelo Departamento de Computação para os cursos de Bacharelado em Ciência da Computação e Bacharelado em Engenharia da Computação da Universidade Federal de São Carlos, motivou a escrita deste livro, pensando em uma abordagem distinta da usualmente feita em cursos básicos de programação.

A versão *Programação em C*

A linguagem C é uma linguagem básica, na qual a proximidade do código com as representações internas da memória é uma característica importante. Outras linguagens possuem nível de abstração mais alto, ocultando muitos detalhes do programador, como é o caso de Python e R, por exemplo.

Na experiência do autor, dominar minimamente uma linguagem de programação de nível de abstração mais baixo auxilia qualquer programador a entender muitos dos aspectos, sejam vantagens ou ciladas, existentes em qualquer outra linguagem procedural ou mesmo orientada a objetos. Isso torna aprender C uma experiência efetivamente enriquecedora.

As questões mais básicas da linguagem C são o assunto deste texto, fornecendo uma visão geral da codificação e de elementos de memória e representação que compõem um conhecimento precioso para quem desenvolve programas.

Disponibilidade *online*

- Algoritmos para quem já sabe programar: https://jandermoreira.github.io/cap-algoritmos
- Programação em C : https://jandermoreira.github.io/cap-linguagem-c
- Prática com algoritmos: https://jandermoreira.github.io/cap-pratica-algoritmos

Programação em C

Existe uma infinidade de linguagens de programação. Algumas são mais fáceis, outras mais complexas, umas mais abstratas, outras muito concretas. A linguagem C se encontra entre as linguagens mais básicas, ou seja, mais próximas ao hardware. Faz parte do grupo conhecido como linguagem de nível baixo.

C abstrai o conceito do processador: o código fonte é escrito de forma a ser independente de qual processador é usado e de qual sistema operacional está instalado. Por outro lado, a abstração em relação à memória é pequena. Em programas mais simples é possível ignorar completamente como a memória é usada para armazenar os dados; porém basta um pequeno aumento na complexidade do problema (e também de sua solução) que questões como quantidade de bytes disponíveis, localização dos dados na memória ou outros aspectos emergem e têm que ser trabalhadas pelo programador.

Esse aspecto da linguagem que exige um conhecimento mais concreto de como a memória é usada e como as instruções devem ser organizadas gera, por uma lado, um ambiente desafiador e até um pouco intimidador, mas, de outro ponto de vista, traz um conhecimento mais sedimentado da programação, o que é algo bastante enriquecedor.

O conteúdo da linguagem C oferecido neste livro não é maior do que muitos outros encontrados por uma simples busca em qualquer ferramenta de busca disponível. O que talvez torne este material diferenciado é a abordagem, a qual visa proporcionar um bom domínio dos principais elementos de forma crescente. O enfoque é a apresentação dos conceitos de forma simples (tanto quanto possível), consolidando-os antes de progredir para a próxima etapa.

Conteúdo

Apresentação i

Programação em C ii

1 Noções de algoritmos 1

I Programação básica 7

2 Introdução à linguagem C 8

3 Tipos de dados da linguagem C 16

4 Variáveis e leituras em C 22

5 Expressões aritméticas de C 43

6 C: expressões relacionais e lógicas 54

II Controle de fluxo simples 67

7 Execução condicional com `if` 68

8 Execução condicional com `switch` 76

III Organização geral do código 81

9 Escopo básico de declarações 82

10 Organização do código fonte 87

IV Controle de repetição do fluxo 101

11 Repetições com `while` 102

12 Repetições com `for` 111

13 Repetições com `do while` 118

14 Arquivos texto 122

15 Desvirtuação das repetições 141

V Cadeias de caracteres **146**

16 Dados textuais em C **147**

17 Manipulação de dados textuais em C **154**

VI Modularização e memória **166**

18 Funções em C **167**

19 Regras de escopo com a modularização **181**

20 Endereçamento de memória e ponteiros nos programas **190**

21 Procedimentos em C **200**

22 Parâmetros das funções na linguagem C **203**

VII Estruturação de dados **211**

23 C: Dados com `struct` **212**

24 C: Dados em vetores **226**

Índice Remissivo **235**

1 Noções de algoritmos

Um algoritmo é uma descrição de passos que, se seguidos, levam à solução de um determinado problema e, genericamente falando, quaisquer instruções, orientações ou coisa similar podem ser classificadas como um algoritmo.

A partir dessa óptica, há uma infinidade de algoritmos. Quando se trata de computação e mais especificamente de programação, há um subconjunto dos algoritmos com características mais particulares. Este capítulo trata da contextualização dos algoritmos computacionais.

1.1 Características gerais dos algoritmos

Por exemplo, as instruções de como higienizar uma máquina de lavar roupas são um algoritmo. Esse algoritmo parte do pressuposto de uma máquina de lavar para ser higienizada, descreve os passos para serem seguidos (deixar o cesto vazio, acrescentar a água sanitária, deixar em um ciclo específico por um determinado tempo) e atinge o resultado desejado, que é a máquina limpa.

A montagem de uma estante comprada *online* também segue o algoritmo estabelecido no manual de montagem, tendo como objetivo partir de um conjunto de peças separadas e obter a estante montada e funcional. Os passos passam pela verificação da disponibilidade de todas as peças e ferramentas necessárias, montagem organizada das diversas partes e devidas finalizações.

Um último (e clássico) exemplo é uma receita culinária, a qual parte dos ingredientes constituintes e chega a um bolo, um assado ou outro prato qualquer.

Todos esses algoritmos possuem três elementos principais:

- A situação inicial;
- A sequência de passos que devem ser seguidos;
- A situação final.

A situação inicial são as pré-condições, ou seja, o que tem haver antes da execução dos passos para que todas as ações possam ser seguidas de forma adequada. Os passos determinam as ações que devem ser executadas e uma ordem coerente para que aconteçam. As pós-condições caracterizam a situação final, ou seja, a completude do que o algoritmo se propôs a resolver.

Na Tabela 1.1 são apresentados esses elementos para dois exemplos específicos, ilustrando-os de forma simplificada.

Tabela 1.1: Exemplos ilustrando os elementos (pré-condições, passos, pós-condições) para dois problemas específicos.

	Cocção de um pão	Atualização de um saldo bancário
Pré-condições	Disponibilidade dos ingredientes e utensílios necessários	O saldo anterior e todas as movimentações no período
Passos	Preparação da massa, descanso, crescimento, forno	Atualização do saldo passando-se por cada movimentação individual

	Cocção de um pão	Atualização de um saldo bancário
Pós-condições	Pão	O saldo atualizado

Algoritmo

Um algoritmo pode ser definido como uma sequência finita de passos que levam de uma situação inicial (pré-condições) a uma situação final (pós-condições) de forma bem definida. A partir desse conceito, é esperado que, a partir do mesmo estado inicial e seguidos os mesmos passos, o estado final seja atingido.

Esta definição não se aplica, em particular, à receita do pão indicada na Tabela 1.1. O resultado tende a variar consideravelmente dependendo de uma variedade de situações não mencionadas, como o tipo e a qualidade da farinha, a temperatura ambiente que influencia no crescimento da massa e o forno usado, que pode aquecer mais ou menos que outro forno, por exemplo. Para se garantir um resultado sempre "igual", todas essas variáveis deveriam entrar nas pré-condições. Felizmente, essas variações são toleradas no resultado final da receita, sendo até esperadas tais diferenças. As pré-condições e pós-condições podem, dependendo do caso, ter graus de especificidade variados.

Essa variação de resultados, porém, não é tolerada na atualização do saldo bancário. Dado o mesmo saldo inicial e as mesmas movimentações, o resultado não pode ser diferente sob nenhuma hipótese. Tem que haver uma previsibilidade do resultado. Neste caso, pré e pós-condições são bastante determinísticas.

Os algoritmos com resultados e passos mais maleáveis, que toleram certas variações no resultado final, enquadram-se como algoritmos gerais. Entre eles estão as receitas, instruções de montagem de móveis, orientações para se chegar a um destino com GPS ou instruções de como inserir um novo contato na agenda do telefone. Em todos eles, até a vivência e experiências pessoais de quem os executa podem ter influência no resultado. Há pessoas com ótima mão para fazer bolos, por exemplo.

Se for pedido a um humano que converta 95 Fahrenheit para graus Celsius, ele pode usar seu *smartphone* para abrir uma ferramenta de busca, digitar "quanto é 95 farenheit em celsius" (sim, com o erro de digitação) e obter a resposta de 35ºC. Ele poderia estar sem bateria e optado por usar um computador para fazer a busca; poderia também ter escolhido uma ferramenta de busca no lugar de outra; poderia até ter digitado o texto da consulta de diversas outras maneiras distintas. Esse humano poderia também ter boa memória e se lembrar da fórmula, além de ter facilidade para fazer contas de cabeça e dar o resultado sem nenhum outro recurso a não ser ele mesmo.

1.2 Algoritmos computacionais

Existe uma classe particular de algoritmos para os quais há um maior rigor nos mais diversos aspectos e eles não podem depender da experiência ou interpretação de que os executa. Esses são os algoritmos escritos para serem executados, em última instância, por um sistema computacional (processador e memória eletrônicos) e, para tanto, têm que ser extremamente claros e precisos em cada instrução a ser executada, bem como possuem pré-condições e pós-condições bastantes específicas.

Caso um sistema computacional, ou seja, um programa, deva realizar a mesma tarefa, ele tem que ter bem definidas todas as etapas. As pré-condições, por exemplo, definiriam que o valor deveria ser um número real, os passos indicariam o cálculo da conversão e qual seria a expressão usada e, finalmente, o resultado produzido como pós-condição estaria bem definido.

Figura 1.1: Fluxograma para cálculo e apresentação das raízes reais de uma equação de segundo grau.

> **Algoritmos computacionais**
>
> Um algoritmo computacional é aquele que define com clareza todas pré-condições e estabelece também claramente as pós-condições. Também define os passos de forma inequívoca e direta, sem margens para interpretações ou variações. Seu objetivo é ser executado em um sistema computacional.

1.2.1 Fluxogramas

Os fluxogramas são representações visuais com os passos que implementam cada algoritmo. Os símbolos (caixas) possuem formas específicas para cada função e setas as ligam indicando a ordem em que devem ser executadas. Na Figura 1.1 é apresentado um fluxograma para o cálculo das raízes reais de equação de segundo grau e apresentação de mensagens de erro nos casos adequados.

1.2.2 Pseudocódigo

Como alternativa aos fluxogramas, é bastante comum o emprego do chamado pseudocódigo, o qual se assemelha a programas, mas é uma abstração da solução. O Algoritmo 1.1 é apresentado na forma de pseudocódigo.

O Algoritmo 1.1 se refere à mesma solução lógica da Figura 1.1.

Algoritmo 1.1: Pseudocódigo para o cálculo e apresentação das raízes reais de uma equação de segundo grau.

Descrição: Cálculo e apresentação das raízes reais de uma equação de segundo grau na forma $ax^2 + bx + c = 0$
Requer: Os coeficientes a, b e c da equação
Assegura: as raízes reais da equação; ou mensagem que a equação é inválida; ou mensagem que não há raízes reais

Obtenha os valores de a, b e c ▷ *Coeficientes da equação*
se a for igual a zero **então**
  Apresente que a equação não é do segundo grau
senão
  Calcule o discriminante Δ como $b^2 - 4ac$
  se Δ for negativo **então** ▷ *Não há raízes reais*
    Apresente que não há raízes reais
  senão se Δ for igual a zero **então** ▷ *Apenas uma raiz*
    Calcule x como $-\frac{b}{2a}$
    Apresente o valor de x
  senão ▷ *Duas raízes*
    Calcule x_1 como $\frac{-b - \sqrt{\Delta}}{2a}$
    Calcule x_2 como $\frac{-b + \sqrt{\Delta}}{2a}$
    Apresente x_1 e x_2
  fim se
fim se

1.3 Algoritmos e programas

O Algoritmo 1.2 é um algoritmo computacional simples com uma solução para a conversão de temperaturas entre duas escalas termométricas: de graus Celsius para Fahrenheit.

Algoritmo 1.2: Conversão de graus Celsius para Fahrenheit.

Descrição: Conversão de escalas termométricas, de graus Celsius para Fahrenheit
Requer: O valor da temperatura em graus Celsius
Assegura: O valor da temperatura em Fahrenheit

Obtenha *celsius*
Calcule *fahrenheit* como $\frac{5}{9} celsius + 32$
Apresente *fahrenheit*

Para exemplificar como esse algoritmo pode se tornar um programa, seguem exemplos de sua implementação em algumas linguagens distintas, sendo importante salientar a grande variação de formatos de comandos e da estrutura de cada linguagem.

Pascal:

```
(*
Conversão de escalas termométricas, de graus Celsius para Fahrenheit
Requer: a temperatura em graus Celsius
```

```
Assegura: a temperatura em Fahrenheit
*)
program ConversaoTemperaturas;
var
    Celsius, Fahrenheit: real;
begin
    read(Celsius);
    Fahrenheit := 9 / 5 * Celsius + 32;
    write(Fahrenheit:5:2);
end.
```

Python:

```
# Conversão de escalas termométricas, de graus Celsius para Fahrenheit
# Pré-condição: a temperatura em graus Celsius
# Pós-condição: a temperatura em Fahrenheit

celsius = float(input())
fahrenheit = 9 / 5 * celsius + 32
print(f"{fahrenheit:.2f}")
```

C:

```
/*
Conversão de escalas termométricas, de graus Celsius para Fahrenheit
Requer: a temperatura em graus Celsius
Assegura: a temperatura em Fahrenheit
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    fgets(entrada, sizeof entrada, stdin);
    double celsius;
    sscanf(entrada, "%ld", &celsius);

    double fahrenheit = (double)9 / 5 * celsius + 32;
    printf("%.2f", fahrenheit);

    return 0;
}
```

R:

```
# Conversão de escalas termométricas, de graus Celsius para Fahrenheit
# Pré-condição: a temperatura em graus Celsius
# Pós-condição: a temperatura em Fahrenheit

celsius <- as.numeric(readline(""))
fahrenheit <- celsius * 9 / 5 + 32
cat(fahrenheit, "\n")
```

Java:

```
// Conversão de escalas termométricas, de graus Celsius para Fahrenheit
// Pré-condição: a temperatura em graus Celsius
// Pós-condição: a temperatura em Fahrenheit

import java.util.Scanner;

public class ConversorTemperatura {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        double celsius = scanner.nextDouble();
```

```
        double fahrenheit = celsius * 9 / 5 + 32;
        System.out.println(fahrenheit);

        scanner.close();
    }
}
```

Ada:

```
-- Conversão de escalas termométricas, de graus Celsius para Fahrenheit
-- Pré-condição: a temperatura em graus Celsius
-- Pós-condição: a temperatura em Fahrenheit

with Ada.Text_IO; use Ada.Text_IO;

procedure Conversor_Temperatura is
   Celsius : Float;
   Fahrenheit : Float;

begin
   Get(Item => Celsius);
   Fahrenheit := Celsius * 9.0 / 5.0 + 32.0;
   Put_Line(Float'Image(Fahrenheit));
end Conversor_Temperatura;
```

Lua:

```
-- Conversão de escalas termométricas, de graus Celsius para Fahrenheit
-- Pré-condição: a temperatura em graus Celsius
-- Pós-condição: a temperatura em Fahrenheit

local celsius = tonumber(io.read())
local fahrenheit = celsius * 9 / 5 + 32
print(fahrenheit)
```

Existe, claramente, uma distância entre a solução (algoritmo) e sua implementação (código da linguagem).

O principal conceito por trás dos algoritmos é ter uma solução mais abstrata, a qual não se restringe aos detalhes que cada linguagem impõe e, entretanto, apresenta uma solução simples de entender e objetiva quanto a como o problema abordado é resolvido.

Este livro não aborda o desenvolvimento de algoritmos, porém faz uso deles quando necessário, dada a inteção de deixar clara uma solução antes de apresentar sua implementação em C. Esta estratégia visa auxiliar o programador menos experiente ou com menor familiaridade com a linguagem a identificar o que fazem as instruções do programa.

Parte I

Programação básica

2 Introdução à linguagem C

Neste capítulo é feita a apresentação da linguagem C, um pouco de suas origens e a evolução de sua especificação. Juntamente a essa introdução, a compilação e os conceitos de código fonte, objeto e executável também são tratados.

2.1 Origens da linguagem C

Conhecer um pouco do caminho de uma das mais importantes linguagens de programação deve trazer ao programador, de certa forma, uma sensação de pertencimento e autoridade no uso dessa linguagem.

A inspiração da linguagem C veio das linguagens BCPL e B, ambas projetadas para a implementação de sistemas operacionais. Muitos elementos de C se originaram dessas linguagens.

C surgiu como uma linguagem de propósito geral que incorporava os principais mecanismos de controle de fluxo, como condicionais e estruturas de repetição, juntamente com estruturas de dados como arranjos e registros. A linguagem não foi estruturada como tendo um alto nível de abstração e não foca em nenhuma área de aplicação em particular.

Dennis Ritchie projetou e implementou a linguagem C em um DEC PDP-11, o qual usava o sistema operacional UNIX. Na realidade, tanto o compilador C quanto o próprio sistema operacional foram implementados em C. Esse desenvolvimento ocorreu de 1969 a 1973. Também à época, compiladores C já haviam sido implementados para executar em diversas outras máquinas, como equipamentos IBM, Honeywell e Interdata.

Este histórico foi baseado em Richards (1969), Johnson e Kernighan (1973), Ritchie et al. (1978) e Ritchie (1993).

Segundo consulta ao índice de popularidade elaborado pela TIOBE[1], C permanece como uma das linguagens de programação mais populares mundialmente, disputando os primeiros lugares com Python, C++ e Java (dados de novembro de 2023).

2.1.1 Padronização da linguagem

Sem uma padronização *de facto* da linguagem C nos anos seguintes a sua criação, havia uma liberdade grande demais na implementação dos diversos compiladores que apareceram. A especificação conhecida por K&R C era, na ocasião, a única disponível.

De 1983 a 1989, o American National Standards Institute (ANSI) formou um grupo de trabalho para a padronização da linguagem, estabelecendo, ao final o padrão conhecido por C89 ou ANSI C. Esse mesmo padrão foi reeditado em 1990, com o rótulo C90, porém sem modificações relevantes na especificação.

Uma extensão foi incorporada à especificação em 1995, com novas definições e adições à biblioteca padrão. Este padrão foi chamado de C95.

[1]TIOBE: http://www.tiobe.com.

Modificações significativas na linguagem foram introduzidas no padrão C99, finalizado em 1999 e adotado a partir do ano 2000. Merecem destaque novos tipos de dados, como `long long` e `_Bool`, a incorporação de arranjos de comprimento definido durante a execução, a inclusão de novos cabeçalhos de bibliotecas, a possibilidade de comentários com `//`, a mistura de declarações e código e funções *inline*.

Em 2011 foi publicada a especificação C11, a qual provê suporte a caracteres Unicode, expressões com tipos genéricos (`_Generic`) e execução paralela multi-plataforma com `threads.h`.

O padrão atual para a linguagem C, no momento da escrita deste texto, é o C17, publicado em 2018. O C17 não acrescenta novos recursos à linguagem, porém corrige falhas na versão C11. A especificação C17 também é referenciada como C18.

Neste ano de 2023 é esperada a próxima edição do padrão da linguagem C, informalmente designado C23.

A especificação K&R C foi descrita em Ritchie et al. (1978); as demais especificações estão apropriadamente descritas em Wikipedia (2023).

2.2 Programa básico

Programar em C não é tão complexo, mas também não é tão natural. Muitos aspectos básicos da linguagem requereriam, em teoria, um conhecimento mais sólido sobre o que são variáveis, como estas existem na memória e como podem ser manipuladas.

Em um primeiro momento, estes detalhes serão ignorados, visto que é possível começar a programar e apenas aceitar que algumas coisas são assim mesmo. Para estes detalhes, por enquanto, basta pensar "não sei, só sei que foi assim"[2].

2.2.1 Primeiro programa: um programa mínimo (e inútil)

Para começar diferente, o primeiro programa exemplo não será o *Hello, world!*, amplamente utilizado na literatura e em tutoriais. Ele será o código mínimo na linguagem que é válido e coerente. Por ser mínimo, porém, não faz nada útil.

```
int main(void) {
    return 0;
}
```

Mesmo sendo minimalista, este código contém vários elementos interessantes:

- A função `main` e seu bloco de comandos
- O comando `return`

Um bloco de comandos é uma coleção de comandos delimitados por chaves. O nome `main` é associado a esse bloco de comandos, que contém apenas um comando simples (o `return 0;`) neste caso. Os tipos `int` e `void` indicados não são relevantes neste momento. Chamar o bloco de comandos de *main* (principal) é obrigatório, pois indica onde a execução do programa começa.

Neste exemplo, há um único comando dentro de `main`: `return 0;`. Este comando indica que, ao ser terminado, o programa devolve ao sistema operacional um valor inteiro, que é um indicador que

[2]Como diria Chicó, em *Auto da Compadecida*, peça de teatro de Ariano Suassuna também retratada em uma minissérie de televisão.

indica em que condições o programa encerrou sua execução. Por convenção, o valor zero significa que a execução se encerrou sem erros.

Ao ser compilado e executado, este programa apenas indica ao sistema operacional que terminou sem erros.

2.2.2 Segundo programa: *Hello, world!*

O segundo exemplo expande o código anterior, agora para apresentar uma mensagem na tela. Para isso, ele usa uma função chamada `printf`, responsável por apresentar uma saída formatada.

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

```
Hello, world!
```

Este programa, além do `printf`, também usa a linha `#include <stdio.h>`. O arquivo `stdio.h` é um arquivo de cabeçalho (`stdio` significa *standard input and output* e `.h` indica *header*) e contém informações sobre como o `printf` deve ser processado pelo compilador. Este arquivo de cabeçalho descreve uma série de funções para entradas e saídas feitas pelos programas.

O argumento do `printf` é o texto *Hello, world!*. Em C, textos são sempre especificados entre aspas duplas. No exemplo, `\n` é um indicador para mudar para a linha seguinte.

Um outro detalhe no código ainda pode ser destacado: cada comando simples é terminado com um ponto e vírgula, que é o caso tanto do `printf` quanto do `return`.

Na sequência é apresentada a versão definitiva do código do *Hello, world!*.

```c
/*
Programa "Hello, world": código exemplo do clássico primeiro programa em C
    que apenas apresenta uma mensagem de saudação na tela
Assegura: a apresentação da mensagem padrão "Hello, world"
*/
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

```
Hello, world!
```

Esta versão acrescenta a documentação do código. Qualquer texto colocado entre `/*` e `*/` é um comentário e é ignorado completamente pelo compilador ao analisar o código fonte. Os comentários não são para o compilador, mas para humanos que lerão o código do programa.

Neste caso específico, o comentário tem a função de documentar o propósito do código. Ele fornece uma descrição do propósito do código e dá informações relevantes. O grau de detalhe depende sempre do contexto; códigos simples podem ter documentação mais simples, enquanto programas que fazem parte de um projeto compartilhado entre vários desenvolvedores deve conter as informações necessárias para que todos da equipe os compreendam.

2.3 Comandos simples

Um componente estrutural da linguagem é o chamado comando simples. Esse comando de caracteriza por uma instrução individual no código do programa.

Os comandos simples seguem uma sintaxe também simples.

Comando simples

instrução ;

Nos exemplos dados, cada `printf` usado para apresentar uma informação e também o `return 0` são comandos simples e, desta forma, obrigatoriamente terminados com um ponto e vírgula. A execução do `printf` é uma *instrução*, assim como o término da execução indicado pelo `return`.

Segue um exemplo simples de programa com alguns comandos simples.

```
/*
Apresentação de uma série de mensagens na tela
*/
#include <stdio.h>

int main(void) {
    printf("Bom dia! ");
    printf("Este é um exemplo de ");
    printf("vários comandos simples.\n");
    printf("Todos são execuções do printf e todos são finalizados com ';'.\n");
    printf("O 'return 0', naturalmente, também é um comando simples.\n");

    return 0;
}
```

```
Bom dia! Este é um exemplo de vários comandos simples.
Todos são execuções do printf e todos são finalizados com ';'.
O 'return 0', naturalmente, também é um comando simples.
```

Dica

Embora a linguagem C não tenha objeções quando a escrever dois comandos simples em uma única linha do programa, sugere-se forntemente que cada comando tenha sua própria linha.

Curiosidade

Em C, existe a possibilidade de que o comando simples seja vazio. Para especificá-lo, bastar inserir o ponto e vírgula.

```
    printf("Um!");
    ;  // comando vazio que não faz nada...
    printf("Dois!");
```

Resta pensar quando um comando que não faz nada pode ser útil.

2.4 Código fonte, objeto e executável

Todos os programas em C são escritos em arquivos de texto simples, que são os que não dão suporte a negrito e itálico, tipos de fonte ou formatação de parágrafo, entre outros recursos. O código em C segue uma sintaxe bastante específica e os programas escritos são chamados de código fonte.

Para um programa escrito em C ser executado, primeiramente é necessário que ele seja compilado. O compilador é um programa cuja atribuição principal é analisar o código fonte, interpretando as letras, símbolos e dígitos ali escritos e gerar o código objeto, que é o código compilado. O código objeto é guardado em um arquivo e já não é mais legível por humanos, já que contém as instruções que o processador é capaz de entender e executar.

O código fonte, depois de compilado, leva a um código objeto que é dependente da máquina. Assim, são produzidos códigos objeto específicos para processadores x86, ARM ou outro. Para que um mesmo programa possa ser executado em plataformas diferentes, é preciso que ele seja compilado para cada plataforma alvo individualmente.

Há inúmeros compiladores disponíveis para C, como o GNU e o CLang, por exemplo. Também há implementações de compiladores específicos para cada plataforma de sistema operacional e e de processador. Ou seja, há um compilador GNU para Windows executando em processadores x86 e outro para Linux em um processador ARM. Cada compilador um possui suas próprias regras internas, mas todos obedecem especificações rigorosas e padronizadas. Desta forma, independente do hardware e do sistema operacional, um mesmo código fonte, ao ser compilado e executado, produz um mesmo resultado. Isso é uma regra e, para ela, naturalmente há exceções.

Neste texto, a especificação conhecida como C17 (da ISO) é a adotada para todas as implementações. Os programas são compilador usando o compilador GNU em um sistema operacional Linux, tudo sobre um hardware x86.

Finalmente, ainda há uma última etapa, na qual o código compilado é colocado em um arquivo que o sistema operacional da vez consegue carregar para a memória e colocá-lo em execução. Este é chamado de código executável e também depende da plataforma.

Usualmente a etapa do código objeto é escondida de quem usa o compilador e, aparentemente, do código fonte é gerado diretamente o executável. Na prática, esse é o efeito final.

2.4.1 Compilação com o `gcc`

Há uma diversidade de compiladores para a linguagem C, além de diversos IDEs[3] com diferentes facilidades para escrever códigos fonte. Há IDEs que podem ser instalados, como Visual Studio Code[4], Code::Blocks[5], Visual Studio[6] ou Eclipse[7], por exemplo, além das disponíveis *online*, como GDB[8], Programiz[9] ou Replit[10].

Neste livro os programas foram compilados invocando no terminal o compilador GNU GCC[11] na versão mais recente disponível no repositório oficial do Debian.

```
$ gcc --version
gcc (Debian 12.2.0-14) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

[3] Um ambiente de desenvolvimento integrado, ou *integrated development environment* (IDE), é um programa que dá suporte para escrever programas, provendo um editor de texto dedicado, destaque de sintaxe (diferentes elementos do programa aparecem em cores diferenciadas), acesso ao compilador com o clique de um botão ou uma combinação simples no teclado.
[4] Visual Studio Code: https://code.visualstudio.com.
[5] Code::Blocks: https://www.codeblocks.org.
[6] Visual Studio: https://visualstudio.microsoft.com.
[7] Eclipse: https://www.eclipse.org.
[8] GDB Online: https://www.onlinegdb.com.
[9] Programiz: https://www.programiz.com.
[10] Replit: https://replit.com.
[11] Gnu Compiler Collection: https://gcc.gnu.org.

As opções `-Wall` e `-pedantic` são incluídas automaticamente e ativam uma vasta gama de mensagens de erro e avisos importantes, dando maior controle sobre o código executável que está sendo gerado. A especificação C17 da linguagem C é adotada com a opção `-std=c17` (Seção 2.1.1).

```
alias gcc='gcc -Wall -pedantic -std=c17'
```

Para exemplificar, considere um arquivo com nome `bom_dia.c` com código fonte seguinte.

```c
/*
Apresentação um bom dia!
Assegura: uma mensagem de bom dia apresentada na tela
*/
#include <stdio.h>

int main(void) {
    printf("Bom dia! Que seu dia seja ótimo!\n");

    return 0;
}
```

Usando o comando `file` é possível ver (ou tentar ver) o tipo de informação guardada em um arquivo. Para o caso do código fonte, o arquivo é identificado como *C source code*, ou seja, código fonte em C.

```
$ file bom_dia.c
bom_dia.c: C source, Unicode text, UTF-8 text
```

A compilação pode ser feita com o comando apresentado. A opção `-o` permite nomear o arquivo executável que é criado.

```
$ gcc -Wall -pedantic -std=c17 bom_dia.c -o bom_dia
```

Esse comando compila `bom_dia.c` e gera o arquivo executável `bom_dia`. No caso de sucesso na compilação, o `gcc` não apresenta saídas; apenas eventuais problemas são apresentados na tela.

```
$ file bom_dia
bom_dia: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=734bc8eaaad06fcbf50bcbed07960614dd9a0077, for GNU/Linux 3.2.0,
not stripped
```

A execução de `file`, nesse caso, apresenta muita informação. A relevante, neste momento, é que o arquivo contém um *ELF 64-bit LSB pie executable*, o que quer dizer que é um programa executável.

Na sequência é apresentada a execução do programa `bom_dia`.

```
$ ./bom_dia
Bom dia! Que seu dia seja ótimo!
```

2.4.2 Erros de sintaxe

Sendo uma linguagem de programação, os comandos são sempre analisados de forma rígida. A falta de um parêntese, um espaço no lugar errado ou uma grafia errada (como `prinft`, por exemplo) já dá margem para o compilador reclamar que algo está errado e se recusar a gerar o executável. Esses são os erros de sintaxe e é função do compilador encontrá-los.

O programador deve aprender a ler as mensagens de erro produzidas pelo compilador e interpretá-las, permitindo a remoção dos erros sintáticos.

Como exemplo, segue uma versão incorreta do programa exemplo. Nela, "acidentalmente" o ponto e vírgula foi esquecido.

```
/*
Programa "Hello, world": código exemplo do clássico primeiro programa em C
    que apenas apresenta uma mensagem de saudação na tela
Assegura: a apresentação da mensagem padrão "Hello, world"
*/
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n")
    return 0;
}
```

```
main.c: In function ´main´:
main.c:9:30: error: expected ´;´ before ´return´
    9 |     printf("Hello, world!\n")
      |                              ^
      |                              ;
   10 |     return 0;
      |     ~~~~~~
```

A mensagem de erro aponta em que linha e coluna deveria haver um ponto e vírgula.

Os erros são apresentados conforme o compilador consegue detectar, apresentando mensagens tão boas quanto possível. Além disso, uma única falha pode desencadear uma sequência de erros, como no exemplo seguinte, na qual apenas faltou fechar as aspas no argumento da função `printf`.

```
/*
Programa "Hello, world": código exemplo do clássico primeiro programa em C
    que apenas apresenta uma mensagem de saudação na tela
Assegura: a apresentação da mensagem padrão "Hello, world"
*/
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n);
    return 0;
}
```

```
main.c: In function ´main´:
main.c:9:12: warning: missing terminating " character
    9 |     printf("Hello, world!\n);
      |            ^
main.c:9:12: error: missing terminating " character
    9 |     printf("Hello, world!\n);
      |            ^~~~~~~~~~~~~~~~~
main.c:10:5: error: expected expression before ´return´
   10 |     return 0;
      |     ^~~~~~
main.c:10:14: error: expected ´;´ before ´}´ token
   10 |     return 0;
      |              ^
      |              ;
   11 | }
      | ~
```

2.4.3 Erros de lógica

Um programa pode ser compilado e gerar o executável sem erros ou qualquer tipo de aviso que o compilador esteja ajustado para dar. Mas apesar disso, a execução não produz o resultado desejado.

Neste caso, mesmo com a sintaxe correta, as instruções contém um erro (uma falha de cálculo, uma comparação equivocada) que invalida o programa. Esse são os erros de lógica e, desta vez, cabe ao programador encontrá-los e corrigi-los.

A evitação de erros de lógica é considerada ao longo de todo o livro.

3 Tipos de dados da linguagem C

Qualquer linguagem de programação, dentre as milhares existentes, têm como premissa a manipulação e a transformação de dados. Esta parte do texto trata de como dados são representados em computação e faz uma apresentação dos principais tipos de dados usados na linguagem C.

A ilustração dos tipos de dados, suas limitações e principais características envolvem, adicionamente, a apresentação da função `printf`, responsável por apresentar os dados do programa para o usuário.

3.1 Representação de dados

Em computação, as informações são estruturadas em bits. A forma mais básica de circuitos eletrônicos representarem algo é pela ausência ou presença de corrente elétrica. Assim, um bit é uma parte do circuito que indica "desligado" ou "ligado", "sem corrente elétrica" ou "com corrente elétrica", "ausente" ou "presente" ou, simplificadamente, "zero" ou "um". O uso dos valores binários 0 e 1 é a forma tradicional de se representar um bit.

Para serem usados de forma prática, os bits são sempre organizados em sequências de comprimento oito. O grupo de oito bits é chamado de byte. Nos sistemas computacionais, os circuitos responsáveis por armazenar os bytes são chamados de memória. Esta é usualmente medida em gigabytes, sendo cada gigabyte correspondente a 1.073.741.824 bytes (2^{30}).

Cada byte, com seus oito bits, pode assumir os valores 00000000, 00000001, 00000010, 000000011... até 11111111. São 2^8 possíveis combinações.

Programas manipulam dados, como nomes de localidades, taxas de câmbio, velocidade de veículos, listas de espera e tantos outros. Para representar os dados, programas usam essa memória estruturada em bytes. Tanto as instruções que serão executadas pelo processador quanto os dados precisam ser representados com bytes.

Para representar um dado qualquer, portanto, é preciso fazer seu mapeamento para bytes.

A letra `A` é usualmente representada pelo byte 01000001 e o símbolo `!`, pelo 00100001. Essas escolhas foram arbitrárias e o mapeamento do conjunto básico de caracteres pode ser consultado em uma tabela chamada de *American Standard Code for Information Interchange*, ou tabela ASCII[1]. Textos são representados por sequências de bytes representando os caracteres.

Números inteiros são frequentemente representados por uma certa quantidade fixa de bytes consecutivos. Assim, considerando-se quatro bytes para um inteiro, é possível associar a sequência 00000000 00000000 00000000 00000000 ao valor zero, 00000000 00000000 00000000 00000001 ao valor 1 e assim, sucessivamente, até 11111111 11111111 11111111 11111111, que equivaleria a 4.294.967.295 (ou 2^{32}-1). Números negativos separam o primeiro bit para indicar o sinal (0 é positivo, 1 é negativo) e os demais 31 bits seriam usados para representar os diversos valores, indo de -2.147.483.648 a 2.147.483.647 considerando-se os quatro bytes. Naturalmente, quanto mais bytes, maior o intervalo de valores representados.

[1] Existem outras estratégias para representação de caracteres, principalmente para incorporar acentuações (como `ñ` do espanhol), caracteres particulares (`ß` do alemão) ou ainda caracteres como os japoneses e hebraicos.

Outros tipos de valores, como números reais, valores lógicos e endereços internos da memória também escolhem uma estratégia para representar seus valores usando um mapeamento adequado para um ou mais bytes.

3.2 Constantes e seus tipos

Tendo em vista que quaisquer dados (números, textos ou qualquer outro) usam bytes para serem representados, a linguagem C também designa uma codificação específica para os valores ao interpretar um código fonte.

Nos programas em C, valores explícitos no código, como um texto ou um valor numérico, é chamado de valor constante. O texto clássico expresso por `"Hello, world!"` é uma constante literal (textual). Em C, as constantes com sequências de caracteres são sempre expressas usando-se aspas duplas.

O código seguinte visa apresentar como saída o valor de π, que foi (grosseiramente) aproximando no código fonte para 3,1416.

```c
/*
Apresentação do valor aproximado de pi
*/
#include <stdio.h>

int main(void) {
    printf("pi vale, aproximadamente, %g\n", 3.1416);
    return 0;
}
```

```
pi vale, aproximadamente, 3.1416
```

O comando `printf` possui dois parâmetros. O primeiro é uma constante textual com o que deve ser apresentado (elemento obrigatório da função); o segundo é um valor real expresso na forma de uma constante (`3.1416`). O símbolo `%g` é um indicador de local usado para valores reais, ou seja ele representa onde o valor 3,1416 será inserido no texto.

A constante `3.1416` (expressa usando o ponto como separador decimal) possui, em C, o tipo `double`. Esse tipo corresponde a uma representação de precisão dupla, seguindo o padrão IEC 60559. Qualquer constante real em C é representada por um `double`, exceto quando explicitamente indicado de outra forma.

Constantes inteiras expressas em base 10 são representadas pelo tipo `int`, que tem mínimo de 16 bits, embora implementações atuais comumente usem 32 bits. Com o mínimo de bits, os valores representados vão de -32.768 a 32.767; já com 32 bits a variação é de -2.147.483.648 a 2.147.483.647. Desta forma, dependendo da implementação do compilador, os limites podem variar consideravelmente. As constantes inteiras seguem essa regra, exceto quando declaradas explicitamente com um tipo específico.

Caso uma constante inteira exceda a capacidade de representação de um `int`, um `long int` é usado em seu lugar, partindo de um mínimo de 32 bits. Essa escolha é transparente para o programador.

```c
/*
Apresentação do valor de 5234 elevado ao cubo
*/
#include <stdio.h>

int main(void) {
    printf("O cubo de %d é igual a %ld\n", 5234, 143384152904);
    return 0;
}
```

```
O cubo de 5234 é igual a 143384152904
```

Neste programa, o `printf` usa a especificação `%d`, significando um `int` apresentado na base 10 (decimal). A especificação `%ld` é usada para um `long int` expresso em decimal. Os valores são substituídos na ordem em que aparecem no `printf`.

Particularmente no compilador usado para produzir esse exemplo, o valor 5.234 pode ser representado em um `int` e escrito com `%d`; já 143.384.152.904 (5.234^3) necessitou de mais bits e foi automaticamente promovido para um `long int`, cuja especificação de formato é `%ld`. O compilador poderá apresentar um aviso caso o valor maior tente ser escrito apenas com `%d`, embora o executável seja gerado e o resultado apresentado seja incorreto.

Da mesma forma que há especificadores para valores numéricos, há também para valores textuais.

```c
/*
Apresentação de valores textuais
*/
#include <stdio.h>

int main(void) {
    printf("%s é primo de %s\n", "Nereu", "Eutália");
    printf("%s é prima de %s\n", "Eutália", "Nereu");
    printf("%s começa com a letra %c\n", "Yolanda", 'Y');

    return 0;
}
```

```
Nereu é primo de Eutália
Eutália é prima de Nereu
Yolanda começa com a letra Y
```

A indicação `%s` é compatível com textos de comprimento variado, ou seja, cadeias de caracteres. Por outro lado, C define um tipo específico para um único caractere: o tipo `char`. Ele usa oito bits (mínimo), ou seja, um byte, e é representado por aspas simples: `'Y'`. Para caracteres simples do tipo `char`, usa-se a especificação de formato `%c`.

A diferença entre `"Y"` e `'Y'` é abordada em mais detalhes no Capítulo 16.

3.3 Mais sobre o `printf`

O nome `printf` vem de *print formatted*, ou seja, apresente algo de forma formatada. O uso desta função requer o carregamento do arquivo de cabeçalho `stdio.h`.

A função apresenta um texto na saída padrão (a tela do terminal) e este deve ser o primeiro parâmetro. Além de apresentar o texto, a função também faz conversões de valores, usando os marcadores iniciados com `%`. Por exemplo, `%d` é usado para um valor inteiro decimal, `%g` para apresentar um valor real e `%s` é usado para apresentar valores textuais, chamados cadeias de caracteres ou *strings*. Outro elemento de formatação são os caracteres especiais como `\n`, que indica a mudança de linha, `\t`, que é uma tabulação similar à dos editores de texto, ou ainda `\"`, que indica aspas duplas.

Cada marcador com o símbolo `%` é substituído em ordem. Assim, se houver várias substituições, cada uma delas é feita sucessivamente.

```c
/*
Exemplo de escrita de vários valores
*/
#include <stdio.h>

int main(void) {
    printf("Meu nome é %s, nasci em %d e ganho R$ %g de salário.\n",
        "Fulano", 2003, 2512.17);

    return 0;
}
```

```
Meu nome é Fulano, nasci em 2003 e ganho R$ 2512.17 de salário.
```

Para valores inteiros, segue um exemplo para a escrita do valor de uma expressão usando variações de formato. Algumas opções diferentes de escrita são acrescentadas.

```c
/*
Exemplo de escrita de valores inteiros
*/
#include <stdio.h>

int main(void) {
    printf("Um valor inteiro    : %d\n", 481);
    printf("Outro valor inteiro : %8d\n", 481);
    printf("E mais outro        : %08d\n", 481);

    return 0;
}
```

```
Um valor inteiro    : 481
Outro valor inteiro :      481
E mais outro        : 00000481
```

Quando um número inteiro é inserido entre o `%` e o `d`, como em `%8d`, e reservado um espaço fixo para o inteiro, que é formatado à direita. No exemplo, como o valor escrito possui três dígitos, antes dele são acrescentados cinco espaços para, no total, preencher as oito posições especificadas. No caso de `%08d`, o dígito `0` indica que os espaços faltantes devem ser preenchidos com o dígito zero.

No caso de valores reais, o formato `%g` é uma representação que busca a "melhor" forma de se apresentar um valor, seguindo um algoritmo interno.

```c
/*
Exemplo de escrita de valores reais com %g
*/
#include <stdio.h>

int main(void) {
    printf("Um valor real    : %g\n", 23.0);
    printf("Outro valor real : %g\n", 163778837773.32998827);
    printf("E mais outro     : %g\n", 3.3/1234567.8);
    printf("E um último      : %g\n", 1.23432624324);

    return 0;
}
```

```
Um valor real    : 23
Outro valor real : 1.63779e+11
E mais outro     : 2.673e-06
E um último      : 1.23433
```

Neste programa, o valor 23,0, como não possui casas decimais, é mostrado como se fosse um valor inteiro. Os outros dois valores usam a notação científica para deixar a escrita mais concisa, lembrando

que `1.63779e+11` equivale a $1{,}63779 \times 10^{11}$ e `2.673e-07`, a $2{,}673 \times 10^{-7}$. O último exemplo mostra um arredondamento do valor escrito.

É possível acrescentar ao `%g` o comprimento total que o valor deve ter (`%10g` para 10 espaços), o número de casas decimais que serão apresentadas (`%.5g` para cinco decimais), além de outras opções.

Além da especificação `%g`, é possível usar `%f` (nunca usa notação científica) ou `%e` (sempre notação científica) para valores reais.

Finalmente, ainda há especificações para apresentação de endereços de memória (`%p`) e valores inteiros nas bases hexadecimal (`%x`) e octal (`%o`).

```
/*
Exemplo de escrita alguns formatos de apresentação
*/
#include <stdio.h>

int main(void) {
    printf("O decimal %d pode ser escrito %x (hexadecimal) ou %o (octal).\n",
        125, 125, 125);
    printf("Este é o endereço nulo: %p\n", NULL);

    return 0;
}
```

```
O decimal 125 pode ser escrito 7d (hexadecimal) ou 175 (octal).
Este é o endereço nulo: (nil)
```

3.4 Constantes com tipo explícito

As constantes expressas em C possuem tipos automáticos ligados à ela, como apresentado na Seção 3.2. A linguagem, porém, permite escrever um valor constante e associar a ele um tipo específico.

```
/*
Exemplos de constantes com tipo explícito
*/
#include <stdio.h>

int main(void) {
    printf("Este %d é int, mas este %ld é long int\n", 10, 10L);
    printf("Este %u é um unsigned int\n", 10U);

    printf("A diferença entre %g (precisão dupla) e %g (simples) é %g\n",
        1e-7, 1e-7f, 1e-7 - 1e-7f);

    return 0;
}
```

```
Este 10 é int, mas este 10 é long int
Este 10 é um unsigned int
A diferença entre 1e-07 (precisão dupla) e 1e-07 (simples) é -1.16861e-15
```

3.5 Hexadecimal e octal

As constantes inteiras são, usualmente, escritas usando a base 10. A linguagem C permite, além desta, escrever constantes na base oito (octal) e 16 (hexadecimal). O uso dessas bases é prático quando quando a linguagem é usada para manipulação em nível baixo, ou seja, no nível dos bits.

```c
/*
Exemplos de constantes em hexadecimal e em octal
Constante hexadecimais se iniciam com 0x
Valores em octal são expressos iniciando o número com zero
*/
#include <stdio.h>

int main(void) {
    printf("Três maneiras de escrever %d: %d e %d\n", 10, 0xA, 012);
    printf("Três maneiras de escrever %x: %x e %x\n", 10, 0xA, 012);
    printf("Três maneiras de escrever %o: %o e %o\n", 10, 0xA, 012);

    return 0;
}
```

```
Três maneiras de escrever 10: 10 e 10
Três maneiras de escrever a: a e a
Três maneiras de escrever 12: 12 e 12
```

4 Variáveis e leituras em C

Um programa em C lida com dados, que podem ser de diferentes tipos, como `int`, `long int` ou `double`, por exemplo. Neste capítulo são apresentados onde os dados são armazenados nos programas (variáveis), seus nomes (identificadores) e como guardar e substituir os valores desses dados (atribuição e leitura).

4.1 Variáveis, declarações e atribuição

Conforme abordado na Seção 3.1, qualquer informação precisa ser mapeada para bytes para poder ser utilizada em um sistema computacional.

Para que um programa faça sua tarefa de resolver um dado problema, é preciso que ele tenha acesso à memória e sua representação. Quase na totalidade das linguagens de programação, uma área da memória na qual está guardado um dado é referenciado por um nome arbitrário. Por exemplo, se o ano de um evento é um dado que precisa ser guardado, os bytes reservados para o armazenamento dessa informação é referenciado por um identificador. E dado que o conteúdo da memória possa eventualmente ser modificado, a esse armazenamento é dado o nome de variável, no sentido de mutável.

Assim, uma variável corresponde a uma área da memória principal e os bytes que a compõe são referenciados por um identificador.

O programa seguinte exemplifica o uso de duas variáveis para o armazenamento de valores de temperatura.

```c
/*
Conversão de escalas termométricas, de graus Celsius para Fahrenheit
*/
#include <stdio.h>

int main(void) {
    double celsius = 25.5;
    double fahrenheit = 1.8 * celsius + 32;

    printf("%g graus Celsius = %g Fahrenheit\n", celsius, fahrenheit);

    return 0;
}
```

```
25.5 graus Celsius = 77.9 Fahrenheit
```

A variável cujo identificador é `celsius` armazena valores reais usando o tipo `double`. Ela representa alguns bytes na memória principal na qual o valor de 25,5ºC é representado. Uma segunda variável do tipo `double` também é usada para armazenar outro valor real. Neste caso, o valor armazenado é um cálculo que envolve a primeira variável e equivale à conversão da escala Celsius para Fahrenheit (notando que o operador `*` denota a multiplicação). O identificador desta segunda variável é `fahrenheit`. Ambos os nomes (identificadores) foram escolhidos pelo programador à sua conveniência.

Há pontos importantes neste programa:

- Há a declaração de duas variáveis;
- Existem valores atribuídos a ambas;

- O valor armazenado nas variáveis é consultado.

A primeira variável, `celsius`, é declarada precedendo-se seu identificador por seu tipo, que, no caso, é `double`. Valores reais, via de regra, devem usar o tipo `double` como tipo para a representação e armazenamento.

```
double celsius = 25.5;
```

O sinal de igual é o operador de atribuição. Ele indica que o valor à direita (o valor `25.5`, que é um `double`) será armazenado na área de memória reservada para a variável. Assim, a variável é criada (declaração) e tem um valor atribuído a ela (com o `=`) na mesma linha de código. Estas duas ações são finalizadas por um ponto e vírgula.

O mesmo ocorre com a segunda variável, que é declarada e a ela é atribuído um valor resultante de uma expressão que envolve uma multiplicação (`*`) e uma soma (`+`).

```
double fahrenheit = 1.8 * celsius + 32;
```

A diferença aqui é que o valor atribuído à variável, isto é, armazenado nela, é o resultado de uma expressão aritmética cujo objetivo é a conversão entre as unidades de temperatura. Outra diferença é que a expressão usa o valor de `celsius`, ou seja, ela usa o conteúdo armazenado nessa variável específica.

Deste modo, é importante salientar que, por meio do identificador de uma variável, um conteúdo pode ser armazenado nela e também esse valor pode ser consultado para uso.

Por fim, vale ainda comentar que na função `printf` os valores armazenados nas duas variáveis são novamente consultados para serem convertidos para uma representação textual e apresentados.

4.2 Identificadores

Nas linguagens de programação, muitos de seus elementos possuem nomes, chamados de identificadores. O nome dado a uma variável é seu identificador; `main` é o identificador da função por onde o programa em C começa sua execução.

Um identificador é uma sequência de caracteres usada como nome. Existem palavras reservadas na linguagem, como `void`, `return`, `int` e `double`, entre outras, que não podem ser usadas como identificadores. Até agora, foram usados nos exemplos identificadores para variáveis (`celsius`) e funções (`printf`).

Um identificador válido na linguagem C é formado exclusivamente por letras, dígitos e pela sublinha (`_`), nunca se iniciando com um dígito. A Tabela 4.1 apresenta exemplos.

As letras podem ser maiúsculas ou minúsculas, como `valor`, `Idade` e `ponto_A`, por exemplo. O jargão da computação para referenciar maiúsculas e minúsculas é caso. A linguagem C tem identificadores sensíveis ao caso: `total`, `Total` e `TOTAL` são identificadores distintos e podem coexistir.

A escolha dos identificadores é responsabilidade do programador.

Tabela 4.1: Identificadores e sua validade.

Tabela 4.2: Identificadores válidos.

Identificador	Validade
`nome`	sim
`idade`	sim
`salario_medio`	sim
`prefixo1`	sim
`prefixo2`	sim
`massa_kg`	sim
`a1b2c3d4__`	sim
`__estado__`	sim

Tabela 4.3: Identificadores inválidos.

Identificador	Validade
`2pi`	não (inicia com dígito)
`salario-medio`	não (hífen presente)
`km/h`	não (barra presente)
`massa total`	não (espaço presente)
`nota.geral`	não (ponto presente)
`total:geral`	não (dois pontos presente)
`valor~`	não (til presente)
`montante_r$`	não (cifrão presente)

4.2.1 Estilo

Nos programas apresentados neste livro, todos os identificadores de variáveis e de funções são escritos em *snake case*, que é um estilo de escrita. No *snake case*, todas as letras usadas são minúsculas e, quando um identificador é composto de duas ou mais palavras, é usada a sublinha para separá-las.

Essa regra é seguida mesmo quando os nomes possuem sua combinação de maiúsculas e minúsculas característicos. Desta forma, o armazenamento do CPF usará uma variável `cpf`, um cálculo envolvendo o pH usará `ph` ou a temperatura em Fahrenheit usará `fahrenheit`.

Esse padrão não é necessariamente adotado pelas bibliotecas da linguagem, que usam abreviações e combinações de estilo próprias e independentes. Como exemplos, não é usado `print_formatted`, mas `printf`, e na manipulação de caracteres há uma função chamada `strcat`, que significa *string concatenation* (sim, *concatenation* é abreviada para *cat*).

Dica

A aderência a um padrão no formato dos identificadores, qualquer que ela seja, é muito importante para códigos claros e compreensíveis. Uma vez escolhido um padrão, este deve ser mantido constante em todo o código.

Os estilos usados nos programas implementados neste material estão descritos no **?@sec-guia-de-estilo**.

Dica

A versões mais atuais das especificações para a linguagem C permitem o uso de caracteres acentuados nos identificadores. Essa é uma prática de uso raro, porém.

```c
/*
Aumento de salário
*/
#include <stdio.h>

int main(void) {
    double salário_atual = 3500.00;
    double porcentagem_aumento = 0.15;  // 15%

    printf("Salário anterior: %.2f.\n", salário_atual);

    salário_atual = salário_atual * (1 + porcentagem_aumento);
    printf("Salário novo: %.2f.\n", salário_atual);

    return 0;
}
```

```
Salário anterior: 3500.00.
Salário novo: 4025.00.
```

Fica registrada uma recomendação de que somente caracteres ASCII simples sejam usados para os identificadores. A manutenção do código por terceiros, por exemplo, pode se tornar complicada se outros programadores, com configurações de teclado diferentes, simplesmente não conseguirem digitar o nome de uma variável, como um código escrito em tcheco com uma variável chamada `stáří` (idade).

4.2.2 Identificadores significativos

Em programas bem escritos a clareza é importante. O compilador não se importa com o nome escolhido para uma variável; essa escolha é para os humanos que leem o código fonte. A escolha de bons nomes ajuda a entender melhor o que o comandos fazem e, em consequência, permitem a identificação de erros, a correção dessas falhas e a incorporação de novas funcionalidades.

A regra básica para escolher um nome de variável é deixar claro o que ela contém. A opção por um ou outro nome depende bastante do contexto e é nesse contexto que deve haver clareza. Como um exemplo, uma variável chamada `nível` pode conter um valor numérico correspondente ao nível de um reservatório ou então ser uma valor textual com valores esperados `"fácil"`, `"médio"` ou `"difícil"`.

Há uma tendência natural (e bem comum) de associar as variáveis dos programas às variáveis da matemática, o que leva a escolha de variáveis com identificadores genéricos e não significativos, como `x`, `t` ou `a`. Em geral, se uma variável possui uma única letra, essa escolha não é uma boa opção. Há, porém, exceções.

Dica

Se um programador precisar explicar, de alguma forma, o que uma variável contém, é porque o identificador dela foi mal escolhido.

Ao longo do livro, os programas usam variáveis com nomes significativos. Muitas vezes os nomes são longos, o que pode levar o programador a ter preguiça de digitá-los. Felizmente, os IDEs modernos possuem recursos de auto-completar as digitações, que eliminam essa dificuldade.

Um problema de identificadores muito longos é que as linhas de código também ficam muito longas. Abreviações nos nomes podem ser empregadas, porém de forma criteriosa. Se `temperatura` é uma escolha clara para guardar um valor de temperatura, `temp` também pode ser. Porém `temp` é uma abreviação comum para um valor temporário e, em um contexto de temperaturas, não deve ser empregado.

Dica

A decisão do comprimento de um identificador envolve clareza do código e a facilidade de visualização do código fonte por um humano. O programador deve balancear esses e quaisquer outros aspectos, sempre com o objetivo de tornar o código o mais inteligível possível.

Uma amostra de um código com identificadores pobremente escolhidos é apresentando na sequência. A ausência de documentação é proposital neste caso.

```
#include <stdio.h>

int main(void) {
    int i = 57;
    int a = 1967;

    int e = a + i;
    printf("%d ou %d\n", e, e + 1);

    return 0;
}
```

```
2024 ou 2025
```

Segue agora uma nova versão do mesmo código, para o qual o compilador gera resultados idênticos (e provavelmente códigos executáveis iguais também).

```
#include <stdio.h>

int main(void) {
    int idade = 10;
    int ano_nascimento = 2013;

    int estimativa_ano_atual = ano_nascimento + idade;
    printf("%d ou %d\n", estimativa_ano_atual, estimativa_ano_atual + 1);

    return 0;
}
```

```
2023 ou 2024
```

Há claramente uma maior compreensão do propósito do programa nesta segunda versão. Nomes significativos ajudam a entender melhor o contexto como um todo.

Segue, para fins didáticos, a versão final do código.

```
/*
Estimativa do ano atual dadas a idade e o ano de nascimento de uma pessoa.
Duas estimativas são feitas, pois o ano corrente depende se a pessoa fez ou
não aniversário nesse ano.
*/
#include <stdio.h>

int main(void) {
    int idade = 10;
    int ano_nascimento = 2013;

    int estimativa_ano_atual = ano_nascimento + idade;
    printf("%d ou %d\n", estimativa_ano_atual, estimativa_ano_atual + 1);

    return 0;
}
```

```
2023 ou 2024
```

4.3 Mais sobre declarações de variáveis

Na prática, a atribuição a uma variável não é sempre necessária quando uma declaração é feita e, portanto, pode ser suprimida. Uma declaração simples de uma variável pode ser feita como se segue.

```c
/*
Escrevendo o valor de pi com cinco casas decimais
*/
#include <stdio.h>

int main(void) {
    double pi;

    pi = 3.141592654;
    printf("pi = %.5f\n", pi);

    return 0;
}
```

```
pi = 3.14159
```

A variável `pi` é criada com valor indefinido, mas antes de ser usada no `printf` tem um valor apropriado atribuído a ela. Sem a atribuição, o conteúdo da variável é considerado indeterminado e, portanto, não deve ser usado antes de se garantir uma atribuição prévia[1].

A declaração segue o formato seguinte.

> **Declaração de variáveis**
>
> *especificação_tipo lista_especificação_identificador* ;

O tipo base da variável, a *especificação_tipo*, é o primeiro elemento de uma declaração e é indicado por um tipo já existentes, como `int`, `long int` ou `double`, por exemplo. A *lista_especificação_identificador* é uma relação de especificações de identificador separados por vírgulas. O ponto e vírgula é obrigatório para indicar o término de uma declaração.

O exemplo seguinte apresenta declarações de variáveis simples, sem atribuição conjugada.

```c
int idade;
int ano;
```

De forma equivalente, essa declaração poderia ser expressa como se segue.

```c
int idade, ano;
```

Qualquer uma das duas formas podem ser usadas.

Dica

Uma fonte de erro comum é o uso do valor de uma variável para a qual nenhuma atribuição ainda foi feita, pois seu conteúdo não pode ser previsto.
O programa seguinte exemplifica esse problema.

[1] Mais detalhes sobre valores iniciais de variáveis são apresentados no Capítulo 19.

```c
/*
Exemplo de uso de uma variável sem valor previamente atribuído
*/
#include <stdio.h>

int main(void) {
    double valor;
    printf("valor: %g\n", valor);

    return 0;
}
```

```
valor: 6.95119e-310
```

O valor que é apresentado é efetivamente o conteúdo da variável dado pelos bytes que estão na memória. A saída produzida pelo programa é imprevisível.

Na declaração, segundo conveniência, cada variável declarada pode ter sua própria atribuição.

```c
int dia = 7, mes = 9, ano = 1822;
double salario_inicial = 4321.12, salario_final;
```

Neste exemplo, cada uma das três variáveis `int` são declaradas já com valores iniciais. Para as variáveis `double`, apenas `salario_inicial` possui atribuição, enquanto `salario_final` permanece sem iniciação.

Nos programas em C, todas as variáveis que forem usadas precisam ser declaradas.

4.4 Mais sobre atribuições

A atribuição de um valor a uma variável utiliza a sintaxe seguinte.

> **Atribuição**
>
> *expressão_esquerda* = *expressão_direita*

Na atribuição, *expressão_esquerda* indica onde será armazenado o valor resultante de *expressão_direita*. Para realizar essa operação, inicialmente a *expressão_direita* é completamente avaliada e, obtido o valor resultante, a *expressão_esquerda* é considerada para indicar o local de armazenamento na memória. Em geral, *expressão_esquerda* é somente um identificador de uma variável, enquanto *expressão_direita* pode ser qualquer expressão cujo resultado tenha tipo compatível.

No exemplo seguinte, para cada das duas atribuições, tanto a variável que é usada como local de armazenamento quanto o valor final da expressão que é atribuído a ela são do tipo `double`.

```c
double valor_cheio, valor_reduzido;

valor_cheio = 417.8;
valor_reduzido = valor_cheio / 3;
```

Ambas as atribuições são caracterizadas como comandos simples e, assim, devem ser terminadas com pontos e vírgulas.

Dica

A atribuição de um valor a uma variável somente deve ser feita se ele for essencial. Uma atribuição desnecessária ou irrelevante pode prejudicar o entendimento do código.

```
/*
Apresentação de um cálculo simples
*/
#include <stdio.h>

int main(void) {
    double fator1 = 1.2;
    double fator2 = 4.8;
    double resultado = 0;  // atribuição irrelevante

    resultado = fator1 * fator2;
    printf("%g * %g = %g\n", fator1, fator2, resultado);

    return 0;
}
```

```
1.2 * 4.8 = 5.76
```

Nesse programa, a atribuição de zero para `resultado` é irrelevante, pois esse valor será substituído logo na sequência. É curioso notar que poderia ser `resultado = -1.2e14` e o programa funcionaria normalmente.
Neste caso, apenas a declaração simples da variável deve ser feita.

4.5 Leitura para programas interativos

Até o momento, a manipulação de variáveis foi exemplificada apenas com atribuições diretas, o que torna o programa demasiadamente restrito. Por exemplo, no exemplo de conversão de Celsius para Fahrenheit, o programa não precisaria fazer as contas nem usar variáveis, pois como tudo é fixo, bastaria conter o comando seguinte e o resultado seria precisamente o mesmo.

```
printf("25.5 graus Celsius = 77.9 Fahrenheit\n");
```

Tornar os programas mais úteis, então, requer escrever códigos mais gerais, como para converter qualquer temperatura em graus Celsius para Fahrenheit. Para isso, o programa precisa obter qual o valor que deve ser convertido e isso não pode ser feito por uma atribuição.

Quando um programa obtém um dado externo ao código, esse processo é chamado de leitura. Desta forma, um programa geralmente faz a leitura de dados, realiza um processamento com eles e escreve os resultados.

A leitura do que o usuário digita em um terminal usualmente opera em duas etapas. Na primeira, o usuário digita seu texto (podendo eventualmente apagar um erro de digitação) e, quando tiver terminado, ele pressiona a tecla `ENTER`. Aí se inicia a segunda etapa, que consiste em repassar para o programa todos os caracteres digitados, incluindo a mudança de linha (`\n`) produzida pelo `ENTER`.

4.5.1 Leitura conteúdo textual com `fgets`

No programa seguinte, um nome é solicitado pelo programa e, em seguida, apresentado de volta juntamente com uma saudação.

```
/*
Saudação
*/
#include <stdio.h>

int main(void) {
    printf("Digite seu nome: ");

    char nome[80];
    fgets(nome, sizeof nome, stdin);

    printf("Olá, %s!\n", nome);

    return 0;
}
```

```
Digite seu nome: Alfonso Cardoso
Olá, Alfonso Cardoso
!
```

Uma variável *string* é usada para o armazenamento de cadeias de caracteres. Em C, o tipo básico `char` é usado para indicar uma único caractere; porém, como textos são sequências de caracteres (letras, dígitos, pontuações), os colchetes colocados depois do identificador `nome` indicam o comprimento máximo de caracteres que a variável suporta. No caso, o programa pode armazenar até 80 caracteres, o que é suficiente para um nome.

A função `fgets` (`stdio.h`) copia o que o usuário digitou no terminal, byte a byte, para a cadeia de caracteres. Esta função possui três parâmetros: o primeiro é para onde os dados digitados serão copiados (variável `nome`), o segundo é o comprimento da memória disponível para copiar (`sizeof nome`) e, por final, o último que é `stdin`, que é o fluxo de bytes vindo do teclado.

Na execução do programa anterior, é possível notar que a exclamação é apresentada na linha de baixo, logo depois do nome. A razão para isso é que o `ENTER` também é passado ao programa. Assim, para que se obtenha apenas o nome, é preciso remover esse `\n`. Essa ação é feita substituindo-se a mudança de linha por um caractere nulo (`\0`).

```
/*
Saudação
*/
#include <stdio.h>
#include <string.h>  // para strlen

int main(void) {
    printf("Digite seu nome: ");

    char nome[80];
    fgets(nome, sizeof nome, stdin);
    nome[strlen(nome) - 1] = '\0';  // sobrepõe '\0' ao '\n'

    printf("Olá, %s!\n", nome);

    return 0;
}
```

```
Digite seu nome: Alfonso Cardoso
Olá, Alfonso Cardoso!
```

O comando `nome[strlen(nome) - 1] = '\0'` determina a posição do `\n` usando o comprimento do texto digitado dado por `strlen` e atribui ali o `\0`. É importante notar que `'\0'` é escrito usando aspas simples, pois é um único caractere. Esses aspectos são abordados em mais detalhes no Capítulo 16.

4.5.2 Leitura de valores numéricos

Toda digitação provida pelo usuário e passada ao programa é textual. Como exemplo, se o usuário digita 10 como entrada, o programa recebe os caracteres `1`, `0` e o `ENTER`, ou seja, `"10\n"`. Para transformar essa sequência de caracteres em um `int`, por exemplo, é preciso convertê-la.

A função `sscanf` pode ser usada para diversas conversões, pois ela analisa os caracteres e os interpreta adequadamente.

Para exemplificar a leitura de dados numéricos, considere o problema de estimar qual é o ano atual baseado na idade de uma pessoa e de seu ano de nascimento. O cálculo é simples, apesar da resposta depender se a pessoa já fez ou não aniversário no ano atual. Desta forma, o Algoritmo 4.1 dá os dois possíveis resultados.

Algoritmo 4.1: Determinação do ano atual baseado na idade e no ano de nascimento de uma pessoa.

Descrição: Determinação do ano atual com base na idade e do ano de nascimento de uma pessoa
Requer: A idade e o ano de nascimento de uma pessoa
Assegura: As duas possibilidades do ano corrente, considerando se a pessoa já fez ou não aniversário

Obtenha *idade* e *ano_nascimento*
Calcule *ano_estimado* como $idade + ano_nascimento$
Apresente *ano_estimado* e $ano_estimado + 1$ como possibilidades

```c
/*
Determinação do ano atual com base na idade e do ano de nascimento de uma
    pessoa
Requer: A idade e o ano de nascimento de uma pessoa
Assegura: As duas possibilidades do ano corrente, considerando se a
    pessoa já fez ou não aniversário
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Qual sua idade? ");
    fgets(entrada, sizeof entrada, stdin);
    int idade;
    sscanf(entrada, "%d", &idade);

    printf("Que ano você nasceu? ");
    fgets(entrada, sizeof entrada, stdin);
    int ano_nascimento;
    sscanf(entrada, "%d", &ano_nascimento);

    int estimativa_ano_atual = ano_nascimento + idade;
    printf("Se você já fez aniversário este ano, estamos em %d.\n",
        estimativa_ano_atual);
    printf("Se não, o ano é %d.\n", estimativa_ano_atual + 1);
    printf("Bem, este é meu chute...\n");

    return 0;
}
```

```
Qual sua idade? 20
Que ano você nasceu? 2003
Se você já fez aniversário este ano, estamos em 2023.
Se não, o ano é 2024.
Bem, este é meu chute...
```

A linha que merece atenção neste programa é a que segue.

```
sscanf(entrada, "%d", &idade);
```

A função `sscanf` faz uma varredura na variável `entrada` (seu primeiro parâmetro), buscando um valor inteiro escrito em decimal (`%d`). Se achar o valor, faz a interpretação adequada e coloca o valor na variável inteira `idade`. É importante neste caso que a função precisa saber onde a variável está na memória e, assim, o operador `&`, que significa algo como "o local onde está", é obrigatório. A linha de comando pode ser, então, lida da seguinte forma: "procure no texto contido em `entrada` um valor no formato `%d` e o armazene na memória onde está a variável `idade`".

Para o ano de nascimento o procedimento é exatamente igual e a variável entrada é reaproveitada para fazer a segunda leitura.

Uma observação relevante é que, na interpretação da linha pela busca do valor inteiro, o `sscanf` ignora qualquer texto em branco antes dos dígitos numéricos esperados, como espaços e tabulações. Ele também encerra a interpretação ao encontrar qualquer coisa que não seja compatível com o tipo buscado e, desta forma, o `\n` no final de `entrada` é automaticamente ignorado.

Segue novo exemplo, com leituras simples, agora usando valores reais armazenados em variáveis `double`.

```
/*
Leitura de variáveis do tipo double
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite um valor real: ");
    fgets(entrada, sizeof entrada, stdin);
    double valor1;
    sscanf(entrada, "%lf", &valor1);

    printf("Digite outro valor real: ");
    fgets(entrada, sizeof entrada, stdin);
    double valor2;
    sscanf(entrada, "%lf", &valor2);

    printf("%g + %g = %g\n", valor1, valor2, valor1 + valor2);

    return 0;
}
```

```
Digite um valor real: 872.2
Digite outro valor real: 1.03e5
872.2 + 103000 = 103872
```

Valores do tipo `double` usam a especificação de formato `%lf` (`%g` só é usado no `printf`) e a interpretação é feita agora pela busca de qualquer combinação que possa ser interpretada como um valor real válido, incluindo a notação científica usada no exemplo.

4.5.3 Leitura de um único caractere

A função `fgets`, por si só, obtém o texto digitado no terminal. O filtro para que apenas o primeiro caractere seja capturado em uma variável do tipo `char` pode ser feito também com o `sscanf` usando-se o indicador de formato `%c`.

```
/*
Leitura de um valor em uma variável do tipo char com sscanf
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite um caractere: ");
    fgets(entrada, sizeof entrada, stdin);

    char caractere;
    sscanf(entrada, "%c", &caractere);
    printf("O caractere digitado foi o %c\n", caractere);
    printf("Seu código hexadecimal ASCII é %X\n", caractere);

    return 0;
}
```

```
Digite um caractere: M
O caractere digitado foi o M
Seu código hexadecimal ASCII é 4D
```

No exemplo, na variável `entrada` é armazenada a sequência `M\n`, ou seja o `M` digitado e o `ENTER` usado para enviar a linha ao programa. Com o `%c` do `sscanf`, somente o primeiro caractere da entrada é considerado, ignorando-se tudo o que existe depois dele. Na prática, depois do `M` do exemplo poderiam vir quaisquer outros caracteres e somente o primeiro é extraído de `entrada`.

Para este exemplo em particular, há uma forma mais simples e direta de obter o primeiro caractere do que o usuário digitou. Isso é feito explicitamente selecionando o primeiro caractere da variável: `entrada[0]`.

```
/*
Leitura de um valor em uma variável do tipo char usando indexação da cadeia
de entrada
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite um caractere: ");
    fgets(entrada, sizeof entrada, stdin);

    char caractere = entrada[0];
    printf("O caractere digitado foi o %c\n", caractere);
    printf("Seu código hexadecimal ASCII é %X\n", caractere);

    return 0;
}
```

```
Digite um caractere: m
O caractere digitado foi o m
Seu código hexadecimal ASCII é 6D
```

Esta última versão é, na opinião do autor, mais simples e direta, superando a leitura a obtenção do caractere com o `sscanf`.

4.5.4 Várias leituras em uma única linha

É bastante comum, em programas que processam dados, que uma linha possa conter mais que um valor. Desta forma, é preciso indicar ao `sscanf` para varrer a cadeia de entrada por mais que um valor.

O programa seguinte implementa o Algoritmo 4.2 e mostra a leitura de coordenadas em $\mathbb{R}^2$. O programa solicita os valores para x e para y, mas ambos devem ser digitados na mesma linha. Como resultado, o programa apresenta a distância desse ponto à origem do sistema de coordenadas.

Algoritmo 4.2: Distância de um ponto (x, y) à origem

Descrição: Cálculo da distância de um ponto em $\mathbb{R}^2$ à origem.
Requer: valores para x e y
Assegura: a distância de (x, y) à origem

Obtenha x e y
Calcule *distância* como $\sqrt{x^2 + y^2}$
Apresente *distância*

```
/*
Cálculo e apresentação da distância de um ponto em R^2 à origem, tendo como
    entrada os valores das coordenadas x e y desse ponto
Requer: x e y
Assegura: a distância de (x, y) à (0, 0)
*/
#include <stdio.h>
#include <math.h>  // para sqrt (raiz quadrada)

int main(void) {
    char entrada[160];

    printf("Digite os valores de x e y: ");
    fgets(entrada, sizeof entrada, stdin);

    double x, y;
    sscanf(entrada, "%lf%lf", &x, &y);

    double distancia_origem = sqrt(x * x + y * y);
    printf("A distância de (%g, %g) a (0, 0) é %g.\n", x, y, distancia_origem);

    return 0;
}
```

```
Digite os valores de x e y: 3.2 -1.8
A distância de (3.2, -1.8) a (0, 0) é 3.67151.
```

Primeiramente é relevante destacar o uso da função `sqrt` (*square root*) para o cálculo da raiz quadrada, a qual está especificada no arquivo de cabeçalho `math.h`, que deve ser incluído no preâmbulo do código fonte. Para o cálculo do quadrado foi usado o "truque" elementar que $x^2 = x \cdot x$. Além disso, como biblioteca de funções matemáticas não é automaticamente incluída durante a compilação, deve ser acrescentada a opção `-lm` (i.e., faça a ligação, *link*, com a biblioteca matemática `m`) no final da linha de compilação com o `gcc`.

Voltando agora para leitura, o destaque é para a especificação de formato `%lf%lf` usada no `sscanf`. Ela indica que dois valores reais devem ser buscados em `entrada` e cada um deve ser armazenado, respectivamente, nas variáveis `x` e `y`, ambas `double`. A ordem das variáveis deve corresponder à ordem em que os valores são digitados. Como já apresentado, as leituras de valores numéricos ignoram caracteres brancos antes de encontrar o valor em si, de forma que espaços ou tabulações antes de cada `%lf` são descartadas na varredura da linha, o que significa que, na digitação, a quantidade de espaços antes de cada valor é irrelevante. De forma complementar, tudo o que não corresponder a um valor real que apareça depois do segundo valor também é descartado.

A mistura de diferentes tipos em uma única linha também é possível, como indica o exemplo na sequência.

```
/*
Exemplos de leituras de tipos diferentes em uma mesma linha
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
    double d;
    int i1, i2;
    char c;

    printf("Digite um inteiro e um real: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%d%lf", &i1, &d);
    printf("O inteiro é %d e o o real é %g.\n\n", i1, d);

    printf("Digite um inteiro, um real e outro inteiro: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%d%lf%d", &i1, &d, &i2);
    printf("Os inteiros são %d e %d; o o real é %g.\n\n", i1, i2, d);

    printf("Digite um real seguido por um caractere: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%lf%c", &d, &c);
    printf("O real é %g e o caractere é %c.\n\n", d, c);

    return 0;
}
```

```
Digite um inteiro e um real: 320 44.5
O inteiro é 320 e o o real é 44.5.

Digite um inteiro, um real e outro inteiro: 10 1.1 20
Os inteiros são 10 e 20; o o real é 1.1.

Digite um real seguido por um caractere: 0.125ee
O real é 0.125 e o caractere é e.
```

Existem, naturalmente e previsivelmente, limitações nas leituras. Por exemplo, a última varredura usando `%lf%c` para obter um número real e um caractere esbarra na capacidade de análise dos caracteres digitados. Por exemplo, se o usuário digitar `0.1235A` é possível separar o 0,125 da letra `A`; se for digitado `0.125 A`, a variável `d` conterá o valor 0,125, mas `c` conterá o espaço, que é o próximo caractere depois do número, sendo o `A\n` que sobram ignorados. Além disso, é impossível com esse formato que o caractere seja um dígito, pois ele seria interpretado como parte do número e não como o caractere depois do número.

O contorno de tais limitações foge do escopo deste material.

4.5.5 Um pouco mais sobre o `sscanf`

O objetivo da função `sscanf` é analisar uma cadeia de caracteres procurando por padrões, os quais, reconhecidos adequadamente, são convertidos para o tipo indicado e atribuído às variáveis indicadas por seus endereços (razão do operador `&` usado nos exemplos diversos). Assim, ao se especificar `%d%d%lf` o `sscanf` espera encontrar dois inteiros e um real, nesta ordem. Para sumarizar, a Tabela 4.4 apresenta as principais especificações de formato usadas no `sscanf`.

Tabela 4.4: Relação entre formatos e tipos utilizados pelo `sscanf`.

Especificação	Tipo associado
`%d`	`int`
`%ld`	`long int`

Especificação	Tipo associado
%f	float
%lf	double
%c	char
%s	char[n]

Sobre os padrões interpretados no `sscanf`

O padrão especificado no segundo parâmetro da função `sscanf` é muito mais poderoso do que apenas a busca por números ou caracteres. Seguem alguns poucos exemplos sobre a versatilidade do `sscanf` na sua interpretação.

```
/*
Cálculo e apresentação da distância de um ponto em R^2 à origem, tendo como
    entrada os valores das coordenadas x e y desse ponto
Requer: o ponto (x, y)
Assegura: a distância de (x, y) à (0, 0)
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    printf("Digite um ponto no formato (x, y): ");
    fgets(entrada, sizeof entrada, stdin);

    double x, y;
    sscanf(entrada, "(%lf,%lf)", &x, &y);

    double distancia_origem = sqrt(x * x + y * y);
    printf("A distância de (%g, %g) a (0, 0) é %g.\n", x, y, distancia_origem);

    return 0;
}
```

```
Digite um ponto no formato (x, y): (3.2, -1.8)
A distância de (3.2, -1.8) a (0, 0) é 3.67151.
```

Este exemplo é uma releitura do programa que calcula a distância de um ponto à raiz, com referência ao Algoritmo 4.2. Nesta nova versão, a digitação da entrada deve seguir o formato convencional de representação de um ponto no plano, ou seja, circundar os valores com parênteses e usar uma vírgula para separar os x de y. O padrão que foi dado é `(%lf,%lf)`, o que significa que a função espera, nesta sequência, um abre parênteses, um valor real, uma vírgula, outro valor real e um fecha parênteses. O conteúdo provavelmente não será interpretado corretamente se o padrão não for completamente satisfeito.

O padrão de interpretação pode indicar que um dado valor será ignorado da interpretação. Para isso, um asterisco deve ser adicionado logo depois do símbolo `%`. Por exemplo, `%*lf` significa que um valor real deve ser reconhecido, mas seu valor será descartado. O exemplo seguinte mostra como, de uma linha com dois valores inteiros, utilizar apenas o segundo.

```
/*
Leitura de uma linha com quatro inteiros, porém descartando o primeiro e
o terceiro
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
```

```
    printf("Digite quatro valores inteiros: ");
    fgets(entrada, sizeof entrada, stdin);

    int segundo, quarto;
    sscanf(entrada, "%*d%d%*d%d", &segundo, &quarto);

    printf("Valores de interesse: %d e %d.\n", segundo, quarto);

    return 0;
}
```

```
Digite quatro valores inteiros: 6652 943 7609 -7
Valores de interesse: 943 e -7.
```

Interpretação em bases decimal, octal e hexadecimal

Além dos valores inteiros em decimal (`%d`), é possível interpretá-los nas bases 8 e 16. Nestes dois últimos casos, os valores devem ser sempre positivos e, para tanto, o tipo da variável tem que ser `unsigned int`, ou seja, um inteiro sem sinal.

```
/*
Leituras de valores inteiros nas bases decimal, octal e hexadecimal (10, 8
e 16, respectivamente)
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
    unsigned int valor;

    printf("Digite inteiro decimal: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%u", &valor);
    printf("O valor é %d(10), %o(8) e %X(16).\n\n", valor, valor, valor);

    printf("Digite inteiro octal: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%o", &valor);
    printf("O valor é %d(10), %o(8) e %X(16).\n\n", valor, valor, valor);

    printf("Digite inteiro hexadecimal: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%x", &valor);
    printf("O valor é %d(10), %o(8) e %X(16).\n\n", valor, valor, valor);

    return 0;
}
```

```
Digite inteiro decimal: 63
O valor é 63(10), 77(8) e 3F(16).

Digite inteiro octal: 63
O valor é 51(10), 63(8) e 33(16).

Digite inteiro hexadecimal: 63
O valor é 99(10), 143(8) e 63(16).
```

Os dígitos 6 e 3 são dígitos válidos nas três bases exemplificadas e 63_{10}, 63_8 e 63_{16} têm sua interpretação descrita na Tabela 4.5. A conversão do valor digitado depende do formato expresso no `sscanf`: decimal (`%d`), octal (`%o`) ou hexadecimal (`%x`). A escolha de um dos formatos invalida a interpretação dos outros dois.

Tabela 4.5: Interpretação dos dígitos 63 nas bases decimal, octal e hexadecimal.

Valor	Interpretação	Valor decimal equivalente
63_{10}	$6 \times 10^1 + 3 \times 10^0$	63
63_8	$6 \times 8^1 + 3 \times 8^0$	51
63_{16}	$6 \times 16^1 + 3 \times 16^0$	99

É possível dar liberdade ao usuário na escolha da base que será usada. A especificação de formato `%i` significa um valor inteiro, independente da base. Valores decimais são, como esperado, interpretados como decimais; valores iniciados com 0 são interpretados como números octais e os precedidos da sequência `0x` são considerados na base 16.

```
/*
Leituras genérica de valores inteiros
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
    int valor;

    printf("Digite inteiro: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%i", &valor);
    printf("O valor é %d(10), %o(8) e %X(16).\n\n", valor, valor, valor);

    printf("Digite inteiro: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%i", &valor);
    printf("O valor é %d(10), %o(8) e %X(16).\n\n", valor, valor, valor);

    printf("Digite inteiro: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%i", &valor);
    printf("O valor é %d(10), %o(8) e %X(16).\n\n", valor, valor, valor);

    return 0;
}
```

```
Digite inteiro: 63
O valor é 63(10), 77(8) e 3F(16).

Digite inteiro: 063
O valor é 51(10), 63(8) e 33(16).

Digite inteiro: 0x63
O valor é 99(10), 143(8) e 63(16).
```

Neste exemplo, `63` é o decimal 63, `063` é 63_8 e `0x63` é 63_{16}.

Erros de interpretação do padrão

A função `sscanf` é capaz de interpretar corretamente um valor numérico tanto quanto o valor faça sentido. Se houver um erro na interpretação, a análise da varredura é interrompida e o valores corretamente convertidos são atribuídos às respectivas variáveis; as variáveis restantes não têm seu valor modificado.

O exemplo seguinte apresenta a situação de duas leituras

```c
/*
Leituras corretas e incorretas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    // Valores iniciais
    int i1 = 1, i2 = 2, i3 = 3, i4 = 4;
    printf("i1 = %d; i2 = %d; i3 = %d; i4 = %d.\n\n", i1, i2, i3, i4);

    // Leitura 1
    printf("Digite os valores para i1, i2, i3 e i4: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%d%d%d%d", &i1, &i2, &i3, &i4);
    printf("i1 = %d; i2 = %d; i3 = %d; i4 = %d.\n\n", i1, i2, i3, i4);

    // Leitura 2
    printf("Digite os valores para i1, i2, i3 e i4: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%d%d%d%d", &i1, &i2, &i3, &i4);
    printf("i1 = %d; i2 = %d; i3 = %d; i4 = %d.\n\n", i1, i2, i3, i4);

    return 0;
}
```

```
i1 = 1; i2 = 2; i3 = 3; i4 = 4.

Digite os valores para i1, i2, i3 e i4: 10 20 30 40
i1 = 10; i2 = 20; i3 = 30; i4 = 40.

Digite os valores para i1, i2, i3 e i4: 100 200 abc 400
i1 = 100; i2 = 200; i3 = 30; i4 = 40.
```

Na primeira leitura do programa, todos os valores são lidos corretamente e todas as atribuições são feitas. Na segunda leitura a varredura falha ao encontrar `abc` quando um número inteiro era esperado. Com o erro na interpretação, apenas `i1` e `i2` são atualizados, enquanto `i3` e `i4` não têm seus valores modificados.

`sscanf` é uma função e tem valor de retorno

Embora frequentemente usada como um comando simples, `sscanf` é, na realidade, uma função que retorna o número de leituras corretamente realizadas. Com essa característica, é possível contornar erros de leitura e deixar o código mais robusto.

```c
/*
Verificação de leituras corretas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite os valores para i1, i2, i3 e i4: ");
    fgets(entrada, sizeof entrada, stdin);
    int i1, i2, i3, i4;
    int numero_atribuicoes = sscanf(entrada, "%d%d%d%d", &i1, &i2, &i3, &i4);
    printf("i1 = %d; i2 = %d; i3 = %d; i4 = %d.\n\n", i1, i2, i3, i4);
    printf("Na leitura feita, %d valores foram corretamente lidos.\n",
        numero_atribuicoes);

    return 0;
}
```

```
Digite os valores para i1, i2, i3 e i4: 10 20 abc 40
i1 = 10; i2 = 20; i3 = 0; i4 = 0.

Na leitura feita, 2 valores foram corretamente lidos.
```

Com apenas dois valores corretamente lidos, a execução do programa mostra que os valores originais de `i3` e `i4` são preservados. Essas variáveis recaem na categoria variáveis não iniciadas, como exemplificado em uma das dicas da Seção 4.3.

4.6 Ressalvas quanto ao `scanf`

Em grande parte do material disponível em páginas na Internet é comum que a leitura use a função `scanf` no lugar de um `fgets` seguido de um `sscanf`. O objetivo da função `scanf` é aplicar as especificações de formato diretamente na entrada de dados, sem usar a variável `entrada` como nos exemplos apresentados.

Essa prática de usar diretamente `scanf` leva a uma dificuldade muito grande quando leituras de cadeias de caracteres e de valores numéricos, visto que essa função varre o que foi digitado, porém mantém o que ainda não foi analisado. O próprio manual do `scanf` apresenta o conteúdo seguinte.

```
The scanf() family  of functions scans input like sscanf(3), but read
from a FILE.  It is very difficult to use  these  functions  correctly,
and  it  is preferable to read entire lines with fgets(3) or getline(3)
and parse them later with sscanf(3) or more specialized functions  such
as strtol(3).
```

Traduzindo livremente as partes mais relevantes: "é muito difícil usar essas funções corretamente"; "é preferível ler linhas inteiras com `fgets` (...) e analisá-las com `sscanf` (...)". Em outras palavras, o uso de `scanf` diretamente exige conhecimento mais detalhado de como o fluxo de entrada é tratado e, desta forma, foi substituído por outros comandos, conforme a recomendação do próprio manual.

O texto também dá como alternativas `getline` (muito similar a `fgets`), e `strtol` (com os mesmos objetivos de `sscanf`). A função `getline` tem os mesmos parâmetros de `fgets` e pode ser usada sem seu lugar, mas tem vantagens quando se usa memória alocada dinamicamente. Por sua vez, `strtol` é também interessante, mas exige o uso de ponteiros e o entendimento de endereçamento de memória.

O emprego de `getline` e `strtol` em substituição a `fgets` e `sscanf` é uma sugestão interessante para quando os conceitos de alocação dinâmica de memória e ponteiros fizerem parte dos conhecimentos do programador. Ponteiros são tratados no Capítulo 20, enquanto o **?@sec-alocacao-dinamica-de-memoria** aborda a alocação dinâmica de memória.

4.7 Sobre a função `gets`

Infelizmente, quando se procura por informações sobre leitura de dados em C nos diversos mecanismos de busca, ainda são comuns os exemplos usando a função `gets`. Esta função é uma "versão simplificada" de `fgets` que não precisa informar o espaço de memória disponível para a leitura nem requer a especificação do `stdin`, que é usado automaticamente.

O problema desta função é exatamente a falta de especificação da área de memória disponível, pois a leitura não respeita qualquer limite e pode sobrescrever outras áreas importantes da memória, modificando indiretamente outras variáveis e até as instruções que serão executadas.

O exemplo seguinte mostra o uso, a compilação e o resultado de uma leitura usando `gets`.

```
/*
Leitura de um texto com gets
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite algo: ");
    gets(entrada);
    printf("Você digitou: '%s'\n", entrada);

    return 0;
}
```

```
main.c: In function ´main´:
main.c:10:5: warning: implicit declaration of function ´gets´; did you
mean ´fgets´? [-Wimplicit-function-declaration]
   10 |     gets(entrada);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccR7ySaS.o: na função "main":
main.c:(.text+0x2f): aviso: the `gets' function is dangerous and should not
be used.
```

```
Digite algo: C é legal, mas não é simples...
Você digitou: 'C é legal, mas não é simples...'
```

Da compilação desse programa, o destaque é feito para a linha com o aviso: “a função `gets` é perigosa e não deve ser usada”.

Além dessa recomendação do próprio compilador, há ainda no manual da função o trecho reproduzido na sequência. Nesse segmento do texto, o destaque é para a sentença: “nunca use esta função”.

```
DESCRIPTION
       Never use this function.

       gets()  reads  a  line from stdin into the buffer pointed to by s until
       either a terminating newline or EOF, which it replaces with a null byte
       ('\0').  No check for buffer overrun is performed (see BUGS below).
```

Para deixar bem claras as consequências dessa função, considere o programa seguinte. Nele, o tamanho disponível para `entrada` é de 20 bytes, o que limita o texto máximo a 19. Também foi acrescentada uma variável inteira, usada apenas para ilustrar o problema.

```
/*
Leitura de um texto com gets
*/
#include <stdio.h>

int main(void) {
    char entrada[20];
    int valor = 10;

    printf("'valor' vale %d\n", valor);
    printf("Digite algo: ");
    gets(entrada);
    printf("Você digitou: '%s'\n", entrada);
    printf("'valor' vale %d\n", valor);

    return 0;
}
```

```
main.c: In function ´main´:
main.c:12:5: warning: implicit declaration of function ´gets´; did you
mean ´fgets´? [-Wimplicit-function-declaration]
   12 |     gets(entrada);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccsz4mDQ.o: na função "main":
main.c:(.text+0x49): aviso: the `gets' function is dangerous and should not
be used.
```

```
'valor' vale 10
Digite algo: C é legal, mas não é simples...
Você digitou: 'C é legal, mas não é simples...'
'valor' vale 779314540
```

É importante reparar que o código permite, inadvertidamente, que a variável inteira `valor` tenha seus bytes modificados pela insegurança de `gets`. Dado que não há verificação do espaço disponível para armazenar a leitura, os bytes digitados pelo usuário ultrapassam os 20 bytes de `entrada` e destroem os bytes de `valor`. Este programa aparentemente não tem problemas, até que uma entrada seja maior que o espaço disponível.

A conclusão simples e prática desta seção é, portanto, não usar `gets`. Nunca.

5 Expressões aritméticas de C

Uma coisa que um computador consegue fazer com eficiência são contas e faz isso muito rapidamente. Nesta parte são tratados os principais aspectos dos cálculos feitos por programas escritos em C.

Uma expressão aritmética é aquela que envolve valores numéricos e, pelo uso operadores, produzem um resultado também numérico. Um exemplo de uma expressão aritmética é o cálculo do discriminante $b^2 - 4ac$, referente a uma equação de segundo grau $ax^2 + bx + c = 0$ ($a \neq 0$).

5.1 Operadores aritméticos

A linguagem C dispõe dos operadores aritméticos tradicionais para soma, subtração, multiplicação e divisão. Como na matemática, os operadores possuem prioridades entre si, sendo que multiplicações e divisões tem precedência sobre somas e subtrações. Os operadores aritméticos são apresentados na Tabela 5.1.

Tabela 5.1: Operadores aritméticos. A ordem precedência vai de 1 (maior precedência) a 3 (menor precedência). A associatividade pode ser da esquerda para a direita (→) ou da direita para a esquerda (←).

Operador	Descrição	Precedência	Sintaxe	Associatividade
`+` (unário)	Mantém o sinal do operando seguinte	1	`+a`	←
`-` (unário)	Alteração do sinal do operando seguinte	1	`-a`	←
`*`	Multiplicação de dois operandos	2	`a * b`	→
`/`	Divisão do operando esquerdo pelo direito	2	`a / b`	→
`%`	Módulo do operando direito pelo esquerdo	2	`a % b`	→
`+` (binário)	Soma de dois operandos	3	`a + b`	→
`-` (binário)	Subtração do operando direito do esquerdo	3	`a - b`	→

O programa seguinte exemplifica o uso dos operadores para valores inteiros.

```c
/*
Operadores aritméticos com argumentos inteiros
*/
#include <stdio.h>

int main(void) {
    int operando1 = 15;
    int operando2 = 6;

    printf("Operandos em uso: op1 = %d e op2 = %d.\n", operando1, operando2);
    printf("+op1 = %d\n", +operando1);
    printf("-op1 = %d\n", -operando1);
    printf("op1 * op2 = %d\n", operando1 * operando2);
    printf("op1 / op2 = %d\n", operando1 / operando2);
    printf("op1 + op2 = %d\n", operando1 + operando2);
    printf("op1 - op2 = %d\n", operando1 - operando2);
    printf("op1 %% op2 = %d\n", operando1 % operando2);

    return 0;
}
```

```
Operandos em uso: op1 = 15 e op2 = 6.
+op1 = 15
-op1 = -15
op1 * op2 = 90
op1 / op2 = 2
op1 + op2 = 21
op1 - op2 = 9
op1 % op2 = 3
```

O resultado da execução é, em grande parte, o esperado. Há, porém, um detalhe importante relativo à divisão. Seria esperado que a expressão `15 / 6` resultasse em aproximadamente 2,14286, porém o resultado foi 2. Em C, a divisão de um inteiro por outro resulta em outro inteiro; a parte decimal resultante da divisão é ignorada. Desta forma, `3 / 2` é igual a 1, `20 / 6`, `32 / 6` é calculado como 5 e `999 / 1000` resulta em zero. Não são feitos arredondamentos; a parte inteira é truncada.

O operador `%` é o não tão conhecido operador modular, escrito $k \bmod n$, e que corresponde ao resto da divisão de k por n. No exemplo, `15 % 6` é o resto da divisão de 15 por 6, ou seja, 3.

A multiplicação é indicada pelo `*`, que deve sempre ser explícito. Nas equações matemáticas, a ausência do operador é imediatamente associada à multiplicação (xy significa x multiplicado por y). Em C, deve-se sempre escrever `x * y`.

> **Aritmética modular**
>
> A aritmética modular, que é um sistema para inteiros, corresponde números cíclicos. Para módulo 3, por exemplo, os números são sequencialmente 0, 1, 2, 0, 1, 2, 0,..., ciclicamente. Assim, para se chegar ao 7, a sequência seria 1, 2, 0, 1, 2, 0, 1. Nesta contagem, $3 \equiv 0 \pmod 3$, pois ambos chegam ao zero no final. Ainda tem-se $1 \equiv 4 \pmod 3$, $2 \equiv 5 \pmod 3$ e $2 \equiv 8 \pmod 3$, por exemplo.
> Um exemplo prático de contagem modular é o relógio de 12 horas, que inicia às 0h e, quando chegaria às 12h, volta a contar do zero novamente. Se o relógio marca, por exemplo, 9h, depois de cinco horas ele marcará 2h, pois opera mod 12. Ou seja, $(9 + 5) \equiv 2 \pmod{12}$.
> O operador módulo, denotado por $k \bmod n$, corresponde a um inteiro único r tal que $0 \leq r \leq n$ e $r \equiv n \pmod n$.
> A consequência prática e útil da operação modular na computação é que, para valores inteiros positivos, `a % b` corresponde ao resto da divisão inteira de `a` por `b`. Assim, `8546 % 43` resulta em 32, ou seja, 8546 dividido por 43 é 198 com resto 32.

Na sequência é apresentado um programa usando os operadores aritméticos com dados do tipo `double`.

```c
/*
Operadores aritméticos com argumentos inteiros
*/
#include <stdio.h>

int main(void) {
    double operando1 = 12.5;
    double operando2 = 3.8;

    printf("Operandos em uso: op1 = %g e op2 = %g.\n", operando1, operando2);
    printf("+op1 = %g\n", +operando1);
    printf("-op1 = %g\n", -operando1);
    printf("op1 * op2 = %g\n", operando1 * operando2);
    printf("op1 / op2 = %g\n", operando1 / operando2);
    printf("op1 + op2 = %g\n", operando1 + operando2);
    printf("op1 - op2 = %g\n", operando1 - operando2);

    return 0;
}
```

```
Operandos em uso: op1 = 12.5 e op2 = 3.8.
+op1 = 12.5
-op1 = -12.5
op1 * op2 = 47.5
op1 / op2 = 3.28947
op1 + op2 = 16.3
op1 - op2 = 8.7
```

Quando os operadores são usados em valores reais (`double`), os resultados são os esperados da matemática. Como a aritmética modular é aplicada exclusivamente a valores inteiros, o operador `%` é inválido quando os operandos são `double`.

5.1.1 Ordem de avaliação

A ordem de avaliação de uma expressão aritmética em C pode ser traiçoeira. Embora, em grande parte das vezes, o valor de uma expressão seja direto, há um não tão pequeno número de situações em que o programador pode se equivocar.

Para avaliar uma expressão, o compilador usa os seguintes passos de decisão:

1. Para operadores com precedências diferentes, a ordem da precedência é seguida;
2. Para operadores de mesma precedência, a associatividade é seguida.

Por exemplo, na expressão `a + b * c`, `b * c` é avaliado primeiro, sendo a soma realizada em um segundo momento.

Para a expressão `a * b * c + d * e`, a ordem de avaliação é:

1. Primeiro ocorre a multiplicação de `a` por `b`, gerando um resultado intermediário r_1;
2. Depois, r_1 é multiplicado por `c` (r_2);
3. Como a multiplicação tem precedência sobre a soma, `d` e `e` são multiplicados (r_3);
4. Por fim, r_2 e r_3 são somados.

A ordem de avaliação, portanto, é `(((a * b) * c) + (d * e))`, embora se saiba que a ordem de multiplicação dos três primeiros termos não interfira no resultado.

Felizmente, para os operadores de soma, subtração e multiplicação, a ordem seguida pelo compilador é normalmente a esperada.

Quando há uma divisão, uma margem para erros acaba se apresentando. Como exemplo, pode-se considerar a expressão `a * b / c * d`, na qual todos os operadores possuem a mesma ordem de precedência, mas diferentes ordens de avaliação levam a resultados distintos. Para este caso, a ordem de avaliação é:

1. É feita inicialmente a multiplicação de `a` por `b` (r_1);
2. Em seguida, r_1 é dividido por `c`, resultando r_2;
3. Por último, é feita a multiplicação de r_2 por `d`, gerando o resultado final.

A ordem de avaliação segue a associatividade dos operadores de mesma precedência, que neste caso é da esquerda para a direita. Assim, `a * b / c * d` é avaliado como `(((a * b) / c) * d)`.

5.1.2 Quebra da ordem de precedência e da associatividade

Conhecido o procedimento usado pelo compilador para avaliar uma expressão aritmética, fica expressa a necessidade de, por vezes, escolher uma ordem diferente. A linguagem C usa os parênteses para definir a ordem de avaliação. Um exemplo importante e simples é o cálculo media: `(v1 + v2)/2`. Sem os parênteses, apenas `v2` seria dividido por 2.

Apenas parênteses são usados nas expressões, pois colchetes e chaves servem a outros propósitos na linguagem.

Considerando a, título de exemplo, que uma variável real `discriminante` contenha o valor $b^2 - 4ac$ correspondente a uma equação de segundo grau $ax^2 + bx + c = 0$ $(a \neq 0)$, a Tabela 5.2 mostra tentativas de escrever o cálculo de uma das raízes da equação, considerando que o discriminante seja não negativo.

Tabela 5.2: Exemplos de tentativas de código para calcular uma raiz de uma equação de segundo grau com discriminante $\Delta \geq 0$.

Expressão do código	Expressão equivalente	Resultado
`-b - sqrt(discriminante) / 2 * a`	$-b - \frac{\sqrt{\Delta}}{2}a$	Incorreto
`-b - sqrt(discriminante) / (2 * a)`	$-b - \frac{\sqrt{\Delta}}{2a}$	Incorreto
`(-b - sqrt(discriminante)) / 2 * a`	$\frac{-b - \sqrt{\Delta}}{2}a$	Incorreto
`(-b - sqrt(discriminante))  /  2*a`	$\frac{-b - \sqrt{\Delta}}{2}a$	Incorreto
`(-b - sqrt(discriminante)) / (2 * a)`	$\frac{-b - \sqrt{\Delta}}{2a}$	Correto
`(-b - sqrt(discriminante)) / 2 / a`	$\frac{-b - \sqrt{\Delta}}{2a}$	Correto

Dica

Em C, o compilador ignora os espaços entre os operadores. A ordem que vale é a da precedência, em primeiro lugar, e da associatividade, para precedências iguais. Dessa forma, é indiferente para o compilador escrever `a/b*c`, `a / b * c` ou `a / b*c`, pois todas serão avaliadas `((a / b) * c)`.
Mesmo que desnecessário, muitas vezes o uso dos parênteses para deixar clara a ordem de avaliação que o programador deseja ajuda na legibilidade e entendimento do código fonte.
Por exemplo, é comum, no lugar de elevar um valor ao quadrado, usar a multiplicação dele por ele mesmo. Uma expressão que usa esse recurso, por exemplo, seria $a^2(b+c)^2$, que poderia ser escrita `(a * a) * ((b + c) * (b + c))`, mesmo que `a * a * (b + c) * (b + c)` seja exatamente equivalente. Os parênteses servem apenas para ênfase.

5.2 Expressões inteiras vs. reais (promoção de tipo)

As expressões aritméticas envolvem valores numéricos, mas C pode tratar de forma diferente valores numéricos inteiros (`int`) e reais (`double`). Um exemplo é a divisão, que resulta em valores diferentes para `18/7` e `18.0/7.0`, por exemplo, ou o operador modular `%`, que não pode ser usado para valores reais.

Para iniciar o entendimento de como expressões com tipos diferentes são tratadas, o exemplo da conversão de temperaturas do Algoritmo 1.2 é revisitado.

```
/*
Conversão de escalas termométricas, de graus Celsius para Fahrenheit
Requer: valor da temperatura em graus Celsius
Assegura: valor da temperatura em Fahrenheit
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Temperatura em Celsius: ");
    fgets(entrada, sizeof entrada, stdin);
    double temperatura_celsius;
    sscanf(entrada, "%lf", &temperatura_celsius);

    double temperatura_fahrenheit = 1.8 * temperatura_celsius + 32;
    printf("%g graus Celsius = %g Fahrenheit.\n", temperatura_celsius,
        temperatura_fahrenheit);

    return 0;
}
```

```
Temperatura em Celsius: 23.5
23.5 graus Celsius = 74.3 Fahrenheit.
```

O interesse neste programa está na expressão que faz a conversão de escalas termométricas.

```
double temperatura_fahrenheit = 1.8 * temperatura_celsius + 32;
```

A expressão multiplica um valor `double`, o 1,8 por outro também `double`, resultando em um valor intermediário que mantém o tipo dos operandos. Em seguida, o valor 32 é somado, mas esse valor é do tipo `int`.

A questão é que claramente `int` somado com `int` resulta em `int` e, de modo análogo, a soma de `double` resulta em `double`. Como o compilador pode, então, lidar com a soma de um `double` com um `int`?

Quando uma expressão envolve tipos diferentes, um mecanismo conhecido como promoção de tipo é empregado. A promoção segue o princípio de que, se os tipos são diferentes, o "menor" deve ser promovido ao tipo do "maior". Neste caso, "menor" e "maior" se referem à capacidade de armazenamento dos tipos envolvidos. Assim, um `double` é considerado "maior" (mais abrangente) que um `int` (mais restrito). Essa análise é feita a cada operação.

Retornando ao caso da soma de um `double` com um `int`, primeiramente o `int` é convertido para `double` e, então, a soma é feita, resultando evidentemente em `double`.

Muitas vezes a expressão usada para a conversão de Celsius para Fahrenheit é dada na forma $t_f = \frac{9}{5}t_c + 32$. Usando-se essa configuração, talvez fosse possível substituir a linha do cálculo no programa pela versão seguinte.

```
double temperatura_fahrenheit = 9 / 5 * temperatura_celsius + 32;  // incorreta!
```

Essa expressão apresenta um erro no cálculo. Considerando-se que a divisão e a multiplicação possuem a mesma precedência, a expressão é avaliada da esquerda para a direita, de forma que `9 / 5` é calculada primeiro, para então multiplicar por `temperatura_celsius`. O problema é que `9 / 5` é igual a 1, pois ambos os operadores são inteiros. Da forma que está escrita, o valor efetivamente atribuído é `1 * temperatura_celsius + 32`.

É nítida a consequência da ordem das operações no resultado final, o que leva a uma nova versão para a expressão.

```
double temperatura_fahrenheit = temperatura_celsius * 9 / 5 + 32;  // correta!
```

Seguindo a ordem de avaliação, a primeira operação realizada é a multiplicação de `temperatura_celsius` por 9, sendo o primeiro operando `double` e o segundo, `int`. Aplicada a promoção do 9 para 9.0, a multiplicação é feita, com um resultado do tipo `double`. Quando é feita a divisão por 5, novamente o segundo operando é promovido (de 5 para 5.0) e novamente se obtém um resultado `double`. Finalmente a soma é feita, havendo uma nova promoção (do 32), obtendo-se o resultado final. É importante notar que o problema da divisão inteira foi evitado, uma vez que, em nenhum momento, a divisão teve que lidar com dois valores `int`.

Um novo exemplo segue, o qual tem que lidar com conversões de tipo. O programa implementa o Algoritmo 5.1.

Algoritmo 5.1: Cálculo de porcentagens considerando o montante de votos.

Descrição: Cálculo simples de porcentagens em uma pesquisa com as seguintes possibilidades de resposta: sim, não e não sei. A pergunta feita é irrelevante.
Requer: a quantidade de votos *sim*, *não* e *não sei*
Assegura: as porcentagens, exibidas no intervalo [0, 1], de cada opção de voto
Obtenha $quantidade_{sim}$, $quantidade_{não}$ e $quantidade_{não\ sei}$
Calcule a porcentagem de cada categoria dividindo o número de votos da categoria pelo total de votos
Apresente as quantidade e porcentagens para cada categoria

```
/*
Cálculo simples de porcentagens em uma pesquisa com as seguintes
    possibilidades de resposta: sim, não e não sei. A pergunta feita é
    irrelevante.
Requer: a quantidade de votos sim, não e não sei.
Assegura: as quantidades de votos e respectivas as porcentagens,
    exibidas no intervalo [0, 1], de cada opção de voto
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    // Obtenção das quantidades
    printf("Número de respostas SIM: ");
    fgets(entrada, sizeof entrada, stdin);
    int quant_votos_sim;
    sscanf(entrada, "%d", &quant_votos_sim);

    printf("Número de respostas NÃO: ");
    fgets(entrada, sizeof entrada, stdin);
    int quant_votos_nao;
    sscanf(entrada, "%d", &quant_votos_nao);

    printf("Número de respostas NÃO SEI: ");
    fgets(entrada, sizeof entrada, stdin);
    int quant_votos_nao_sei;
    sscanf(entrada, "%d", &quant_votos_nao_sei);

    // Cálculo das porcentagens (COM ERROS)
    int quant_total = quant_votos_sim + quant_votos_nao + quant_votos_nao_sei;

    double porcentagem_sim = quant_votos_sim / quant_total;
    double porcentagem_nao = quant_votos_nao / quant_total;
    double porcentagem_nao_sei = quant_votos_nao_sei / quant_total;

    // Resultados
```

```
    printf("Sim: %d votos (%.2f).\n", quant_votos_sim, porcentagem_sim);
    printf("Não: %d votos (%.2f).\n", quant_votos_nao, porcentagem_nao);
    printf("Não sei: %d votos (%.2f).\n", quant_votos_nao_sei,
           porcentagem_nao_sei);

    return 0;
}
```

```
Número de respostas SIM: 852
Número de respostas NÃO: 432
Número de respostas NÃO SEI: 73
Sim: 852 votos (0.00).
Não: 432 votos (0.00).
Não sei: 73 votos (0.00).
```

O exemplo mostra que todas as porcentagens foram calculadas como zero e o problema é a divisão de dois inteiros. A divisão inteira ocorre antes da atribuição, pois toda a expressão já foi calculada. Mesmo com um resultado inteiro, há uma promoção para `double` ao se fazer a atribuição, dado o tipo da variável.

Agora o problema é fazer com que a divisão de dois inteiros resulte em um valor real. Para isso pode ser empregada uma promoção de tipo explícita, usando o que é chamado *type cast* em C. Um *cast* de tipo é indicado pelo tipo desejado colocado entre parênteses antes do valor. Por exemplo, ao se escrever `(double)10`, o valor 10 é convertido para `double`; a expressão `(int)valor` converte o conteúdo armazenado na variável para `int`. O *cast* é sempre aplicado ao item imediamente seguinte na expressão, sendo que em `(double)valor_inteiro1 + valor_inteiro2`, apenas `valor_inteiro1` é afetado pelo modificador de tipo.

Usando-se essa modificação de tipo, segue uma versão corrigida do programa.

```
/*
Cálculo simples de porcentagens em uma pesquisa com as seguintes
    possibilidades de resposta: sim, não e não sei. A pergunta feita é
    irrelevante.
Requer: a quantidade de votos sim, não e não sei.
Assegura: as quantidades de votos e respectivas as porcentagens,
    exibidas no intervalo [0, 1], de cada opção de voto
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    // Obtenção das quantidades
    printf("Número de respostas SIM: ");
    fgets(entrada, sizeof entrada, stdin);
    int quant_votos_sim;
    sscanf(entrada, "%d", &quant_votos_sim);

    printf("Número de respostas NÃO: ");
    fgets(entrada, sizeof entrada, stdin);
    int quant_votos_nao;
    sscanf(entrada, "%d", &quant_votos_nao);

    printf("Número de respostas NÃO SEI: ");
    fgets(entrada, sizeof entrada, stdin);
    int quant_votos_nao_sei;
    sscanf(entrada, "%d", &quant_votos_nao_sei);

    // Cálculo das porcentagens
    int quant_total = quant_votos_sim + quant_votos_nao + quant_votos_nao_sei;

    double porcentagem_sim = (double)quant_votos_sim / quant_total;
    double porcentagem_nao = (double)quant_votos_nao / quant_total;
    double porcentagem_nao_sei = (double)quant_votos_nao_sei / quant_total;

    // Resultados
```

```
    printf("Sim: %d votos (%.2f).\n", quant_votos_sim, porcentagem_sim);
    printf("Não: %d votos (%.2f).\n", quant_votos_nao, porcentagem_nao);
    printf("Não sei: %d votos (%.2f).\n", quant_votos_nao_sei,
           porcentagem_nao_sei);

    return 0;
}
```

```
Número de respostas SIM: 852
Número de respostas NÃO: 432
Número de respostas NÃO SEI: 73
Sim: 852 votos (0.63).
Não: 432 votos (0.32).
Não sei: 73 votos (0.05).
```

Neste caso, vale o comentário de que em `(double)quant_votos_sim / quant_total` somente a primeira variável é explicitamente promovida a `double`, sendo que a segunda é promovida em função da realização da divisão.

Uma estratégia comum é o uso do elemento neutro para forçar o resultado desejado. Essa alternativa permitiria escrever, por exemplo `1.0 * quant_votos_sim / quant_total`. O autor tem como opinião que o uso do *cast* de tipo explícito é mais elegante, embora o resultado seja equivalente.

Dica

Em uma expressão aritmética que tenha uma divisão, sempre é preciso atenção quanto ao tipo dos operandos para que se tenha certeza que o resultado seja corretamente calculado.

Algumas conversões são comuns nas atribuições e, em geral, passam despercebidas. Considerando-se as atribuições abaixo, as conversões implícitas fazem sentido.

```
double d1 = 0;
double d2 = 10;
```

As constantes `0` e `10` são intrinsecamente inteiras, mas devido à variável ser real, ambas são promovidas antes da atribuição. Não há necessidade, por exemplo, de se escrever `d1 = 0.0`.

Como mencionado, também é possível converter um valor mais abrangente, como um `double` para um menos abrangente, como o `int`. Segue um exemplo.

```
/*
Exemplo de conversão de double para inteiro
*/
#include <stdio.h>

int main(void) {
    double d = 3.75;

    int i1 = 2 * (int)d;
    int i2 = 2 * d;

    printf("i1 = %d e i2 = %d.\n", i1, i2);

    return 0;
}
```

```
i1 = 6 e i2 = 7.
```

Para a variável `i1`, primeiro a parte decimal de `d` é eliminada e o resultado é multiplicado por 2. Já para `i2`, primeiro é feita a multiplicação de 2 por `d` e somente então o resultado é truncado.

A noção de promoção de tipo é valida não somente de `int` para `double`, mas entre quaisquer outros tipos específicos.

```
/*
Exemplo de conversão entre double e float
*/
#include <stdio.h>

int main(void) {
    float f = 3.75;
    double d = 2 * f;

    printf("f = %g e d = %g.\n", f, d);

    return 0;
}
```

```
f = 3.75 e d = 7.5.
```

Neste programa, `d` é `double`e `f` possui de precisão simples `float`. Na atribuição para `f` deve-se notar que `3.74` é `double`, o qual é "reduzido" para `float` antes da atribuição. Já na atribuição para `d`, `f` é promovida para `double` ao ser multiplicada por 2 (que é `double`).

Um programa que ilustra promoções entre inteiros é apresentada na sequência.

```
/*
Exemplo de conversão entre variações de inteiros
*/
#include <stdio.h>

int main(void) {
    int i = 2000000000;
    long int li;

    li = 4 * i;
    printf("%ld.\n", li);

    li = 4L * i;
    printf("%ld.\n", li);

    return 0;
}
```

```
-589934592.
8000000000.
```

Nesse programa, o valor 2.000.000.000 é intencionalmente grande, pois chega quase ao limite de representação para valores do tipo `int` e, quando multiplicado por 4 certamente causará um transbordo de bits (*overflow*)[1]. A operação `4 * i` multiplica dois valores `int` e o resultado transborda, produzindo um valor incorreto. Quando a operação feita é `4L * i`, o valor `4L` é uma constante `long int` e a multiplicação primeiro promove `i` para `long int` e o resultado agora tem seus bits comportados pela representação.

5.3 funções matemáticas

Existe um arquivo de cabeçalho chamado `math.h`, o qual define a interface a uma série de funções matemáticas, como radiciação, potenciação e logaritmos, entre outras. O uso das funções requer a inclusão do cabeçalho no código fonte.

[1] Há um transbordo quando o resultado de uma operação precisa de mais bits que o tipo tem disponível, corrompendo a representação por causa dos bits perdidos.

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

Na Tabela 5.3 são apresentadas algumas das principais funções matemáticas disponíveis. Os argumentos da função são sempre especificados usando-se parênteses e, caso haja mais que um parâmetro, eles são separados por vírgulas. Quando uma função é usada, o tipo de dado usado na expressão é o indicado por seu tipo de retorno.

Como um exemplo específico, a função `pow` aceita dois argumentos, ambos do tipo `double`: `pow(1.5, 2.3)` indica o cálculo de $1{,}5^{2{,}3}$. Se essa função for usada em uma expressão como `pow(1.5, 2.3) - 10`, a subtração vê o operando à esquerda como `double` (tipo de retorno da função) e o à esquerda como `int`, fazendo a promoção devida antes de gerar o resultado.

Tabela 5.3: Algumas funções matemáticas importantes disponibilizadas por meio de `math.h`.

Função	Tipo de retorno	Descrição
`sin(double x)`	`double`	Retorna o seno de x (em radianos).
`cos(double x)`	`double`	Retorna o cosseno de x (em radianos).
`tan(double x)`	`double`	Retorna tangente de x (em radianos).
`exp(double x)`	`double`	Retorna e^x.
`log(double x)`	`double`	Retorna $\log_e x$.
`log10(double x)`	`double`	Retorna $\log_{10} x$.
`sqrt(double x)`	`double`	Retorna $\sqrt{x}$.
`fabs(double x)`	`double`	Retorna $\lvert x \rvert$ (valor absoluto).
`pow(double x, double y)`	`double`	Retorna x^y.

Os parâmetros e valor de retorno da função são, em geral, do tipo `double`. Há versões para outros tipos, como o valor absoluto dado por `fabsf`, que aceita e retorna `float`, e a raiz quadrada calculada com `sqrtl`, que aceita e retorna `long double`.

Para uso das funções matemáticas, a biblioteca matemática tem que ser conectada ao código executável, de forma que a opção `-lm` (“*link to the math library*”) tem que ser acrescentada às opções de compilação.

```
gcc main.c -lm
```

```
/*
Distância entre dois pontos no plano
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    printf("X e Y para P1: ");
    fgets(entrada, sizeof entrada, stdin);
    double x1, y1;
    sscanf(entrada, "%lf%lf", &x1, &y1);

    printf("X e Y para P2: ");
    fgets(entrada, sizeof entrada, stdin);
    double x2, y2;
    sscanf(entrada, "%lf%lf", &x2, &y2);

    double distancia = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
    printf("D[(%g, %g), (%g, %g)] = %g.\n", x1, y1, x2, y2, distancia);

    return 0;
}
```

```
X e Y para P1: 1.5 4.7
X e Y para P2: -3.2 0.5
D[(1.5, 4.7), (-3.2, 0.5)] = 6.30317.
```

Como as conversões de tipo são um tópico importante nesta seção do texto, é interessante identificar que, ao elevar ao quadrado com a função `pow`, para o segundo argumento foi usado `2` (que é `int`) e a função espera um `double`; neste caso o `int` é automaticamente promovido para `double` ao ser chamada a função.

5.4 Atribuições são operadores

Da mesma forma que `+` ou `/` são operadores, também é a atribuição com `=`. Ao se escrever `valor = 10`, por exemplo, o sinal de igual funciona como um operador com dois efeitos:

- O valor da expressão à direita do sinal é atribuído à variável indicada à esquerda;
- Essa operação resulta no valor atribuído.

Embora possa parecer estranho em um primeiro momento, C admite que um operador de atribuição possa ser usado em qualquer lugar onde caiba uma expressão. O programa seguinte ilustra o uso desse recurso.

```
#include <stdio.h>

int main(void) {
    int valor1, valor2;

    valor2 = 10 * (valor1 = 5);
    printf("valor1 = %d e valor2 = %d.\n", valor1, valor2);

    return 0;
}
```

```
valor1 = 5 e valor2 = 50.
```

Dica

O uso de atribuições em locais inusitados deve ser evitado. Não é só porque um determinado recurso funciona é que ele deva ser usado indiscriminadamente. A clareza do código deve sempre ser uma prioridade.

6 C: expressões relacionais e lógicas

Além de expressões aritméticas, há outros dois tipos de expressões relevantes, ambas tratadas neste capítulo: relacionais e lógicas.

6.1 Expressões relacionais

Uma expressão relacional é a que compara (i.e., relaciona) valores. Assim, para determinar de um valor é maior que zero, usa-se uma expressão relacional: `valor > 0`.

As expressões relacionais resultam em valores lógicos: verdadeiro ou falso. Em C, é possível introduzir a noção de algumas expressões relacionais por meio do programa seguinte.

```
/*
Ilustração de expressões relacionais
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
    printf("Digite dois valores inteiros: ");
    fgets(entrada, sizeof entrada, stdin);
    int valor1, valor2;
    sscanf(entrada, "%d%d", &valor1, &valor2);

    printf("%d == %d? %d.\n", valor1, valor2, valor1 == valor2);
    printf("%d != %d? %d.\n", valor1, valor2, valor1 != valor2);
    printf("%d < %d?  %d.\n", valor1, valor2, valor1 < valor2);
    printf("%d > %d?  %d.\n", valor1, valor2, valor1 > valor2);
    printf("%d <= %d? %d.\n", valor1, valor2, valor1 <= valor2);
    printf("%d >= %d? %d.\n", valor1, valor2, valor1 >= valor2);

    return 0;
}
```

```
Digite dois valores inteiros: 27 12
27 == 12? 0.
27 != 12? 1.
27 < 12?  0.
27 > 12?  1.
27 <= 12? 0.
27 >= 12? 1.
```

O resultado de uma operação relacional é um valor inteiro (tipo `int`, escrito com `%d`), podendo ser apenas dois valores: 0 se a condição é falsa ou 1, se for verdadeira. Com essa informação é possível interpretar corretamente a saída produzida pelo programa, embora essa saída não seja apropriada para um usuário comum.

Outro ponto a observar observar são os operadores relacionais em si. Alguns são de fácil compreensão, outros nem tanto. A Tabela 6.1 apresenta os operadores relacionais da linguagem e seus significados.

Tabela 6.1: Operadores relacionais da linguagem C.

Operador	Significado
`==`	Igual
`!=`	Diferente
`<`	Menor que
`>`	Maior que
`<=`	Menor ou igual
`>=`	Maior ou igual

Pela ordem de precedência das operações, todas as expressões aritméticas são avaliadas antes dos operadores relacionais. Desse modo, ao se escrever `2 * valor1 < 3 * valor2`, as duas multiplicações são realizadas antes da comparação, o que é bastante intuitivo.

Um problema prático interessante e simples é a verificação se alguém está "bem de vida". No Brasil, uma pessoa é considerada pertencente à classe A (a de maior renda) se tiver renda familiar mensal acima de 20 salários mínimos.

O Algoritmo 6.1 mostra uma solução que determina se uma renda familiar mensal fornecida pertence ou não à classe A. A verificação é feita por uma comparação de valores e é usada uma variável lógica para guardar o resultado. A saída esperada do algoritmo é verdadeiro ou falso apenas.

Algoritmo 6.1: Determinação de uma dada renda mensal familiar caracteriza o pertencimento à classe A.

Descrição: Determinação se uma renda familiar mensal é classificada como classe A no Brail; Classe A: 20 salários mínimos de
Requer: o valor da renda familiar
Assegura: VERDADEIRO se se enquadrar na classe A; FALSO se não

Obtenha *renda*
Calcule *é_classe_a* como *renda* > 28240,00 ▷ 20 × *R$1412 (janeiro/2024)*
Apresente *é_classe_a*

Este algoritmo pode ser implementado em C conforme o código seguinte.

```c
/*
Determinação se uma renda familiar mensal é classificada como classe A no
    Brasil; Classe A: 20 salários mínimos de R\$1.412,00 (valor de janeiro
    de 2024)
Requer: o valor da renda familiar
Assegura: verdadeiro se se enquadrar na classe A; falso se não
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite o valor da renda mensal familiar: ");
    fgets(entrada, sizeof entrada, stdin);
    double renda_familiar;
    sscanf(entrada, "%lf", &renda_familiar);

    _Bool eh_classe_a = renda_familiar > 20 * 1412.00;  // mínimo em 1/2024

    printf("Sua família é classe A? %d\n", eh_classe_a);

    return 0;
}
```

```
Digite o valor da renda mensal familiar: 15000
Sua família é classe A? 0
```

A variável `eh_classe_a` é declarada para guardar o resultado da comparação e ela é do tipo `_Bool`, que guarda um valor inteiro 0 ou 1, seguindo o padrão de representação da linguagem.

Dica

As variáveis que armazenam valores lógicos são usadas de forma diferenciada das numéricas, por exemplo. Em consequência disso, os identificadores usados também devem ser diferenciados.
Uma recomendação importante para as variáveis lógicas é que possuam um verbo no nome. Assim, bons identificadores para variáveis lógicas são `existe_taxa`, `possui_filhos` ou `eh_divisor`, por exemplo.
Vale notar que, embora `filhos` possa ser usada com significando se tem ou não filhos, não é óbvio que a variável seja mesmo um valor lógico, pois poderia conter, por exemplo, o número de filhos. E, assim, a clareza volta a ser um ponto relevante na escrita dos programas.

6.1.1 Afinal, `_Bool` ou `bool`?

A linguagem C, no início, não possuía um tipo lógico específico. Eram empregadas variáveis do tipo `int` para representar tanto os valores realmente inteiros (como idades ou contadores) quanto para valores lógicos. Assim, para o programador com alguma experiência, não é estranha essa mistura de significados nos códigos mais antigo e também nos novos.

Como não havia a necessidade de uma variável lógica, a palavra `bool` nunca foi reservada para a linguagem e poderia ser usada livremente nos programas. Em decorrência de uma posterior criação do tipo booliano, reservar uma palavra poderia implicar a reescrita em massa dos códigos já existentes. Dessa forma, a palavra-chave escolhida for `_Bool`.

Para os programas para os quais não haveria conflito, foi incluída à biblioteca padrão o arquivo de cabeçalho `stdbool.h`, o qual define `bool` como um novo nome para `_Bool`. Além do novo nome, também define `true` e `false` para serem usados no lugar de 1 e 0, respectivamente.

A modificação do programa para usar essas definições é pequena, bastando incluir o arquivo de cabeçalhos e modificar o tipo na declaração da variável.

```c
#include <stdbool.h>
```

```c
bool eh_classe_a = renda_familiar > 20 * 1412.00;
```

A forma de escrita usando `bool` parece, para o autor, a forma mais natural de declaração, pois segue um padrão visual similar ao dos outros tipos. Este será empregado ao longo do texto.

Antes de apresentar a versão final do código, agora com o cabeçalho `stdbool.h`, é interessante notar que existe em C uma construção que pode ser usada neste programa para melhorar a apresentação do resultado. Esse elemento é conhecido como condicional ternário.

Operador condicional ternário

expressão_lógica `?` *resultado_verdadeiro* `:` *resultado_falso*

Esse condicional, assim como os aritméticos e os relacionais, resultam em um valor dependente de seus operandos. Neste caso, há três operandos e dois operadores: `?` e `:`. Se o valor de *expressão_lógica*

for verdadeiro, então o resultado final da expressão é o valor *resultado_verdadeiro* e, caso contrário, o resultado é *resultado_falso*.

Segue um exemplo simples desse condicional ternário.

```
int a = 20;
int b = 12;

int maior = a > b ? a : b;
```

Na atribuição a `maior`, a *expressão_lógica* do condicional ternário é `a > b`. Se o resultado dessa expressão for verdadeiro, então o valor de `a` é o resultado atribuído a `maior`. Se o valor de `a` for menor ou igual ao de `b`, o resultado final será `b`.

Não há restrições dos tipos de dados que podem ser usados como resultado do condicional ternário e o programa usa cadeias de caracteres como resultado.

```
/*
Determinação se uma renda familiar mensal é classificada como classe A no
    Brasil; Classe A: 20 salários mínimos de R\$1.412,00 (valor de janeiro
    de 2024)
Requer: o valor da renda familiar
Assegura: verdadeiro se se enquadrar na classe A; falso se não
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    printf("Digite o valor da renda mensal familiar: ");
    fgets(entrada, sizeof entrada, stdin);
    double renda_familiar;
    sscanf(entrada, "%lf", &renda_familiar);

    bool eh_classe_a = renda_familiar > 20 * 1412.00;  // mínimo em 1/2024

    printf("Sua família %s classe A!\n", eh_classe_a ? "é" : "não é");

    return 0;
}
```

```
Digite o valor da renda mensal familiar: 30000
Sua família é classe A!
```

As modificações para o uso de `bool` foram feitas e, para o condicional ternário, se `eh_classe_a` for verdadeiro, o resultado é o texto `"é"` e, se não for, `"não é"`. Um desses dois valores ocupará o lugar do `%s` no texto do `printf`.

6.2 Expressões lógicas

Neste tópico serão detalhadas as expressões lógicas, as quais foram usadas nos exemplos anteriores mas não foram explicadas completamente.

Uma expressão lógica é uma expressão que resulta em verdadeiro ou falso pela combinação de expressões lógicas. Existem apenas três operadores lógicos: **e**, **ou** e **não**.

A combinação com o operador de disjunção **e** somente resulta em verdadeiro quando os dois operandos forem verdadeiros e, caso contrário, em falso. O conceito desse operador pode ser exemplificado por "há isenção de tarifa se o cliente possuir cartão de crédito e o volume de aplicações for superior a R$100.000,00". Nessa situação, apenas ter o cartão de crédito ou apenas ter aplicações suficientes

não dá direito ao desconto. O resultado somente é verdadeiro se ambas as condições forem satisfeitas simultaneamente.

O **ou** é o operador de conjunção, que resulta em verdadeiro se qualquer um de seus operadores for verdadeiro, sendo falso apenas quando ambos forem falsos. Por exemplo, "a pessoa tem direito a meia entrada no baile se for sócio do clube ou se for estudante". Este é o caso em que o resultado é falso apenas para os não sócios e os não estudantes.

O operador **não** é a negação, apenas invertendo o resultado. Não verdadeiro é falso e não falso é verdadeiro. Como exemplo, em "se não estiver chovendo, vou sair", a situação de chuva é verdadeira, o resultado é falso, o que impede a saída da pessoa.

Em C, a operação **e** é indicada pelo operador `&&`, **ou** usa `||` e **não** é o operador unário `!`. As combinações possíveis de resultados são apresentadas na Tabela 6.2, chamada de tabela verdade.

Tabela 6.2: Tabela verdade para os operadores `&&` (**e**), `||` (**ou**) e `!` (**não**).

`e1`	`e2`	`e1 && e2`	`e1 || e2`	`!e1`	`!e2`
falso	falso	falso	falso	verdadeiro	verdadeiro
falso	verdadeiro	falso	verdadeiro	verdadeiro	falso
verdadeiro	falso	falso	verdadeiro	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro	verdadeiro	falso	falso

Como os demais operadores, há também uma ordem de precedência quando são usados operadores lógicos. O de maior precedência é o `!`, seguido pelo `&&` e, por último, `||`. A quebra da precedência é feita com o uso de parênteses. Seguem exemplos de expressões e como são avaliadas (usando-se parênteses adicionais para reforçar a ordem).

```
a > b && a != 0 || b != 0  // ((a > b && a != 0) || b != 0)
```

```
a != 0 || b != 0 && a > b  // (a != 0 || (b != 0 && a > b))
```

```
// assumindo-se v1 e v2 como variáveis com valores lógicos
!v1 && v2  // ((!v1) && v2)
```

```
!v1 && !v2  // ((!v1) && (!v2))
```

Curiosidade

Existe uma operação chamada **ou exclusivo**, que diferentemente do **ou**, resulta verdadeiro se apenas um dos dois operandos for verdadeiro, mas não os dois. Para se obter esse resultado, a seguinte expressão pode ser usada.

```
// assumindo-se v1 e v2 como variáveis com valores lógicos
!(v1 && v2) && (v1 || v2)
```

A opção que segue é também equivalente.

```
(v1 && !v2) || (!v1 && v2)
```

Note-se que os parênteses nesta última expressão servem apenas para ênfase e são desnecessários dada a ordem de precedência dos operadores.

6.2.1 Relações matemáticas ternárias e sucessivas

No cotidiano é comum usar expressões ternárias como $0 < x < 10$ para indicar que x está em um determinado intervalo. Em C é preciso cuidado com tais expressões, pois cada expressão relacional resulta em um valor inteiro.

Por exemplo, a expressão `0 < valor < 10` é avaliada na seguinte ordem, considerando-se `valor` contendo 20:

1. `(0 < valor) < 10`: avaliação que ocorre da esquerda para a direita;
2. `1 < 10`: substituição do resultado intermediário na expressão;
3. `1`: o resultado final é verdadeiro.

Porém $0 < 20 < 10$ é esperado que resulte em falso.

Expressões ternárias em C não são possíveis e, assim, devem ser escritas como uma combinação de expressões binárias. A expressão deve ser escrita: `0 < valor && valor < 10`, sendo que o `&&` significa "e". O modo com que as expressões relacionais são avaliadas e o uso de valores inteiros como resultado levam à necessidade de que expressões sejam escritas, muitas vezes, de forma extensa.

Estendendo esse conceito, matematicamente se pode escrever $a = b = c = d$, por exemplo. Em C, a expressão requer expressões relacionais isoladas combinadas com operadores lógicos. Assim, `a == b && b == c & c == d` é uma expressão equivalente. Seguem exemplos.

```
a == b && b == c && c == d  // a = b = c = d
a < b && b < c && c < d     // a < b < c < d
a == b && b < c             // a = b < c
```

6.2.2 Comparações com vários valores

Outra questão que surgem com frequência é a verificação de uma variável com uma série de valores. Este é o caso, por exemplo, para verificar se uma variável inteira `mes` é um dos meses com 31 dias, ou seja, se $m \in \{1, 3, 5, 7, 8, 10, 12\}$. Não é incomum que programados iniciantes tentem fazer essa comparação como se segue.

```
mes == 1 || 3 || 5 || 7 || 8 || 10 || 12  // incorreto!
```

Essa expressão, pelas regras da linguagem, primeiro avalia se `mes == 1` e esse resultado é 0 ou 1. A próxima comparação verifica, portanto, `0 || 3` ou `1 || 3`, o que resulta em 1, qualquer que seja o valor obtido na primeira comparação. É importante lembrar que o valor falso está associado ao 0, enquanto qualquer valor não nulo é verdadeiro. A expressão acima significa, na prática, "mês igual a 1 ou verdadeiro ou verdadeiro ou verdadeiro...", o que é sempre verdadeiro, ou seja, 1.

O ajuste necessário para essa expressão é individualizar as comparações e a comparação seguinte produz o resultado desejado.

```
mes == 1 || mes == 3 || mes == 5 || mes == 7 ||  mes == 8 ||
    mes == 10 || mes == 12  // OK!
```

Dica

Elaborar expressões lógicas requer prática e atenção. Nem sempre as verificações são óbvias e testar a avaliação da expressão passo a passo pode ser uma estratégia muito interessante ou até necessária.

6.3 Ordem de precedência entre operadores relacionais e lógicos

A linguagem C estabelece um padrão curioso (e potencialmente confuso) para ordem de precedência dos operadores relacionais e lógicos. A regra geral é que operações relacionais sejam feitas antes das lógicas, ou seja, comparações antes das combinações com **e** e **ou**. Isso é verdade, exceto para o **não**. Em C, o operador `!` tem precedência sobre os operadores relacionais.

Com exemplo o trecho de código seguinte pode ser considerado.

```
int a = 1;
int b = 2;
int c = 0;

bool r = a > b && ! b >= c;
```

É importante observar que, embora `b` tenha valor 2, ao ser avaliado `! b > c` a primeira operação é `!b`. Como todo valor diferente de zero é verdadeiro, `!b` é falso e, portanto, igual a 0. Assim, a expressão se torna equivalente a `0 >= c`, o que resulta em verdadeiro, uma vez `c` também é zero.

A ordem é, portanto:

1. O operador `!`;
2. Os operadores relacionais;
3. O operador `&&`;
4. O operador `||`.

6.4 Exemplos práticos

Esta seção cobre exemplos diversos de implementações que usam expressões relacionais e lógicas em problemas diretos, com maior ou menor grau de complexidade.

6.4.1 Verificação de cédulas válidas

Este é um exemplo para determinar se um determinado valor é ou não um valor de cédula válido. Atualmente no Brasil, há cédulas e moedas usadas sendo usada cotidianamente no comércio. As cédulas em papel possuem valores de face de 2, 5, 10, 20, 50, 100 e 200 reais.

Assim, é proposto o Algoritmo 6.2 para determinar se um valor qualquer é ou não o valor de face de uma cédula brasileira.

Algoritmo 6.2: Determinação se um valor é um valor de face de uma cédula brasileira.

Descrição: Verificação se um dado valor inteiro qualquer é ou não um valor de face válido entre as cédulas do Brasil
Requer: um valor inteiro qualquer
Assegura: Verdadeiro se o valor corresponder a uma das cédulas ou Falso se não corresponder

Obtenha *valor_de_face*
Faça *é_válido* verdadeiro se e somente se $valor_de_face \in \{2, 5, 10, 20, 50, 100, 200\}$ ▷ *senão é falso*
Apresente *é_válido*

O programa seguinte codifica esse algoritmo.

```c
/*
Verificação se um dado valor inteiro qualquer é ou não um valor de face
    válido entre as cédulas do Brasil
Requer: um valor inteiro qualquer
Assegura: verdadeiro se o valor corresponder a uma das cédulas; falso
    caso contrário
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    printf("Digite um valor inteiro para verificação de cédula: ");
    fgets(entrada, sizeof entrada, stdin);
    int valor_candidato;
    sscanf(entrada, "%d", &valor_candidato);

    bool eh_valido = valor_candidato == 2 || valor_candidato == 5 ||
                     valor_candidato == 10 || valor_candidato == 20 ||
                     valor_candidato == 50 || valor_candidato == 100 ||
                     valor_candidato == 200;

    printf("O valor %d é %s.\n", valor_candidato,
        eh_valido ? "válido" : "inválido");

    return 0;
}
```

```
Digite um valor inteiro para verificação de cédula: 100
O valor 100 é válido.
```

6.4.2 Verificação de aprovação em disciplina

Considerando-se que a aprovação de um aluno em uma disciplina exija que ele tenha média maior ou superior a 6 e que sua frequência não seja inferior a 75%, o problema que deve ser resolvido é determinar se um aluno foi ou não aprovado. Para determinar média final, há duas notas e deve ser calculada sua média. Para a frequência, estão disponíveis o número de faltas e o número total de aulas. Assim, é proposto como solução o Algoritmo 6.3.

Algoritmo 6.3: Verificação da aprovação ou reprovação de um aluno em função de notas e frequência.

Descrição: Determinação de aprovação em disciplina em função da média de duas notas e frequência baseada no número de faltas e número de aulas
Requer: duas notas de provas, número de faltas e número de aulas
Assegura: A média, a frequência e Verdadeiro ou Falso conforme aprovado ou reprovado

Obtenha $nota_1$ e $nota_2$
Calcule *média* como $\dfrac{nota_1 + nota_1}{2}$
Calcule *tem_média* como $média \geq 6$

Obtenha n_{faltas} e n_{aulas}
Calcule *frequência* $\dfrac{n_{aulas} - n_{faltas}}{n_{aulas}}$
Calcule *tem_pouca_frequência* como $frequência < 0{,}75$

Calcule *tem_aprovação* como *tem_média* e não *tem_pouca_frequência*
Apresente *média*, *frequência* e *tem_aprovação*

A implementação é apresentada na sequência.

```
/*
Determinação de aprovação em disciplina em função da média de duas notas e frequência baseada no número de fal
Requer: duas notas de provas, número de faltas e número de aulas
Assegura: A média, a frequência e verdadeiro/falso conforme aprovado ou reprovado
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    // Média
    printf("Digite as duas notas de provas: ");
    fgets(entrada, sizeof entrada, stdin);
    double nota1, nota2;
    sscanf(entrada, "%lf%lf", &nota1, &nota2);
    double media = (nota1 + nota2) / 2;
    bool tem_media = media >= 6;

    // Frequência
    printf("Digite o número de faltas e o de aulas: ");
    fgets(entrada, sizeof entrada, stdin);
    int numero_faltas, numero_aulas;
    sscanf(entrada, "%d%d", &numero_faltas, &numero_aulas);
    double frequencia = (double) (numero_aulas - numero_faltas) / numero_aulas;
    bool tem_pouca_frequencia = frequencia < 0.75;

    // Aprovação/reprovação e resultado
    bool tem_aprovacao = tem_media && !tem_pouca_frequencia;
    printf("Média: %.1f; frequência: %.1f%%.\n", media, 100 * frequencia);
    printf("Situação: %s.\n", tem_aprovacao ? "aprovado" : "reprovado");

    return 0;
}
```

```
Digite as duas notas de provas: 5.8 9.1
Digite o número de faltas e o de aulas: 8 30
Média: 7.4; frequência: 73.3%.
Situação: reprovado.
```

Este programa usa variáveis lógicas para as diversas condições. Um detalhe interessante é a variável `tem_pouca_frequencia`, a qual indica que a frequência não é suficiente para aprovação. Na verificação final, é usado o operador **não** para negar essa condição e conceder a aprovação: `!tem_pouca_frequencia` (i.e., "não tem pouca frequência").

O operador `!` é pouco usado juntamente com expressões relacionais, pois escrever `a <= b` é melhor que `!(a > b)`. Por outro lado, se existem uma variável `detectou_problema`, é mais natural negar escrevendo `detectou_problema` ou `!detectou_problema`.

Dica

Operadores lógicos nunca devem ser comparados com os valores `true` e `false`, como exemplificado na sequência.

```
tem_aprovacao ? "aprovado" : "reprovado"           // bom!
tem_aprovacao == true ? "aprovado" : "reprovado"   // ruim e desnecessário!
tem_aprovacao != false ? "aprovado" : "reprovado"  // pior ainda!
```

Pode-se considerar que comparar uma variável com `true` ou `false` faz tanto sentido como escrever `(a > b) == true`.

6.4.3 Verificação de existência de triângulo

Para ilustrar tanto as expressões como o uso de valores lógicos, o problema de verificar se três segmentos de reta podem ser os lados de um triângulo é apresentado. Um triângulo não pode ser formado caso um dos segmentos seja maior ou igual à soma dos outros dois, situação em que o triângulo "não fecha". Seguem a solução no Algoritmo 6.4 e sua implementação em C.

Algoritmo 6.4: Verificação se três segmentos de reta podem ou não formar um triângulo

Descrição: Verificação se três segmentos de reta com determinados comprimentos podem ou não formar um triângulo
Requer: os comprimentos l_1, l_2 e l_3 dos segmentos de reta
Assegura: VERDADEIRO se podem formar um triângulo; FALSO caso contrário

Obtenha os comprimentos l_1, l_2 e l_3
Calcule $\triangle_{válido}$ como $l_1 < l_2 + l_3$ e $l_2 < l_1 + l_3$ e $l_3 < l_1 + l_2$ ▷ *verdadeiro ou falso*
Apresente $\triangle_{válido}$

```
/*
Verificação se três segmentos de reta com determinados comprimentos podem
    ou não formar um triângulo
Requer: os comprimentos $l_1$, $l_2$ e $l_3$ dos segmentos de reta
Assegura: Verdadeiro se podem formar um triângulo; falso caso contrário
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    printf("Digite os comprimentos dos segmentos de reta: ");
    fgets(entrada, sizeof entrada, stdin);
    double lado1, lado2, lado3;
    sscanf(entrada, "%lf%lf%lf", &lado1, &lado2, &lado3);

    bool triangulo_eh_valido = lado1 < lado2 + lado3 && lado2 < lado1 + lado3 &&
            lado3 < lado1 + lado2;

    printf("Triângulo %s.\n", triangulo_eh_valido ? "válido" : "inválido");

    return 0;
}
```

```
Digite os comprimentos dos segmentos de reta: 3 4 5
Triângulo válido.
```

Neste programa, a variável `triangulo_eh_valido` é usada como uma variável lógica.

6.4.4 Verificação de validade de uma data

Como exemplo com expressões lógicas, segue um algoritmo para determinar se valores para dia, mês e ano formam uma data válida (Algoritmo 6.5). Os valores formam uma data válida se o valor do dia estiver dentro do intervalo do mês e o ano for diferente de zero[1]. A solução desconsidera anos bissextos, de forma que fevereiro considera apenas 28 dias. Também considera que valores negativos para um ano indicam AC (antes de Cristo).

[1] O calendário da era cristã (*Anno Domini*, ou ano do Senhor) se inicia com o ano 1 DC e anos antes de Cristo se iniciam com 1 AC. Não há ano zero.

Algoritmo 6.5: Determinação se valores inteiros para dia, mês e ano formam uma data válida.

Descrição: Determinação se valores inteiros para dia, mês e ano formam uma data válida, desconsiderando anos bissextos e entendendo anos negativos como AC
Requer: *dia*, *mês* e *ano* inteiros
Assegura: Verdadeiro se os valores formarem uma data válida; Falso caso contrário

Obtenha *dia*, *mês* e *ano*
Calcule *data_válida* com a expressão $ano \neq 0$ e $dia > 0$ e $mês > 0$ e $(dia \leq 28$ e $mês == 2$ ou $dia \leq 31$ e $mês \in \{1, 3, 5, 7, 8, 10, 12\}$ ou $dia \leq 30$ e $mês \in \{4, 6, 9, 11\})$
Apresente o valor de *data_válida*

```
/*
Determinação se valores inteiros para dia, mês e ano formam uma data
válida,desconsiderando anos bissextos e entendendo anos negativos como AC
Requer: valores inteiros para dia, mês e ano
Assegura: verdadeiro se a data é válida ou falso caso contrário
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    printf("Digite os valores para dia, mês e ano: ");
    fgets(entrada, sizeof entrada, stdin);
    int dia, mes, ano;
    sscanf(entrada, "%d%d%d", &dia, &mes, &ano);

    bool data_valida = dia > 0 && mes > 0 && ano != 0 && (
            dia <= 28 && mes == 2 ||
            dia <= 31 && (mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
                mes == 8 || mes == 10 || mes == 12) ||
            dia <= 30 && (mes == 4 || mes == 6 || mes == 9 || mes == 11)
        );

    printf("%02d/%02d/%04d é %s.\n", dia, mes, ano,
        data_valida ? "válida" : "inválida");

    return 0;
}
```

```
Digite os valores para dia, mês e ano: 30 2 2050
30/02/2050 é inválida.
```

Embora a ordem de precedência dos operadores permita escrever as expressões usando apenas os parênteses somente necessário, é usual que expressões mais longas usem o recurso dos parênteses para dar maior clareza à expressão. Um desses casos é envolver o `&&` com parênteses, mesmo sabendo-se que o `||` será avaliado posteriormente. O programa de verificação de datas é escrito lançando mão dessa estratégia.

```
/*
Determinação se valores inteiros para dia, mês e ano formam uma data
    válida,desconsiderando anos bissextos e entendendo anos negativos
    como AC
Requer: valores inteiros para dia, mês e ano
Assegura: verdadeiro se a data é válida ou falso caso contrário
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];
```

```
    printf("Digite os valores para dia, mês e ano: ");
    fgets(entrada, sizeof entrada, stdin);
    int dia, mes, ano;
    sscanf(entrada, "%d%d%d", &dia, &mes, &ano);

    bool data_valida = dia > 0 && mes > 0 && ano != 0 && (
            (dia <= 28 && mes == 2) ||
            (dia <= 31 && (mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
                mes == 8 || mes == 10 || mes == 12)) ||
            (dia <= 30 && (mes == 4 || mes == 6 || mes == 9 || mes == 11))
        );

    printf("%02d/%02d/%04d é %s.\n", dia, mes, ano,
        data_valida ? "válida" : "inválida");

    return 0;
}
```

```
Digite os valores para dia, mês e ano: 18 12 1892
18/12/1892 é válida.
```

6.4.5 Mais sobre valores lógicos em C

Como já exposto no início do capítulo, se um valor for igual a zero ele é considerado falso e se for diferente de zero é verdadeiro. A questão é que, na linguagem C, isso é válido para qualquer valor e não somente os do tipo `bool` e inteiros.

Para exemplificar, o programa abaixo pode ser considerado.

```
/*
Exemplo de programa com negação de valor double
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    printf("Digite um valor real: ");
    fgets(entrada, sizeof entrada, stdin);
    double d;
    sscanf(entrada, "%lf", &d);

    bool r = !d;

    printf("r é %s.\n", r ? "verdadeiro" : "falso");
    return 0;
}
```

```
Digite um valor real: 0.0
r é verdadeiro.
```

Se o valor digitado para `d` for igual a zero, `!d` é 1; se for diferente de zero, `r` receberá 0. Para a linguagem, é irrelevante que `d` seja `double`.

Segue mais um exemplo.

```
/*
Exemplo de programa com negação de valor double
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];
```

```
    printf("Digite um valor real: ");
    fgets(entrada, sizeof entrada, stdin);
    double d;
    sscanf(entrada, "%lf", &d);

    printf("O valor digitado é %s zero.\n", d ? "diferente de" : "igual a");
    return 0;
}
```

```
Digite um valor real: 5.2
O valor digitado é diferente de zero.
```

Neste programa, o condicional ternário é escrito `d ? "diferente de" : "igual a"`. Se `d` for zero, a expressão é assumida como falsa; se não for, é verdadeira.

> **Cuidado**
>
> Não é uma boa prática usar um valor que não seja lógico como se assim fosse. A escrita do código deve deixar claro o que é o que.
> Por exemplo, mesmo que `2 * i - 1 ? "Ok" : "Não ok"` funcione, resultando em `"Ok"` sempre que `2 * i - 1` for diferente de zero, deve-se sempre escrever `2 * i - 1 != 0 ? "Ok" : "Não ok"`, já que a operação aritmética não é, em si, uma expressão lógica.

Parte II

Controle de fluxo simples

7 Execução condicional com `if`

A maioria dos programas requer que alguns comandos apenas sejam executados em algumas condições e isso é chamado, naturalmente, de execução condicional.

A principal estrutura na linguagem C para determinar se determinados comandos são ou não executados é o `if`. Esta estrutura é o assunto deste capítulo.

7.1 A estrutura condicional `if`

Para introduzir a estrutura condicional é apresentado um programa que implementa o Algoritmo 7.1.

Algoritmo 7.1: Apresentação de dois valores reais em ordem não decrescente.

Descrição: Apresentação de dois valores reais quaisquer em ordem não decrescente.
Requer: $v_1, v_2 \in \mathbb{R}$
Assegura: $v_1 \leq v_2$

Obtenha v_1 e v_2
se $v_2 < v_1$ **então**
  Troque o valor v_1 com o de v_2
fim se
Apresente v_1 e v_2

A codificação em C deste algoritmo pode ser feita como se segue.

```
/*
Apresentação de dois valores em ordem não decrescente
Requer: dois valores reais v1 e v2
Assegura: v1 <= v2
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite dois valores reais: ");
    fgets(entrada, sizeof entrada, stdin);
    double v1, v2;
    sscanf(entrada, "%lf%lf", &v1, &v2);

    if(v2 < v1) {
        double temporario = v1;
        v1 = v2;
        v2 = temporario;
    }

    printf("Valores em ordem não decrescente: %g e %g.\n", v1, v2);

    return 0;
}
```

```
Digite dois valores reais: 4.5 1.3
Valores em ordem não decrescente: 1.3 e 4.5.
```

A estratégia da implementação é a mesma do algoritmo, que opta por fazer a troca dos valores (e isso requer o uso de uma variável temporária auxiliar). As três instruções que realizam a troca somente são executadas quando `v2` for menor que `v1` e ignoradas se isso não for verdade

É importante ressaltar que as chaves agrupam as três atribuições do `if`, de forma que todos os comandos ficam condicionados. O uso das chaves para agrupar comandos cria um comando composto, o qual pode ser usado no lugar de qualquer comando simples.

Em C, a estrutura básica do `if` se estabelece conforme destacado na sequência.

> **Estrutura `if` básica**
>
> `if (` *condição* `)` *comando*

A *condição* é uma expressão que deve resultar em verdadeiro ou falso. Somente se a condição for verdadeira *comando* é executado. O *comando*, por sua vez, pode ser tanto um comando simples (terminado com ponto e vírgula) ou um comando composto (usando chaves).

Seguem dois exemplos válidos de estruturas `if`: a primeira usa um comando simples e, na segunda, um comando composto contendo apenas um comando simples. É interessante observar que o ponto e vírgula ocorre apenas no final do comando simples, enquanto comandos compostos não usam esse terminador depois do fecha chaves.

```
if(valor < 0)
    valor = -valor; // torna positivo
```

```
if(valor < 0) {
    valor = -valor; // torna positivo
}
```

7.2 `if` + `else`

A cláusula `else` pode ser adicionada ao `if` para indicar uma ação a ser feita caso a condição de teste seja avaliada como falsa.

> **Estrutura `if` completa**
>
> `if (` *condição* `)` *comando_verdade* `else` *comando_falso*

Como exemplo, considere o problema de determinar o peso (massa) ideal de uma pessoa baseada em seu sexo biológico. Para o sexo feminino, o peso em quilogramas é dado pela expressão $62{,}1h - 44{,}7$, sendo h a altura da pessoa em metros; para o masculino, o cálculo é $72{,}7h - 58$. Seguem a solução algorítmica e uma implementação em C para esse problema.

```
/*
Cálculo estimado da massa ideal de uma pessoa baseada em seu sexo biológico
    e sua altura
Requer: o sexo biológico (masculino ou feminino) e a altura em metros
Assegura: a massa ideal da pessoa apresentada (kg)
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
```

Algoritmo 7.2: Estimativa do peso ideal conforme sexo biológico e altura.

Descrição: Cálculo estimado da massa ideal de uma pessoa baseada em seu sexo biológico e sua altura
Requer: o sexo biológico (masculino ou feminino) e a altura em metros
Assegura: a massa ideal da pessoa

Obtenha o sexo biológico e a altura
se sexo for feminino **então**
  Calcule *massa_ideal* como $62{,}1h - 44{,}7$ ▷ *feminino*
senão
  Calcule *massa_ideal* como $72{,}7h - 58$ ▷ *masculino*
fim se
Apresente *massa_ideal*

```
    printf("Digite o sexo (M ou F) e a altura em quilogramas: ");
    fgets(entrada, sizeof entrada, stdin);
    char sexo;
    double altura;
    sscanf(entrada, "%c%lf", &sexo, &altura);

    double massa_ideal;
    if(sexo == 'F')
        massa_ideal = 62.1 * altura - 44.7;  // feminino
    else
        massa_ideal = 72.7 * altura - 58;  // masculino

    printf("Massa ideal: %.1fkg.", massa_ideal);

    return 0;
}
```

```
Digite o sexo (M ou F) e a altura em quilogramas: F 1.68
Massa ideal: 59.6kg.
```

Neste há dois caminhos possíveis: o cálculo como sexo biológico feminino ou masculino. Pela especificação do problema, apenas `M` ou `F` são entradas válidas, o que permitiu associar ao `else`, inequivocamente, o valor `M`. Em C, vale o destaque que `=` é usado para a atribuição, sendo que a comparação de igualdade é feita com `==`.

Em relação ao `if`, tanto no caso verdadeiro quanto no falso foram inseridos comandos simples.

Como exemplo adicional, um algoritmo para classificar um triângulo como equilátero, escaleno ou isósceles a partir do comprimento de seus lados é apresentado no Algoritmo 7.3.

Em C, a codificação pode ser expressa na forma seguinte.

```
#include <stdio.h>
/*
Classificação de um triângulo em relação aos comprimentos de seus lados
Requer: Os comprimentos dos lados de um triângulo válido
Assegura: uma classificação em equilátero, escaleno ou isósceles
*/
int main(void) {
    char entrada[160];

    printf("Digites os comprimentos dos lados de um triângulo: ");
    fgets(entrada, sizeof entrada, stdin);
    double lado1, lado2, lado3;
    sscanf(entrada, "%lf%lf%lf", &lado1, &lado2, &lado3);

    char *classificacao;
    if(lado1 == lado2 && lado2 == lado3)
        classificacao = "equilátero";
```

Algoritmo 7.3: Classificação de um triângulo em relação aos comprimentos de seus lados

Descrição: Classificação de um triângulo em relação aos comprimentos de seus lados
Requer: Os comprimentos l_1, l_2 e l_3 dos lados de um triângulo válido
Assegura: uma classificação em equilátero, escaleno ou isósceles

Obtenha os valores dos lados l_1, l_2 e l_3
se $l_1 = l_2 = l3$ **então**
  Faça *classificação* igual a equilátero
senão se $l_1 = l_2$ ou $l_2 = l_3$ ou $l_1 = l_3$ **então**
  Faça *classificação* igual a isósceles
senão
  Faça *classificação* igual a escaleno
fim se
Apresente *classificação*

```
    else if(lado1 == lado2 || lado2 == lado3 || lado1 == lado3)
        classificacao = "escaleno";
    else
        classificacao = "isósceles";

    printf("O triângulo é %s.\n", classificacao);

    return 0;
}
```

```
Digites os comprimentos dos lados de um triângulo: 9.1 4.3 9.1
O triângulo é escaleno.
```

O programa usa comandos simples para as atribuições, o que dispensa os comandos compostos com as chaves. Na lógica da solução inicialmente verificada a hipótese dos três lados iguais (equilátero). Caso essa verificação dê falso, então há pelo menos um lado diferente e isso leva ao segundo `if`, que verifica a hipótese de haver algum par de lados de mesmo comprimento.

Este programa, além do `if` e algumas operações lógicas com **e** e **ou**, também usa uma variável `classificacao` declarada como do tipo `char *`. Quando são feitas atribuições como as usadas no exemplo a esse tipo de variável, é feita apenas uma referência à constante e não uma atribuição convencional, na qual uma cópia do texto é guardada na variável. A manipulação de cadeias de caracteres em C é, para se bem dizer, chata e difícil. Como ela não é tão natural, acaba exigindo conhecimento de outros elementos da linguagem. O Capítulo 16 aborda esse tema com mais detalhes.

Dica

Nas comparações com valores `double` resultantes de expressões aritméticas, nunca se deve verificar pela igualdade. Os valores reais são armazenados com precisão limitada. É conveniente o uso uma tolerância nessas comparações.

```c
/*
Ilustração do problema de precisão nas operações com double
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    double a = 40.96;
    double b = 6.4 * 6.4;  // 40.96

    // Opção ruim
    if(a == b)
        printf("%g == %g.\n", a, b);
    else {
        printf("%g != %g.  :-(\n", a, b);
        printf("Diferença: %g.\n\n", a - b);
    }

    // Opção melhor, contornando o problema
    if(fabs(a - b) < 1e-5)  // cinco casas decimais...
        printf("%g == %g. (o suficiente!)\n", a, b);
    else {
        printf("%g != %g.  :-(\n", a, b);
        printf("Diferença: %g.\n\n", a - b);
    }

    return 0;
}
```

```
40.96 != 40.96.  :-(
Diferença: -7.10543e-15.

40.96 == 40.96. (o suficiente!)
```

Neste programa, se a diferença for menor que 0,00001 (segundo if), então os valores são considerados iguais. A tolerância, é claro, depende do contexto do problema.

Como um último exemplo, segue o algoritmo para o cálculo das raízes reais de uma equação de segundo grau $ax^2 + bx + c = 0$ dados seus coeficientes (Algoritmo 1.1, apresentado no Capítulo 1).

Sua codificação em C pode ser apresentada conforme se segue.

```c
/*
Cálculo e apresentação das raízes reais de uma equação de segundo grau na
    forma ax^2 + bx + c = 0
Requer: Os coeficientes a, b e c da equação
Assegura: as raízes reais da equação; ou mensagem que a equação é
    inválida; ou mensagem que não há raízes reais
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    printf("Digite os valores de a, b e c da equação: ");
    fgets(entrada, sizeof entrada, stdin);
    double a, b, c;
    sscanf(entrada, "%lf%lf%lf", &a, &b, &c);

    if(a == 0)
        printf("Não é equação do segundo grau (a = 0).\n");
    else {
        double discriminante = pow(b, 2) - 4 * a * c;
        if(discriminante < 0)
            printf("A equação não possui raízes reais.\n");
        else if(discriminante < 1.0e-10) {  // tolerância: < 1.0e-10 é "zero"
            // Uma raiz real
            double x = -b / (2 * a);
            printf("Uma raiz: %g.\n", x);
```

```
        }
        else {
            // Duas raízes reais
            double x1 = (-b - sqrt(discriminante)) / (2 * a);
            double x2 = (-b + sqrt(discriminante)) / (2 * a);
            printf("Raízes reais: %g e %g.\n", x1, x2);
        }
    }

    return 0;
}
```

```
Digite os valores de a, b e c da equação: 1 -5.9 7.14
Raízes reais: 1.7 e 4.2.
```

7.3 De quem é esse `else`?

Um ponto relevante quando se usa `if` dentro de `if` é a quem pertence um determinado `else`. Para considerar esse problema é apresentado um código em C.

```
/*
Ilustração do problema do else pendente
*/
#include <stdio.h>

int main(void) {
    int i = 0;
    int j = 0;

    int k = 5;

    if(k > 10)
        if (k > 20)
            i = 10;
    else
        j = 10;

    printf("i = %d e j = %d.\n", i, j);

    return 0;
}
```

```
i = 0 e j = 0.
```

A intenção do código seria fazer `j = 10` para valores de `k` menores ou iguais a 10. Porém, apesar da organização visual sugerir que o else pertence ao `if(k > 10)`, ele se liga ao `if(k > 20)`. Assim, independentemente da disposição dos comandos no codigo fonte, o programa será sempre interpretado como se segue. E, neste caso, o primeiro `if` não possui `else`.

```
/*
Ilustração do problema do else pendente
*/
#include <stdio.h>

int main(void) {
    int i = 0;
    int j = 0;

    int k = 5;

    if(k > 10)
        if (k > 20)
            i = 10;
        else
```

```
            j = 10;

    printf("i = %d e j = %d.\n", i, j);

    return 0;
}
```

O `else` sempre se ligará ao `if` mais próximo da sequência das instruções. Para resolver esse problema do exemplo, é preciso isolar o `if` mais interno, colocando-o em um bloco, formando um comando composto.

O resultado é apresentado na sequência.

```
/*
Ilustração do problema do else pendente
*/
#include <stdio.h>

int main(void) {
    int i = 0;
    int j = 0;

    int k = 5;

    if(k > 10) {
        if (k > 20)
            i = 10;
    }
    else
        j = 10;

    printf("i = %d e j = %d.\n", i, j);

    return 0;
}
```

```
i = 0 e j = 10.
```

É ainda interessante notar que, caso o `if(k > 20)` tivesse seu próprio `else`, a ambiguidade desapareceria e o comando composto seria desnecessário.

```
/*
Ilustração do problema do else pendente
*/
#include <stdio.h>

int main(void) {
    int i = 0;
    int j = 0;

    int k = 15;

    if(k > 10)
        if (k > 20)
            i = 10;
        else
            i = 20;
    else
        j = 10;

    printf("i = %d e j = %d.\n", i, j);

    return 0;
}
```

```
i = 20 e j = 0.
```

Dica

A manutenção da organização visual do código com as indentações corretas ajuda na identificação dos erros.

8 Execução condicional com `switch`

A linguagem C provê, além do `if`, uma segunda estrutura para execução condicional: o `switch`.

8.1 Entendendo o `switch`

O `switch` é uma estrutura da linguagem usada para estabelecer uma sequência de instruções.

Estrutura `switch`

`switch` (*expressão*) *comando*

A *expressão* é qualquer expressão que resulte em um valor escalar, como `int`, `char` ou mesmo `bool`. Por sua vez, *comando* é um bloco delimitado com chaves com uma lista de comandos que serão executados condicionalmente.

O trecho de código seguinte ilustra a estrutura do `switch` com vários chamadas para a função `printf`. A variável `valor` pode ser um `int`, por exemplo.

```
switch(valor) {
    printf("A\n");
    printf("B\n");
    printf("C\n");
    printf("D\n");
    printf("E\n");
    printf("F\n");
    printf("G\n");
    printf("H\n");
    printf("I\n");
}
```

Esse código, do jeito que está apresentado, não terá nenhum de seus comandos executados. A especificação para que algo seja executado é feito por rótulos que indicam em que posição da sequência se inicia a execução.

Os rótulos são especificados com `case`.

Estrutura da rotulação com `case`

`case` *constante* :

Segue a lista de comandos com rótulos inseridos. Esses rótulos são as constantes 1, 3 e 10.

```
    switch (valor) {
        case 1:
            printf("A\n");
            printf("B\n");
            printf("C\n");
            printf("D\n");
        case 3:
            printf("E\n");
            printf("F\n");
            printf("G\n");
        case 10:
            printf("H\n");
```

```
            printf("I\n");
    }
```

Para essa organização, o funcionamento do `switch` avalia a expressão (variável `valor`) e inicia a execução da lista de comandos a partir do `case` em que houver igualdade.

Assim, se `valor` for igual a 1, todos os `printf` são executados; se for igual a 3, a execução se inicia no `printf("E\n")` e vai até o final; e se for igual a 10, somente H e I são escritos. Para qualquer outro valor, nada é escrito, pois não existe o rótulo indicando em que posição da lista começa a execução.

O programa seguinte contém a implementação completa do exemplo apresentado.

```
/*
Exemplo de escolha de execução com switch
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite um valor inteiro: ");
    fgets(entrada, sizeof entrada, stdin);
    int valor;
    sscanf(entrada, "%d", &valor);

    switch (valor) {
        case 1:
            printf("A\n");
            printf("B\n");
            printf("C\n");
            printf("D\n");
        case 3:
            printf("E\n");
            printf("F\n");
            printf("G\n");
        case 10:
            printf("H\n");
            printf("I\n");
    }

    return 0;
}
```

```
Digite um valor inteiro: 3
E
F
G
H
I
```

O conceito da estrutura `switch` é ter uma lista de comandos e, por meio dos rótulos, indicam em que posição da lista se inicia a execução.

Para ter um último exemplo de como essa estrutura de seleção funciona, é adicionada uma cláusula `default`, que indica que, se não houver nenhum rótulo coincidente, é nesse ponto que a execução se inicia.

```
/*
Exemplo de escolha de execução com switch
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

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

```
    fgets(entrada, sizeof entrada, stdin);
    int valor;
    sscanf(entrada, "%d", &valor);

    switch (valor) {
        case 1:
            printf("A\n");
            printf("B\n");
            printf("C\n");
            printf("D\n");
        case 3:
            printf("E\n");
            printf("F\n");
            printf("G\n");
        case 10:
            printf("H\n");
            printf("I\n");
        default:
            printf("Y\n");
            printf("Z\n");
    }

    return 0;
}
```

```
Digite um valor inteiro: 4
Y
Z
```

Talvez sejam raros os casos em que um problema consiga usar o `switch` conforme apresentado até o momento, visto que a lista de comandos é sempre executada até o final.

Dessa forma, é comum a inserção de uma interrupção na sequência de comandos e, para isso, é usada a instrução `break`. O exemplo seguinte é uma nova versão do programa, agora limitando até onde a lista executa.

Quando um `break` é encontrado, o `switch` é interrompido naquele ponto.

```
/*
Exemplo de escolha de execução com switch
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite um valor inteiro: ");
    fgets(entrada, sizeof entrada, stdin);
    int valor;
    sscanf(entrada, "%d", &valor);

    switch (valor) {
        case 1:
            printf("A\n");
            printf("B\n");
            printf("C\n");
            printf("D\n");
            break;
        case 3:
            printf("E\n");
            printf("F\n");
            printf("G\n");
            break;
        case 10:
            printf("H\n");
            printf("I\n");
            break;
        default:
            printf("Y\n");
            printf("Z\n");
```

```
    }

    return 0;
}
```

```
Digite um valor inteiro: 1
A
B
C
D
```

Nesta versão, a mais completa e usual, mostra como, dependendo de um dado valor, apenas uma lista específica de instruções é executada.

Para um exemplo mais prático, pode ser o problema de ler uma expressão com dois operandos e um operador aritmético simples e apresentar o resultado (Algoritmo 8.1).

Algoritmo 8.1: Cálculo de uma expressão aritmética simples a partir dos operandos e do operador.

Descrição: Realização de uma operação aritmética simples dados os operandos e o operador
Requer: dois operandos e um operador
Assegura: o resultado da operação ou mensagem se o operador for desconhecido

Obtenha $operando_1$, *operador* e $operando_2$

Faça *é_operador_válido* igual a VERDADEIRO
se *operador* for `+` **então**
  Calcule *resultado* como $operando_1 + operando_2$
senão se *operador* for `-` **então**
  Calcule *resultado* como $operando_1 - operando_2$
senão se *operador* for `*` **então**
  Calcule *resultado* como $operando_1 \cdot operando_2$
senão se *operador* for `/` **então**
  Calcule *resultado* como $operando_1 / operando_2$
senão
  Faça *é_operador_válido* igual a FALSO
fim se

se *é_operador_válido* **então**
  Apresente o *resultado*
senão
  Apresente mensagem de operador inválido
fim se

```
/*
Realização de uma operação aritmética simples dados os operandos e o
    operador
Requer: operando1, operador e operando2
Assegura: o resultado da operação ou mensagem se o operador for
    desconhecido
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    printf("Digite uma operação aritmética (sem espaços): ");
    fgets(entrada, sizeof entrada, stdin);
```

```
    double operando1, operando2;
    char operador;
    sscanf(entrada, "%lf%c%lf", &operando1, &operador, &operando2);

    double resultado;
    bool eh_operador_valido = true;
    switch (operador) {
        case '+':
            resultado = operando1 + operando2;
            break;
        case '-':
            resultado = operando1 - operando2;
            break;
        case '*':
            resultado = operando1 * operando2;
            break;
        case '/':
            resultado = operando1 / operando2;
            break;
        default:
            eh_operador_valido = false;
    }

    if (eh_operador_valido)
        printf("%g %c %g = %g.\n", operando1, operador, operando2, resultado);
    else
        printf("Operador não reconhecido.\n");

    return 0;
}
```

```
Digite uma operação aritmética (sem espaços): 1.75*-3.1
1.75 * -3.1 = -5.425.
```

8.2 Limitações do `switch`

Embora bastante útil, o `switch` serve apenas para comparações de igualdade, não permitindo verificações de intervalos, por exemplo.

Outra limitação é que a expressão usada tem que ser inteira ou `char`. Valores reais não podem ser usados.

Parte III

Organização geral do código

9 Escopo básico de declarações

Os códigos fonte em C permitem muitas declarações, como as de variáveis. Este capítulo trata da declaração de variáveis e onde são válidas, além de introduzir superficialmente outras declarações da linguagem.

9.1 Declarações e validade

O primeiro ponto é, em princípio, bastante intuitivo: para usar uma variável é preciso declará-la antes desse uso. Portanto, a validade de uma variável depende de onde, no código fonte, sua declaração aparece.

O programa seguinte pode ser considerado para exemplificar as validades.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite uma distância em metros: ");
    fgets(entrada, sizeof entrada, stdin);
    double em_metros;
    sscanf(entrada, "%lf", &em_metros);

    // Centímetros
    double em_centimetros = em_metros * 100;
    printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);

    // Pés
    double em_pes = em_metros * 3.281;
    printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);

    // Jardas
    double em_jardas = em_metros * 1.094;
    printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);

    // Centímetros
    double em_polegadas = em_metros * 39.37;
    printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);

    return 0;
}
// Fim do código fonte
```

```
Digite uma distância em metros: 300
> 300.0m = 30000.0cm
> 300.0m = 984.3 pés
> 300.0m = 328.2 jardas
> 300.0m = 11811.0 polegadas
```

Neste programa, a variável entrada existe desde a linha 9 e é válida até a linha 33. A variável `em_metros` inicia sua validade na linha 13, `em_pes` na linha 21, `em_jardas` na 25 e `em_polegadas` na 29 e, para todas, a validade termina na linha 33.

Note-se que, porém, que nenhuma variável é válida na linha 34, depois do encerramento do bloco do `main`. As variáveis são válidas apenas dentro do bloco onde foram declaradas. E é importante destacar que um bloco de comandos é aquele iniciado por `{` e finalizado por `}`.

> **Dica**
>
> É indicado que a declaração de uma variável seja feita o mais próximo possível de seu uso, o que promove clareza de padronização ao programa.

O programa que implementa o Algoritmo 7.1 é reproduzido na sequência.

```c
/*
Apresentação de dois valores em ordem não decrescente
Requer: dois valores reais v1 e v2
Assegura: v1 <= v2
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite dois valores reais: ");
    fgets(entrada, sizeof entrada, stdin);
    double v1, v2;
    sscanf(entrada, "%lf%lf", &v1, &v2);

    if(v2 < v1) {
        double temporario = v1;
        v1 = v2;
        v2 = temporario;
    }

    printf("Valores em ordem não decrescente: %g e %g.\n", v1, v2);

    return 0;
}
```

O ponto de destaque desse código é a variável `temporario` usada para troca de valores. Ela é declarada dentro de um bloco de comandos, formando o comando composto condicionado pelo `if`. Sua validade é efêmera, pois é válida apenas da linha 18 até a 21, quando o bloco é encerrado. A tentativa de referenciar `temporario` fora do bloco do `if` gera um erro de identificador não declarado.

Nesse ponto específico do programa, é relevante perceber que a utilidade de `temporario` é apenas local e não há qualquer necessidade de que seja declarada como válida em todo o bloco do `main`.

> **Dica**
>
> Váriáveis com uso localizado devem ser declaradas apenas dentro do bloco onde são úteis, evitando que tenham validade fora dessa abrangência.

9.1.1 Duplicidade de identificadores

Não é possível o uso do mesmo identificador no mesmo escopo de abrangência. Segue um exemplo simples que ilustra o problema.

```c
/*
Exemplo de declaração repetida de um identificador no mesmo escopo
*/
#include <stdio.h>

int main(void) {
```

```
    int valor = 10;
    printf("valor: %d.\n", valor);

    int valor = 100;
    printf("valor: %d.\n", valor);

    return 0;
}
```

```
main.c: In function ´main´:
main.c:10:9: error: redefinition of ´valor´
   10 |     int valor = 100;
      |         ^~~~~
main.c:7:9: note: previous definition of ´valor´ with type ´int´
    7 |     int valor = 10;
      |         ^~~~~
```

Há, porém, a possibilidade de que um mesmo identificador seja usado em um novo escopo, valendo a regra que sempre a declaração "mais local" é a válida.

```
/*
Exemplo de declaração repetida de um identificador em escopos diferentes
*/
#include <stdio.h>

int main(void) {
    int valor = 10;
    printf("valor: %d (antes do if).\n", valor);

    if (valor == 10){
        int valor = 100;
        printf("valor: %d (dentro do if).\n", valor);
    }

    printf("valor: %d (depois do if).\n", valor);

    return 0;
}
```

```
valor: 10 (antes do if).
valor: 100 (dentro do if).
valor: 10 (depois do if).
```

Nesse programa, uma variável chamada `valor` é criada dentro do bloco de comandos do condicional `if` e sua validade é apenas local, apesar de uma declaração mais externa de `valor` existir. Dentro do bloco, apenas a variável `valor` interna pode ser utilizada, apesar de `valor` externo continuar existindo. Declarações mais locais obscurecem a visão de identificadores mais externos com mesmo nome.

Apesar de isso ser possível de ser feito, não é uma boa prática redeclarar identificadores, visto que a clareza fica severamente comprometida.

Dica

O uso não justificado de um mesmo identificador em escopos diferentes, porém próximos, dificulta a compreensão do código e pode levar a erros difíceis de serem localizados e corrigidos.
O uso de identificadores iguais, porém, é válido e será usado em contextos nos quais a multiplicidade de um mesmo nome seja favorável à clareza ao invés de prejudicial.

Como outro exemplo, o programa seguinte replica a necessidade de escrever dois valores em ordem não decrescente, porém havendo dois valores inteiros e dois valores reais.

```c
/*
Apresentação de dois valores em ordem não decrescente
Requer:  dois valores inteiros vi1 e vi2 e dois valores reais vr1 e vr2
Assegura: vi1 <= vi2 e vr1 <= vr2
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    // Obtenção dos valores
    printf("Digite dois valores inteiros: ");
    fgets(entrada, sizeof entrada, stdin);
    int v_int_1, v_int_2;
    sscanf(entrada, "%d%d", &v_int_1, &v_int_2);

    printf("Digite dois valores reais: ");
    fgets(entrada, sizeof entrada, stdin);
    double v_real_1, v_real_2;
    sscanf(entrada, "%lf%lf", &v_real_1, &v_real_2);

    // Verificação da ordem
    if (v_int_2 < v_int_1) {
        int temporario = v_int_1;
        v_int_1 = v_int_2;
        v_int_2 = temporario;
    }

    if (v_real_2 < v_real_1) {
        double temporario = v_real_1;
        v_real_1 = v_real_2;
        v_real_2 = temporario;
    }

    // Apresentação de resultados
    printf("Valores inteiros em ordem não decrescente: %d e %d.\n",
           v_int_1, v_int_2);
    printf("Valores reais em ordem não decrescente: %g e %g.\n",
           v_real_1, v_real_2);

    return 0;
}
```

```
Digite dois valores inteiros: 300 200
Digite dois valores reais: 118.2 302.75
Valores inteiros em ordem não decrescente: 200 e 300.
Valores reais em ordem não decrescente: 118.2 e 302.75.
```

É interessante observar que há duas variáveis chamadas `temporario` existindo em momentos distintos do programa. Não há, porém, problemas com essa duplicidade de uso, uma vez que suas existências são muito localizadas e não se misturam. Naturalmente cabe ao programador optar ou não por usar nomes distintos.

9.2 Outras declarações da linguagem

Um programa não declara apenas variáveis. A própria linha `int main(void)` dos programas é uma declaração da função `main`. Sem essa declaração, o sistema operacional não teria como saber por qual instrução começar a execução. Além disso, dentro de `stdio.h`, por exemplo, estão declarados protótipos das funções `printf`, `sscanf`, `fgets` e etc. Desta forma, depois do `#include` essas funções passam a ser conhecidas e podem ser corretamente usadas.

Uma visão completa de declarações e regras de escopo é tratada no Capítulo 19.

Este programa seguinte ilustra uma das grandes vantagens das declarações locais suplantarem as declarações mais externas existentes.

```c
/*
Exemplo de declaração de uma variável lógica
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool remove = false;
    printf("Vai remover? %s!\n", remove ? "SIM" : "NÃO");

    return 0;
}
```

```
Vai remover? NÃO!
```

A declaração deste código cria uma variável lógica chamada `remove`, a qual é usada sem erros. Porém, em `stdlib.h` é declarada uma função `remove` (em inglês, do verbo *to remove*) que pode ser usada para apagar um arquivo. A sobreposição dos identificadores permite ter um código simples, funcional e claro sem conflito com outras declarações preexistentes.

10 Organização do código fonte

Abelson e Sussman (1996) escreveram uma vez que "programas devem ser escritos para que humanos os leiam e para que, somente como consequência, máquinas os executem"[1]. Em outras palavras, compiladores não ligam para como os programas são escritos, para os nomes das variáveis nem para o bom ou mau gosto do programador. Programas apenas são compilados e executados.

10.1 Documentação

A primeira etapa para se ter um bom código fonte passa longe do código C em si. Ela consiste em ter, no início do código, um comentário com a descrição do conteúdo do arquivo.

Esse comentário estabelece a documentação do código e deve conter uma descrição de seu propósito e de seu contexto, bem como as condições necessárias para sua utilização e execução. Seguem, a título de exemplo, documentações encontradas no Github[2].

Arquivo `sysctl.c`[3]:

```
/*
 * sysctl wrapper for library version of Linux kernel
 * Copyright (c) 2015 INRIA, Hajime Tazaki
 *
 * Author: Mathieu Lacage <mathieu.lacage@gmail.com>
 *         Hajime Tazaki <tazaki@sfc.wide.ad.jp>
 */
```

Arquivo `ubsan.c`[4]

```
// SPDX-License-Identifier: GPL-2.0-only
/*
 * UBSAN error reporting functions
 *
 * Copyright (c) 2014 Samsung Electronics Co., Ltd.
 * Author: Andrey Ryabinin <ryabinin.a.a@gmail.com>
 */
```

Arquivo `seg6.py`[5]:

```
/*
 *  SR-IPv6 implementation
 *
 *  Author:
 *  David Lebrun <david.lebrun@uclouvain.be>
 *
 *
 *  This program is free software; you can redistribute it and/or
```

[1] Tradução livre de "*programs must be written for people to read, and only incidentally for machines to execute*".
[2] Github: https://github.com.
[3] https://github.com/libos-nuse/net-next-nuse/blob/46e2206969943ba3fb87441dee0b433624daf35c/arch/lib/sysctl.c.
[4] https://github.com/KSPP/linux/blob/e0756cfc7d7cd08c98a53b6009c091a3f6a50be6/lib/ubsan.c.
[5] https://github.com/linux-wpan/linux-wpan-next/blob/107bc0aa95ca572df42da43c30a2079266e992e4/net/ipv6/seg6.c.

```
 *    modify it under the terms of the GNU General Public License
 *    as published by the Free Software Foundation; either version
 *    2 of the License, or (at your option) any later version.
 */
```

Arquivo `writing_a_custom_training_loop_in_tensorflow.py`[6] (em Python):

```
"""
Title: Writing a training loop from scratch in TensorFlow
Author: [fchollet](https://twitter.com/fchollet)
Date created: 2019/03/01
Last modified: 2023/06/25
Description: Writing low-level training & evaluation loops in TensorFlow.
Accelerator: None
"""
```

O conteúdo e o nível de detalhe de cada comentário depende do programador, das regras da empresa em que trabalha ou da equipe envolvida, entre outros muitos fatores.

Neste livro, os programas são sempre precedidos por um comentário contendo:

- O propósito do programa;
- As pré e pós-condições a que o código atende.

Segue um exemplo simples desta forma de documentação.

```
/*
Cálculo de juros compostos
Requer: o capital investido, a taxa de juros (a.m.) e o tempo
    de investimento em meses
Assegura: o montante final da transação
*/
```

Com base nessa descrição, que é relativamente simples, não é necessário olhar os comandos do programa para se saber a que ele se propõe, o que ele requer para ser executado e qual o resultado que apresentará ao final.

10.2 Organização visual

O código fonte de um programa é escrito para que outros programadores (ou seja, pessoas) leiam e entendam os comandos e a intenção do programador.

Segue um programa em C válido. Sua qualidade, entretanto, é discutível e é apenas uma nova versão do código que implementa o Algoritmo 1.1.

```
/*
Cálculo e apresentação das raízes reais de uma equação de segundo grau na
    forma ax^2 + bx + c = 0
Requer: Os coeficientes a, b e c da equação
Assegura: as raízes reais da equação; ou mensagem que a equação é
    inválida; ou mensagem que não há raízes reais
*/
#include <stdio.h>
#include <math.h>
int main(void) { char entrada[160];printf(
"Digite os valores de a, b e c da equação: "); fgets(entrada, sizeof
entrada
```

[6] https://github.com/keras-team/keras/blob/419973ee15ecd0e2d085e077399ce3bd5437df15/guides/writing_a_custom_training_loop_in_tensorflow.py.

```
, stdin); double a, b,          c; sscanf(entrada
,    "%lf%lf%lf" , &    a, &b, &
c);
if(a == 0) printf(            "Não é equação do segundo grau (a = 0).\n"    )

; else { double discriminante = pow
(
b
                                                                              ,
2) - 4 * a * c; if(discriminante <
0) printf("A equação não possui raízes reais.\n");
else
if
(discriminante < 1.0e-10) {  double x = -b / (2*    a
            ); printf("Uma raiz: %g.\n", x); } else { double x1 = (-b -
            sqrt(discriminante)) / (2 * a); double x2 = (-b +
sqrt(discriminante))
            / (2 * a); printf("Raízes reais: %g e %g.\n", x1


,

 x2);}}return
 0;}
```

```
Digite os valores de a, b e c da equação: -2 10 13
Raízes reais: 6.07071 e -1.07071.
```

Para ilustrar a dificuldade que o código, mesmo correto na sua lógica e sintaxe, proporciona quando é mal escrito, considere o problema de localizar a vírgula na expressão `pow(b, 2)` que calcula b^2 para o discriminante.

Se a ordem dos comandos estiver correta, o compilador entende o que tem que ser feito e gera o código executável sem problemas ou dificuldades. Porém, caso o compilador aponte um erro como seguinte, qual seria a dificuldade de encontrar o problema?

```
main.c: In function ´main´:
main.c:28:62: warning: unused variable ´x1` [-Wunused-variable]
   28 |             ); printf("Uma raiz: %g.\n", x); } else { double x1 = (-b -
      |                                                              ^~
main.c:31:59: error: ´x1` undeclared (first use in this function);
      |              did you mean ´x2`?
   31 |             / (2 * a); printf("Raízes reais: %g e %g.\n", x1
      |                                                           ^~
      |                                                           x2
main.c:31:59: note: each undeclared identifier is reported only once for each
                    function it appears in
main.c: At top level:
main.c:35:8: error: expected identifier or ´(` before ´return`
   35 |  x2);}}return
      |        ^~~~~~
main.c:36:4: error: expected identifier or ´(` before ´}` token
   36 |  0;}
      |    ^
```

O erro acima foi causado apenas por um `}` inserido no lugar errado. Todas as outras mensagens são decorrentes desta falha na interpretação, a qual impede a sequência da análise e gera uma cascata de erros.

Em suma, os programas são para as pessoas lerem.

No **?@sec-guia-de-estilo** são apresentadas as orientações de escrita seguidas neste livro, mas as seções seguintes introduzem os pontos iniciais, justificando suas importâncias.

A escrita de um bom programa em C recai, necessariamente, na organização visual do código, ou seja, pela disposição espacial dos comandos, instruções e pontuações ao longo do código.

10.3 Indentação

A indentação é o espaço dado da margem esquerda até o comando e ela serve para indicar a hierarquia dos comandos. Esse recurso visual é essencial para um bom programa e tem sido consistentemente usado em todos os exemplos de programas dados.

Segue o exemplo simples de um programa que faz conversões de unidades de distância.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>
int main(void) {
char entrada[160];
printf("Digite uma distância em metros: ");
fgets(entrada, sizeof entrada, stdin);
double em_metros;
sscanf(entrada, "%lf", &em_metros);
double em_centimetros = em_metros * 100;
printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);
double em_pes = em_metros * 3.281;
printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);
double em_jardas = em_metros * 1.094;
printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);
double em_polegadas = em_metros * 39.37;
printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);
return 0;
}
```

O código acima é apresentando sem indentações e isso dificulta enxergar os comandos e até a abrangência do `main`. O acréscimo da indentação indica um nível para os comandos contidos no bloco da função principal e isso leva à versão seguinte do programa.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>
int main(void) {
    char entrada[160];
    printf("Digite uma distância em metros: ");
    fgets(entrada, sizeof entrada, stdin);
    double em_metros;
    sscanf(entrada, "%lf", &em_metros);
    double em_centimetros = em_metros * 100;
    printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);
    double em_pes = em_metros * 3.281;
    printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);
    double em_jardas = em_metros * 1.094;
    printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);
    double em_polegadas = em_metros * 39.37;
    printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);
    return 0;
}
```

A indentação tem que ser constante para um mesmo nível de comandos, o que leva a que comandos de mesmo nível sempre se iniciem na mesma coluna.

No exemplo seguinte há problemas com a declaração da variável `entrada`, que deveria estar indentada, e também com o conjunto de comandos que realizam as conversões. Neste último caso, estes comandos parecem estar de alguma forma "subordinados" aos comandos anteriores, o que não o caso.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>
int main(void) {
char entrada[160];
    printf("Digite uma distância em metros: ");
    fgets(entrada, sizeof entrada, stdin);
    double em_metros;
    sscanf(entrada, "%lf", &em_metros);
        double em_centimetros = em_metros * 100;
        printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);
        double em_pes = em_metros * 3.281;
        printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);
        double em_jardas = em_metros * 1.094;
        printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);
        double em_polegadas = em_metros * 39.37;
        printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);
    return 0;
}
```

Segue, por fim, um último péssimo exemplo de indentação, no qual transparece o desleixo do programador.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>
int main(void) {
 char entrada[160];
    printf("Digite uma distância em metros: ");
   fgets(entrada, sizeof entrada, stdin);
  double em_metros;
    sscanf(entrada, "%lf", &em_metros);
        double em_centimetros = em_metros * 100;
       printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);
       double em_pes = em_metros * 3.281;
       printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);
          double em_jardas = em_metros * 1.094;
        printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);
     double em_polegadas = em_metros * 39.37;
        printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);
  return 0;
}
```

10.4 Linhas de espaçamento

Outro ponto que auxilia uma melhor visualização do código é o uso de linhas em branco, cuja função é separar as diferentes tarefas que o programa tem que cumprir.

É apresentado na sequência o programa de conversão de unidades de distância com o acréscimo de linhas em branco.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
```

```
    printf("Digite uma distância em metros: ");
    fgets(entrada, sizeof entrada, stdin);
    double em_metros;
    sscanf(entrada, "%lf", &em_metros);

    double em_centimetros = em_metros * 100;
    printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);

    double em_pes = em_metros * 3.281;
    printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);

    double em_jardas = em_metros * 1.094;
    printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);

    double em_polegadas = em_metros * 39.37;
    printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);

    return 0;
}
```

Uma linha em branco deve ser incluída para separar as linhas com `#include` do início da função `main`, o que dá destaque a esta função.

Foram introduzidas linhas em branco dentro do `main` para isolar, primeiramente, os comandos que fazem a leitura da distância em metros, incluindo mensagem, leitura, declaração da variável e conversão para `double`. Todos estes comandos são, em essência, "obtenha a distância em metros".

Outros blocos que são agrupados são os de cada uma das conversões, com declaração de variável e a escrita do resultado.

Por último aparece o indicador de sucesso da execução: `return 0`, também destacado dos demais comandos.

Dica

As linhas em branco devem ser, via de regra, apenas uma, devendo ser evitadas duas ou mais linhas separando os grupos de comandos.

O agrupamento de comandos depende do programador, de forma que o mesmo código possa ter sua organização conforme o exemplo seguinte.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite uma distância em metros: ");
    fgets(entrada, sizeof entrada, stdin);
    double em_metros;
    sscanf(entrada, "%lf", &em_metros);

    double em_centimetros = em_metros * 100;
    double em_pes = em_metros * 3.281;
    double em_jardas = em_metros * 1.094;
    double em_polegadas = em_metros * 39.37;

    printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);
    printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);
    printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);
    printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);
```

```
    return 0;
}
```

Neste caso, o agrupamento se refere à leitura, seguida pelos cálculos e finalizada pelas apresentações dos resultados.

10.5 Um comando por linha

Uma regra básica de clareza é separar os comandos de forma que cada um fique em sua própria linha.

Como exemplo, o programa de conversão de distâncias foi reescrito de forma a violar essa regra e é apresentado na sequência.

```c
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite uma distância em metros: "); fgets(entrada,
          sizeof entrada, stdin);
    double em_metros; sscanf(entrada, "%lf", &em_metros);

    double em_centimetros = em_metros * 100; double em_pes = em_metros *
        3.281; double em_jardas = em_metros * 1.094; double em_polegadas
        = em_metros * 39.37;

    printf("> %.1lfm = %.1fcm\n> %.1lfm = %.1f pés\n", em_metros,
           em_centimetros, em_metros, em_pes);
    printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas); printf(
        "> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);

    return 0;
}
```

Essa versão coloca alguns comandos e declarações individuas em uma mesma linha, o que está sintaticamente correto, mas dificulta a visão dos limites entre os diversos elementos.

É importante lembrar que quando os problemas se tornam mais complexos, o código será mais longo e com mais nuances, assim como terá maior número de variáveis. Nesta situação, mais preciosa essa recomendação se torna.

10.6 Padronização de identificadores

Os identificadores usados para as variáveis e demais elementos da linguagem devem ser significativos. Além disso, também devem seguir um mesmo padrão ao longo do código.

O entendimento de um programa pode ser prejudicado se houver uma variável chamada `salario_inicial` e o salário final for indicado por outra com nome `s_final`. O uso de `salario_final` é, sem dúvidas, mais consistente.

Da mesma forma, em um mesmo programa há variáveis como `taxa_juros`, `montante_final` e `saldo_atual` juntamente com outras como `tx_br`, `tx_liq`, `sf` e `p`, por exemplo. As abreviações excessivas não estabelecem um padrão minimamente compatível em contraposição aos nomes mais explícitos.

Nada disso implica em que não possa haver abreviações, desde que atendem à legibilidade. Se em um aplicação há variáveis para diversas taxas diferentes, o uso de `tx_inicial` e `tx_inercial` são razoáveis para o contexto. Porém todas devem prefixadas com `tx_`, sem misturas o prefixo `taxas_`.

10.7 Localização da declaração de variáveis

Embora uma variável possa ser declarada em inúmeras localizações dentro de um código fonte e ainda mantê-lo correto e coerente, é preciso reforçar que o local em que as declarações ocorrem favorecem o entendimento do propósito do programa e de sua lógica. Este tema é abordado de maneira rápida na Seção 9.1.

Por hábitos "ancestrais", de quando a linguagem C não admitia a mistura entre declarações e código, ainda persiste a prática de declarar todas as variáveis juntas no início do bloco. Assim, nas versões iniciais da linguagem os programas deveriam necessariamente declarar todas suas variáveis antes da primeira linha que tivesse um comando ou chamada de função. Segue um exemplo.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
    double em_metros, em_centimetros, em_pes, em_jardas, em_polegadas;

    printf("Digite uma distância em metros: ");
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%lf", &em_metros);

    em_centimetros = em_metros * 100;
    em_pes = em_metros * 3.281;
    em_jardas = em_metros * 1.094;
    em_polegadas = em_metros * 39.37;


    printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);
    printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);
    printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);
    printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);

    return 0;
}
```

```
Digite uma distância em metros: 1.0
> 1.0m = 100.0cm
> 1.0m = 3.3 pés
> 1.0m = 1.1 jardas
> 1.0m = 39.4 polegadas
```

O impacto para programas simples não é, em geral, grande. À medida que o número de linhas cresce, juntamente à complexidade da solução implementada, o "amontoado" de declarações no começo do programa prejudica a legibilidade. Muitas vezes, por exemplo, é preciso verificar se uma variável tem o tipo necessário e fazer as adequações pode trazer impactos não previstos. Por exemplo, ao alterar uma declaração de `int` para `long int` pode afetar não somente a variável de interesse, mas outras declaradas juntas.

10.8 Alinhamento dos finais de bloco

As chaves da função `main` estabelecem o bloco da função principal e inclui as declarações e os comandos que formam o programa. Um bloco de comandos, na forma de um comando composto, é muitas vezes necessário para incluir mais que um comando dentro de um condicional, por exemplo.

Não é incomum que, em programas em C, venha uma sequência de vários fechamentos de bloco, ou seja, vários `}`. Segue um programa que calcula as raízes de uma equação $ax^2 + bx + c = 0$ $(\forall a, b, c)$, mesmo que a equação se reduza a forma linear.

```
/*
Determinação das raízes reais de uma equação ax^2 + bx + c = 0, sem restrições
    aos coeficientes a, b, ou c
Pré-condições: Os valores dos coeficientes a, b e C
Pós-condições: uma ou duas raízes reais ou mensagem indicando que não há
    raízes reais

Quando a é zero, a equação é tratada como linear; caso contrário, é
quadrática. Divisões por zero são tratadas pela divisão de double, ou
seja, podendo resultar em 'not a number' ou +-inf.
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    printf("Considere uma equação ax^2 + bx + c = 0.\n"
           "Digite os valores a, b e c da equação: ");
    fgets(entrada, sizeof entrada, stdin);
    double a, b, c;
    sscanf(entrada, "%lf%lf%lf", &a, &b, &c);

    if (a == 0) {
        // Equação do primeiro grau
        double x = -c / b;
        printf("Raiz: %g.\n", x);
    }
    else {
        // Equação do segundo grau
        double discriminante = pow(b, 2) - 4 * a * c;

        // Determina o número de raízes reais
        if (discriminante < 0)
            printf("Não há raízes reais.\n");
        else if (discriminante < 1e-5) {  // quase zero
            // Raíz única
            double x = -b / (2 * a);
            printf("Raíz: %g.\n", x);
        }
        else {
            // Duas raízes distintas
            double x1 = (-b - sqrt(discriminante)) / (2 * a);
            double x2 = (-b + sqrt(discriminante)) / (2 * a);
            if (x2 < x1) { // ordem não decrescente
                double temporario = x1;
                x1 = x2;
                x2 = temporario;
            }
            printf("Raízes: %g e %g.\n", x1, x2);
        }
    }

    return 0;
}
```

```
Considere uma equação ax^2 + bx + c = 0.
Digite os valores a, b e c da equação: -6 5 28
Raízes: -1.7834 e 2.61673.
```

É necessário que todos os fechamentos de bloco sejam, de alguma forma, alinhados a sua abertura. Esse alinhamento é essencial para localizar, por exemplo, um `}` que foi esquecido.

Nos programas apresentados neste livro, os blocos de comandos compostos são sempre iniciados na linha da estrutura que o utiliza, como um `if`, por exemplo.

```
if(a > 0) {
    // Instruções...
}
else {
    // Outras instruções...
}
```

Desta forma, o fim do bloco fica com a mesma indentação da estrutura que o iniciou.

Por questões de estilo (**?@sec-guia-de-estilo**), há programadores que preferem que as chaves sejam abertas na linha consecutiva. Isso naturalmente não é um problema, desde que o padrão seja mantido para todo o código. Na sequência são apresentadas variações do posicionamento das chaves, porém em todas o alinhamento de indentação é mantido.

```
if(a > 0)
{
    // Instruções...
}
else
{
    // Outras instruções...
}
```

```
if(a > 0) {
    // Instruções...
} else {
    // Outras instruções...
}
```

Como contra exemplo, na sequência é apresentado o programa anterior usando uma "indentação alternativa", que alinha o `}` com seu respectivo `}`, qualquer que seja sua posição. É preciso observer que o resultado quebra a visualização do código e dificulta exatamente saber onde termina cada bloco.

```
/*
Determinação das raízes reais de uma equação ax^2 + bx + c = 0, sem restrições
    aos coeficientes a, b, ou c
Pré-condições: Os valores dos coeficientes a, b e C
Pós-condições: uma ou duas raízes reais ou mensagem indicando que não há
    raízes reais

Quando a é zero, a equação é tratada como linear; caso contrário, é
quadrática. Divisões por zero são tratadas pela divisão de double, ou
seja, podendo resultar em 'not a number' ou +-inf.
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    printf("Considere uma equação ax^2 + bx + c = 0.\n"
           "Digite os valores a, b e c da equação: ");
    fgets(entrada, sizeof entrada, stdin);
    double a, b, c;
    sscanf(entrada, "%lf%lf%lf", &a, &b, &c);

    if (a == 0) {
        // Equação do primeiro grau
        double x = -c / b;
        printf("Raiz: %g.\n", x);
```

```
            }
    else {
        // Equação do segundo grau
        double discriminante = pow(b, 2) - 4 * a * c;

        // Determina o número de raízes reais
        if (discriminante < 0)
            printf("Não há raízes reais.\n");
        else if (discriminante < 1e-5) {  // quase zero
            // Raíz única
            double x = -b / (2 * a);
            printf("Raíz: %g.\n", x);
                                        }
        else {
            // Duas raízes distintas
            double x1 = (-b - sqrt(discriminante)) / (2 * a);
            double x2 = (-b + sqrt(discriminante)) / (2 * a);
            if (x2 < x1) { // ordem não decrescente
                double temporario = x1;
                x1 = x2;
                x2 = temporario;
                         }
            printf("Raízes: %g e %g.\n", x1, x2);
             }
         }

    return 0;
           }
```

```
Considere uma equação ax^2 + bx + c = 0.
Digite os valores a, b e c da equação: -6 5 28
Raízes: -1.7834 e 2.61673.
```

Dica

A maioria dos editores de programas atualmente já provê, ao digitar, uma alinhamento adequado das chaves, o que facilita a digitação. É preciso cuidar para que a própria edição do código não acabe com o alinhamento inadvertidamente.

Dica

O auto-formatador é um recurso muito comum aos IDEs de programação. O uso desse recurso pode ser amplamente empregado!

10.9 Comentários

Comentários devem ser esparsos e minimalistas no programa. Um programa que precisa de muitos comentário é porque o código não está bem escrito. A versão seguinte do programa do peso ideal inclui apenas dois comentários considerados pertinentes. Eles apenas guiam o humano que lê o código para que localize facilmente o significado dos dois cálculos.

```
/*
Cálculo estimado da massa ideal de uma pessoa baseada em seu sexo biológico
e sua altura
Requer: o sexo biológico (masculino ou feminino) e a altura em metros
Assegura: a massa ideal da pessoa apresentada (kg)
*/
#include <stdio.h>

int main(void) {
    char entrada[160];
```

```
    printf("Digite o sexo (M ou F) e a altura em quilogramas: ");
    fgets(entrada, sizeof entrada, stdin);
    char sexo;
    double altura;
    sscanf(entrada, "%c%lf", &sexo, &altura);

    double massa_ideal;
    if(sexo == 'F')
        massa_ideal = 62.1 * altura - 44.7;  // feminino
    else
        massa_ideal = 72.7 * altura - 58;  // masculino

    printf("Massa ideal: %.1fkg.", massa_ideal);

    return 0;
}
```

Embora a comparação `sexo == 'F'` seja suficiente, não custa nem atrapalha indicar a diferença entre as duas fórmulas.

Em nível próximo ao exagero, considerando-se um programa curto como esse, ainda é possível ter comentários indicando a razão dos agrupamentos de comandos.

```
/*
Cálculo estimado da massa ideal de uma pessoa baseada em seu sexo biológico
e sua altura
Requer: o sexo biológico (masculino ou feminino) e a altura em metros
Assegura: a massa ideal da pessoa apresentada (kg)
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    // Obtenção de sexo biológico e altura
    printf("Digite o sexo (M ou F) e a altura em quilogramas: ");
    fgets(entrada, sizeof entrada, stdin);
    char sexo;
    double altura;
    sscanf(entrada, "%c%lf", &sexo, &altura);

    // Cálculo da massa ideal
    double massa_ideal;
    if(sexo == 'F')
        massa_ideal = 62.1 * altura - 44.7;  // feminino
    else
        massa_ideal = 72.7 * altura - 58;  // masculino

    // Apresentação do resultado
    printf("Massa ideal: %.1fkg.", massa_ideal);

    return 0;
}
```

> **Cuidado**
>
> É importante não "supercomentar" um programa. O excesso de comentários tende a tornar o código ilegível.
> O exemplo seguinte pertence à classe "não faça assim".

```
/*
Cálculo estimado da massa ideal de uma pessoa baseada em seu sexo biológico
e sua altura
Requer: o sexo biológico (masculino ou feminino) e a altura em metros
Assegura: a massa ideal da pessoa apresentada (kg)
*/

// Inclusão dos arquivos de cabeçalho
#include <stdio.h>  // funções de entrada e saída

// Programa principal
int main(void) {
    // Variável para a leitura da linha
    char entrada[160];

    // Obtenção de sexo biológico e altura
    printf("Digite o sexo (M ou F) e a altura em quilogramas: ");
    fgets(entrada, sizeof entrada, stdin);  // leitura da linha digitada
    // Declaração das variáveis de entrada
    char sexo;  // para o sexo biológico
    double altura;  // para a altura da pessoa
    // Conversão da linha para obtenção dos valores
    sscanf(entrada, "%c%lf", &sexo, &altura);

    // Declaração da massa ideal
    double massa_ideal;

    // Verificação do cálculo conforme feminino ou masculino
    // (se o sexo for diferente de 'F', assume que seja 'M')
    if(sexo == 'F')
        // Realização do cálculo para o sexo feminino
        massa_ideal = 62.1 * altura - 44.7;
    else
        // Realização do cálculo para o sexto masculino
        massa_ideal = 72.7 * altura - 58;

    // Apresentação do resultado
    printf("Massa ideal: %.1fkg.", massa_ideal);  // massa ideal em kg

    // Indicação de término do programa com sucesso
    return 0;
}
```

Nesta versão, há comentários óbvios (como em `double altura;  // para a altura da pessoa`) e até a visualização da estrutura do `if`, que é simples, fica um tanto comprometida com o excesso de informações.

10.10 Organização consistente do código

A organização do código deve ser consistente. As conversões de distância podem usar um programa correto, porém inconsistente quanto a sua organização. Essa é uma situação que deve ser firmemente evitada.

```
/*
Conversões de distância de metros para centímetros, pés, jardas e polegadas
Requer: uma distância em metros
Assegura: O valor equivalente em centímetros, pés, jardas e polegadas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    double em_centimetros;
    printf("Digite uma distância em metros: ");
    fgets(entrada, sizeof entrada, stdin);
    double em_metros;
```

```
    sscanf(entrada, "%lf", &em_metros);

    em_centimetros = em_metros * 100;
    double em_pes = em_metros * 3.281;

    printf("> %.1lfm = %.1fcm\n", em_metros, em_centimetros);
    printf("> %.1lfm = %.1f pés\n", em_metros, em_pes);

    double em_jardas = em_metros * 1.094, em_polegadas;
    printf("> %.1lfm = %.1f jardas\n", em_metros, em_jardas);

    em_polegadas = em_metros * 39.37;
    printf("> %.1lfm = %.1f polegadas\n", em_metros, em_polegadas);

    return 0;
}
```

```
Digite uma distância em metros: 10
> 10.0m = 1000.0cm
> 10.0m = 32.8 pés
> 10.0m = 10.9 jardas
> 10.0m = 393.7 polegadas
```

Neste programa pode ser até complicado localizar em que momento houve a declaração da variável `em_polegadas`, por exemplo.

Parte IV

Controle de repetição do fluxo

11 Repetições com `while`

A linguagem C permite laços de repetição usando `while`, `for` e `do while`. Neste capítulo a estrutura e aspectos lógicos do `while` são apresentados. O `for` é abordado no Capítulo 12 e o `do while`, no Capítulo 13.

11.1 A estrutura de repetição `while`

A estrutura de repetição que o `while` implementa pode ser chamada de indefinida, pois o número de repetições depende de uma condição que se altera ao longo do tempo. Sua estrutura é apresentada na sequência.

> **Estrutura do `while`**
>
> `while (` *expressão_lógica* `)` *comando*

O uso dessa estrutura é intuitivo: enquanto a *expressão_lógica* for avaliada como verdadeira, o *comando* é executado. Este último pode ser um comando simples terminado em ponto e vírgula ou um comando composto delimitado por chaves.

O `while` sempre testa a condição antes de iniciar cada execução, o que leva à possibilidade que nada seja repetido se a condição já for falsa previamente. Caso a condição seja verdadeira, todo os comandos condicionados são executados antes de ser feita nova verificação.

Para que a estrutura de repetição em si faça sentido, em *comando* deve haver alguma alteração que leve a condição a se tornar falsa em algum momento.

Segue um exemplo simples de uso do `while` na forma de um programa que implementa o Algoritmo 11.1.

Algoritmo 11.1: Conversão de unidades de pressão para uma sequência de valores.

Descrição: Conversão de uma sequência de valores de pressão dadas em mmHg para atm, usando um valor nulo como sentinela
Requer: Uma sequência potencialmente vazia de valores de pressão (mmHg) não nulos seguidos por um valor sentinela nulo
Assegura: O valor de cada medida de pressão convertido para atm

Obtenha $pressão_{mmHg}$ ▷ *primeiro valor da entrada*
enquanto $pressão_{mmHg}$ não for zero **faça** ▷ *verificação se é sentinela*
  Calcule $pressão_{atm}$ como $\frac{1}{760} \times pressão_{mmHg}$
  Apresente $pressão_{atm}$

  Obtenha $pressão_{mmHg}$ ▷ *próximo valor da sequência*
fim enquanto

A implementação desse algoritmo em C pode ser dada como o código seguinte.

```c
/*
Conversão de uma sequência de valores de pressão dadas em mmHg para atm,
    usando um valor nulo como sentinela
Requer: Uma sequência potencialmente vazia de valores de pressão (mmHg)
    não nulos seguidos por um valor sentinela nulo
Assegura: O valor de cada medida de pressão convertido para atm
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite o valor da pressão (mmHg) ou 0 para terminar: ");
    fgets(entrada, sizeof entrada, stdin);
    double pressao_mmhg;
    sscanf(entrada, "%lf", &pressao_mmhg);

    while (pressao_mmhg != 0) {
        double pressao_atm = pressao_mmhg / 760;
        printf("Pressão em ATM: %.2f.\n", pressao_atm);

        printf("Digite o valor da pressão (mmHg) ou 0 para terminar: ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%lf", &pressao_mmhg);
    }

    return 0;
}
```

```
Digite o valor da pressão (mmHg) ou 0 para terminar: 500
Pressão em ATM: 0.66.
Digite o valor da pressão (mmHg) ou 0 para terminar: 1000
Pressão em ATM: 1.32.
Digite o valor da pressão (mmHg) ou 0 para terminar: 2565.26
Pressão em ATM: 3.38.
Digite o valor da pressão (mmHg) ou 0 para terminar: 760.0
Pressão em ATM: 1.00.
Digite o valor da pressão (mmHg) ou 0 para terminar: 0
```

O mapeamento do Algoritmo 11.1 para C é bastante direto. Inicialmente é feita a primeira leitura, que pode ser tanto um valor válido quanto o valor sentinela e, em seguida, é feito o teste. Sendo bem sucedida a verificação (i.e, não é o sentinela), é feita apresentado o resultado da conversão de unidade e solicitado o próximo valor da sequência. Este ciclo de verificação e execução é repetido, sendo encerrado quando a verificação se torna falsa e `return 0` é finalmente executado.

Se a sequência for vazia e na entrada de dados constar apenas o valor sentinela, a condição do `while` falha já na primeira tentativa e o programa se encerra sem que nenhuma conversão seja feita.

Outro ponto curioso sobre a implementação é a verificação de igualdade feita com um valor `double`. Neste caso, a tolerância não é necessária, pois `pressao_mmhg` não é o resultado calculado e, portanto, quando for digitado o valor nulo, a conversão será exata.

Para outro exemplo, segue um algoritmo para contabilização do resultado de uma pesquisa, que considera

11.2 O `double` nas verificações do `while`

Um cuidado essencial ao programador são as verificações que, embora façam sentido no código, falham por conta da precisão dos tipos numéricos reais, como `double` ou `float`.

Para exemplificar essa questão, o trecho de código seguinte pode ser considerado.

```
double a = 0;
while (a != 1) {
    printf("a = %g.\n", a);
    a += 0.1;
}
```

Essa repetição nunca termina, pois `a` nunca será igual a 1. Na precisão dupla que o `double` proporciona, o valor mais próximo de 0,1 é, na realidade, 0.1000000000000000055511151231 26. Assim, quando se espera que a variável tenha valor 1, ha uma diferença de 1,11022 $\times$ 10^{-6} pelos erros acumulados.

Assim, salvaguardas devem ser adotadas para evitar essas ocorrências.

No exemplo dado, uma solução seria interromper a repetição caso o valor de `a` ultrapassasse 1.

```
double a = 0;
while (a <= 1) {  // garantia de não ultrapassar o limite
    printf("a = %g.\n", a);
    a += 0.1;
}
```

Outra alternativa é o estabelecimento de uma tolerância.

```
double a = 0;
while (fabs(a - 1) < 0.00001) {  // considera próximo o suficiente
    printf("a = %g.\n", a);
    a += 0.1;
}
```

Curiosidade

No exemplo de incremento desta seção foi destacado que 0,1 não tem representação exata em um `double`. O valor 0,25, porém, é exato. Nesse caso, o código seguinte funcionaria conforme o esperado.

```
double a = 0;
while (a != 1) {
    printf("a = %g.\n", a);
    a += 0.25;
}
```

Mesmo assim, recomenda-se fortemente que a igualdade ou desigualdade não seja empregada. O uso de `while (a <= 1)` deve ser usado mesmo neste caso.

11.3 Repetições baseadas na entrada

Muitas das repetições nos programas são necessárias para processar uma sequência de dados de entrada. De forma geral, o `while` é usado quando não se conhece a quantidade de repetições que serão executadas, relacionadas neste caso ao número de dados que servirão como entrada para o programa.

Nesse contexto, há duas formas importantes para a entrada, uma usando um valor sentinela (que indica o fim dos dados) e outra que reconhece que os dados acabaram por si, em um indicador explícito.

11.3.1 Entrada com sentinela

Uma forma comum de entrada de dados é o uso de um valor sentinela que indica o fim dos dados. O programa seguinte apresenta uma sequência de leituras para a soma de valores inteiros maiores que zero, usando o valor nulo como sentinela para indicar o fim dos dados.

```
/*
Exemplo de leitura de valores inteiros positivos, usando o zero como sentinela
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite um valor inteiro positivo ou zero para terminar: ");
    fgets(entrada, sizeof entrada, stdin);
    int valor;
    sscanf(entrada, "%d", &valor);

    while (valor != 0) {
        printf("Valor digitado: %d.\n", valor);

        printf("Digite um valor inteiro positivo ou zero para terminar: ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%d", &valor);
    }
    printf("Encerrado!\n");

    return 0;
}
```

```
Digite um valor inteiro positivo ou zero para terminar: 12
Valor digitado: 12.
Digite um valor inteiro positivo ou zero para terminar: 26
Valor digitado: 26.
Digite um valor inteiro positivo ou zero para terminar: 13
Valor digitado: 13.
Digite um valor inteiro positivo ou zero para terminar: 3
Valor digitado: 3.
Digite um valor inteiro positivo ou zero para terminar: 20
Valor digitado: 20.
Digite um valor inteiro positivo ou zero para terminar: 0
Encerrado!
```

Não é incomum o programador optar por expandir o conceito do valor sentinela. Sem prejuízo aos propósitos do programa original, o programador poderia usar na condição do `while` a a expressão `valor > 0`, o que incluiria qualquer valor negativo como sentinela.

11.3.2 Detecção do encerramento do fluxo de entrada

Outra alternativa para indicar o fim da sequência de dados é o encerramento da entrada em si. Em sistemas Linux, ao se digitar `Ctrl-D` em uma linha vazia é indicado ao programa que a entrada de dados foi encerrada. No Windows, o equivalente é digitar `Ctrl-Z` seguido de `ENTER`.

Uma leitura com função `fgets` é capaz de entender o encerramento dos dados. Essa função, como outras funções, retorna um valor resultante, o qual tem sido sistematicamente ignorado. Como exemplo, os comandos seguintes fazem uma leitura e apresentam o resultado lido. Neste caso, o valor de retorno do `fgets` é ignorado.

```
char entrada[160];
fgets(entrada, sizeof entrada, stdin);
printf("%s", entrada);
```

Entretanto, essa função retorna um valor que pode ser verificado. O valor `NULL` é retornado em caso de erro. Assim, com uma verificação extra, a leitura da linha poderia ser escrita conforme apresentado na sequência.

```
char entrada[160];
if (fgets(entrada, sizeof entrada, stdin) != NULL)
    printf("%s", entrada);
else
    printf("Houve algum problema com a leitura.\n");
```

O "problema" mais comum da falha na leitura é o fim do fluxo de entrada. Assim, um programa pode usar essa verificação para repetir a leitura condicionalmente. O programa seguinte exemplifica leituras de linha de texto e contagem do número de linhas[1].

```
/*
Contagem do número de linhas no fluxo de entrada
Requer: uma sequência de linhas de texto
Assegura: a apresentação do número de linhas desse texto
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    printf("Digite seu texto linha a linha (Ctrl-D) para terminar.\n");
    int contador_linhas = 0;
    while (fgets(entrada, sizeof entrada, stdin) != NULL)
        contador_linhas++;

    printf("Número de linhas: %d.\n", contador_linhas);

    return 0;
}
```

```
Digite seu texto linha a linha (Ctrl-D) para terminar.
CAPITULO V
O AGREGADO

Nem sempre ia naquele passo vagaroso e rígido. Também se descompunha em
acionados, era muita vez rápido e lépido nos movimentos, tao natural nesta como
naquela maneira. Outrossim, ria largo, se era preciso, de um grande riso sem
vontade, mas comunicativo, a tal ponto as bochechas, os dentes, os olhos, toda a
cara, toda a pessoa, todo o mundo pareciam rir nele. Nos lances graves,
gravíssimo.
Número de linhas: 9.
```

Dica

As funções de leitura, quando detectam o fim do fluxo de entrada associado a `stdin`, ignoram entradas subsequentes. Assim, depois de terminada uma sequência com `Ctrl-D`, outras chamadas a `fgets` são ignoradas. Isso acontece devido a um controle do fluxo feito pelo sistema operacional. Entretanto, é possível retomar a leitura com `clearerr`, que limpa o indicador de fim de fluxo.

[1] O texto digitado como exemplo foi extraído de Dom Casmurro, de Machado de Assis, obtido em http://www.educadores.diaadia.pr.gov.br/arquivos/File/2010/literatura/obras_completas_literatura_brasileira_e_portuguesa/MACHADO_ASSIS/DCASMURRO/CASMURRO5.HTM.

```c
/*
Contagem do número de linhas no fluxo de entrada
Requer: uma sequência de linhas de texto
Assegura: a apresentação do número de linhas desse texto
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    char entrada[160];

    printf("Digite seu texto linha a linha (Ctrl-D) para terminar.\n");
    int contador_linhas = 0;
    while (fgets(entrada, sizeof entrada, stdin) != NULL)
        contador_linhas++;

    printf("Número de linhas: %d.\n", contador_linhas);

    // Reassumindo a leitura de stdin
    clearerr(stdin);
    printf("Digite seu nome: ");
    char nome[80];
    fgets(nome, sizeof nome, stdin);
    nome[strlen(nome) - 1] = '\0';  // eliminação do \n
    printf("Obrigado, %s, por sua digitação!\n", nome);

    return 0;
}
```

Sem a chamada a `clearerr`, a leitura do nome é ignorada.

11.4 Exemplos

Esta seção apresenta alguns exemplos mais realistas do uso do `while` na codificação de alguns algoritmos.

11.4.1 Contagem regressiva

Um exemplo simples de repetição com `while` é apresentado na sequência. O programa implementa um cronômetro regressivo simples e segue o Algoritmo 11.2.

Algoritmo 11.2: Contagem regressiva segundo a segundo

Descrição: Realiza a contagem regressiva para um tempo determinado
Requer: o tempo total para contagem
Assegura: a apresentação de uma contagem regressiva e mensagem de finalização

Obtenha o tempo de contagem
enquanto o tempo for maior ou igual a zero **faça**
  Apresente o tempo restante
  Aguarde 1s
  Reduza tempo de contagem em 1s
fim enquanto
Apresente uma mensagem de término da contagem

```c
/*
Realiza a contagem regressiva para um tempo determinado
Requer: o tempo para contagem (inteiro)
```

```
Assegura: a apresentação de uma contagem regressiva e mensagem de finalização
*/
#include <stdio.h>
#include <unistd.h>

int main(void) {
    char entrada[160];

    printf("Digite o tempo para contagem.\n"
           "Exemplos: '10' ou '10s' para 10s, '3m' para 3 minutos "
           "ou '5h' para 5 horas.\n"
           "Tempo: ");
    fgets(entrada, sizeof entrada, stdin);
    int tempo_restante;
    char unidade_tempo = 's';  // padrão: segundos
    sscanf(entrada, "%d%c", &tempo_restante, &unidade_tempo);

    // Conversão de unidades se necessário; senão considera segundos
    switch (unidade_tempo) {
        case 'h':  // horas
            tempo_restante = tempo_restante * 3600;
            break;
        case 'm':  // minutos
            tempo_restante = tempo_restante * 60;
            break;
    }

    // Contagem
    printf("Contagem para %d segundo(s):\n", tempo_restante);
    while (tempo_restante >= 0) {
        int tempo_horas = tempo_restante / 3600;
        int tempo_minutos = (tempo_restante - tempo_horas * 3600) / 60;
        int tempo_segundos = tempo_restante % 60;
        printf("> Tempo restante: %02dh %02dmin %02ds\r", tempo_horas,
               tempo_minutos, tempo_segundos);
        fflush(stdout);

        sleep(1);  // aguarda 1 segundo
        tempo_restante--;
    }
    printf("\nObrigado pela paciência!\n");

    return 0;
}
```

Este código usa o `while` para a contagem regressiva. A cada execução, é apresentando o valor do cronômetro (dado por `tempo_restante`) e esse tempo é decrescido de um segundo. A chamada `sleep(1)` (declarada em `unistd.h`) suspende a execução do processo por um segundo antes de retomar a contagem.

Valem ainda dois comentários sobre esse programa. O primeiro é sobre o uso de `\r` ao apresentar o cronômetro. Esse caractere faz com que o cursor do terminal volte para o início da linha sem passar para a linha de baixo e, assim, o próximo `printf` sobrescreve o horário anterior, atualizando-o. O outro ponto é o uso da função `fflush(stdout)` (incluído via `stdio.h`), a qual faz com que o texto escrito pelo programa seja imediatamente colocado no terminal. Sem ele, há atrasos na escrita e o contador parece irregular.

O programa de contagem regressiva não é preciso. Ele se baseia em intervalos de um segundo, mas a cada repetição outras operações são feitas. A longo prazo, o cronômetro ficará atrasado em relação ao tempo real. Este programa é para ser um mero exemplo da repetição e não tem maiores pretensões.

11.4.2 Atualização de saldo

Um saldo bancário tem um valor inicial e, dependendo de uma série de transações de crédito ou débito, o montante é modificado. Esta é a proposta do Algoritmo 11.3.

Algoritmo 11.3: Atualização de saldo bancário a partir de créditos e débitos.

Descrição: Atualização de um saldo bancário a partir de uma sequência de movimentações de crédito ou débito ocorridas em um período
Requer: o saldo inicial e uma sequência de transações compostas pelo valor da transação e indicador se é crédito ou débito
Assegura: a apresentação do saldo atualizado

Obtenha o saldo inicial
enquanto existem transações para serem processadas **faça**
  Obtenha o tipo da transação ▷ *tipo = crédito ou débito*
  Obtenha o valor da transação
  Atualize o valor do saldo de acordo com a transação
fim enquanto
Apreente o saldo atualizado

A implementação em C apresentada na sequência considera que cada transação é entrada expressando o tipo da transação com `C` para crédito ou `D` para débito em uma linha e o valor monetário em outra. Não há valor sentinela para finalizar a entrada e o programa assume que o encerramento do fluxo de entrada implica no fim dos dados.

```
/*
Atualização de um saldo bancário a partir de uma sequência de movimentações de crédito ou débito ocorridas em
Requer: o saldo inicial e uma sequência de transações, cada um fazendo indicando separadamente o tipo 'C' ou '
Assegura: a apresentação do saldo atualizado

Transações não reconhecidas são rejeitadas
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    // Obtenção do saldo inicial
    printf("Digite o saldo inicial: ");
    fgets(entrada, sizeof entrada, stdin);
    double saldo;
    sscanf(entrada, "%lf", &saldo);

    // Atualização do saldo de acordo com as transações
    printf("Digite Ctrl-D (Linux) ou Ctrl-Z (Windows) para encerrar.\n\n"
           "Tipo (C ou D): ");
    while (fgets(entrada, sizeof entrada, stdin) != NULL) {  // leitura do tipo
        char tipo_transacao = '*';  // '*' = inválido
        sscanf(entrada, "%c", &tipo_transacao);

        printf("Valor: R$ ");
        double valor_transacao;
        fgets(entrada, sizeof entrada, stdin);  // leitura do valor
        sscanf(entrada, "%lf", &valor_transacao);

        switch (tipo_transacao) {
            case 'C':
                saldo += valor_transacao;
                break;
            case 'D':
                saldo -= valor_transacao;
                break;
            default:
                printf("Transação não reconhecida (%c).\n", tipo_transacao);
        }

        printf("Tipo (C ou D): ");  // próxima transação
    }

    // Apresentação do saldo atualizado
```

```
    printf("\nSaldo final: R$ %.2f.\n", saldo);

    return 0;
}
```

```
Digite o saldo inicial: 5800.00
Digite Ctrl-D (Linux) ou Ctrl-Z (Windows) para encerrar.

Tipo (C ou D): D
Valor: R$ 300.00
Tipo (C ou D): C
Valor: R$ 82.50
Tipo (C ou D): C
Valor: R$ 181.00
Tipo (C ou D): D
Valor: R$ 1.25
Tipo (C ou D): D
Valor: R$ 789.42
Tipo (C ou D):
Saldo final: R$ 4972.83.
```

12 Repetições com `for`

A linguagem C permite laços de repetição, sendo um deles dado pelo `for`. Neste capítulo essa estrutura é abordada.

12.1 A estrutura de repetição `for`

O `for` é um comando usado para repetições definidas, ou seja, para as quais o número de vezes que os comandos serão executados já é conhecida. Por exemplo, se houver um dado disponível para cada hora do dia, sabe-se de antemão que há 24 dados e uma repetição com esse número de vezes pode ser empregado.

Estrutura do `for`

`for (` *iniciação* `;` *condição_de_continuidade* `;` *incremento* `)` *comando*

Na linha das demais estruturas de fluxo, também o `for` admite a repetição de um único *comando*, que pode ser simples ou composto. Suas demais partes são:

- A *iniciação*, a qual é composta pela atribuição inicial e é executada uma única vez antes das repetições;
- A *condição_de_continuidade*, cujo valor verdadeiro é condição para iniciar uma execução de *comando*;
- O *incremento*, o qual é usado para atualizar o valor da variável usada no laço.

O seguinte trecho de código pode ser usado como exemplo.

```
for (int i = 0; i < 10; i++)
    printf("%d ", i);
```

Ele pode ser considerado como "para `i` variando de 0 até 9" em um total de 10 repetições. A variável `i` é declarada dentro do `for` e, como tal, possui validade apenas nesse escopo. Seu valor inicial é zero e essa atribuição é feita antes das repetições. Antes de qualquer repetição, a condição `i < 10` é avaliada e o `printf` somente é executado se a condição for verdadeira. Terminada a execução do comando do `for`, o incremento `i++` é executado, alterando o valor de `i` antes da próxima verificação de continuidade.

Em termos gerais, o `for` exemplificado é absolutamente equivalente trecho seguinte.

```
{
    int i = 0;  // iniciação
    while (i < 10) {  // condição de continuidade
        printf("%d ", i);  // comando
        i++;  // incremento
    }
}
```

Observar que todo o código está dentro de um comando composto (entre chaves) é importante, visto que `i` tem validade exclusiva dentro do bloco de comandos.

Outros exemplos de repetição com `for` são apresentados na sequência.

```
// contagem de 10 até 1
for (int i = 10; i > 0; i--)
    printf("%d ", i);
```

```
// contagem de 1 até 100
for (int i = 1; i <= 100; i++)
    printf("%d ", i);
```

```
// contagem de 0 a 1 de 0,1 em 0,1
for (double i = 0; i <= 1; i += 0.1)
    printf("%d ", i);
```

12.2 `for` *versus* `while`

Conforme apresentado na seção anterior, a estrutura do `for` equivale à de um `while`. A questão, então, é porque usar o `for`?

A estrutura do `for` é um recurso vastamente utilizada nas linguagens de programação e comum também no pseudocódigo. Como ela é indicada para repetições já previsíveis, ter todas as informações necessárias logo no início do comando ajuda a compreensão do propósito do trecho de código.

Em princípio, ao se olhar a linha em que está o `for` já se tem ideia de como começa e como termina. O uso do `while`, por sua vez, indica uma imprevisibilidade do término, o que leva um eventual leitor do programa a procurar como os dados são modificados e como isso afeta a condição da repetição.

Dica

Não se deve usar um `for` para substituir um `while` nem um `while`para substituir um `for`. Quando o número de repetições é conhecido, o `for` deve ser empregado; se esse número estiver em aberto, a melhor opção é o `while`.
Porém, não é incomum o uso do `while` em repetições definidas.
Um discussão mais ampla sobre o uso do `for`, complementando esta dica, está na seção Capítulo 15.

12.3 Repetições baseadas na entrada

Uma forma de se processar uma sequência de valores é prefixá-la com a quantidade de valores de entrada. Assim, se existirem 25 dados para serem processados, o valor 25 é introduzido antes dos dados, o que viabiliza ter, antes da repetição, o número de vezes de execução e, assim, usar o `for` adequadamente e com clareza.

Segue um exemplo simples de um programa que apresenta os primeiros n números ímpares naturais.

```
/*
Apresentação dos n primeiros números naturais ímpares
Requer: a quantidade n
Assegura: a apresentação dos n primeiros números ímpares
*/
#include <stdio.h>
```

```
int main(void) {
    char entrada[160];

    printf("Digite a quantidade desejada de ímpares: ");
    fgets(entrada, sizeof entrada, stdin);
    int quantidade;
    sscanf(entrada, "%d", &quantidade);

    for(int i = 0; i < quantidade; i++)
        printf("%d ", 2 * i + 1);
    printf("\n");

    return 0;
}
```

```
Digite a quantidade desejada de ímpares: 12
1 3 5 7 9 11 13 15 17 19 21 23
```

12.4 Exemplos

Nesta seção são apresentados alguns exemplos de programas simples com repetições para as quais o `for` é a escolha mais direta.

12.4.1 Média de 24 temperaturas

Na linha do exemplo das temperaturas hora a hora é apresentado o Algoritmo 12.1, o qual determina a média das temperaturas colhidas a cada hora de um dia.

Algoritmo 12.1: Cálculo da média das temperaturas horárias de um dia

Descrição: Cálculo da média de 24 medidas de temperatura colhidas hora a hora durante um dia
Requer: uma sequência de 24 temperaturas em Celsius
Assegura: a média dessas temperaturas

> **para** *hora* $\leftarrow$ 0 **até** 23 **faça**
>   Obtenha uma medida de temperatura
>   Acumule essa medida em *soma*
> **fim para**
> Calcule *média* como $\frac{1}{24}soma$
> Apresente *média*

Segue, agora, a implementação desse algoritmo.

```
/*
Cálculo da média de 24 medidas de temperatura colhidas hora a hora durante um dia
Requer: uma sequência de 24 temperaturas em Celsius
Assegura: a média dessas temperaturas
*/
#include <stdio.h>

int main(void) {
    printf("Digite as temperaturas.\n");

    double soma_temperaturas = 0;
    for (int hora = 0; hora < 24; hora++) {
        char entrada[160];
        printf("Temperatura de %2dh: ", hora);
        fgets(entrada, sizeof entrada, stdin);
```

```
        double temperatura;
        sscanf(entrada, "%lf", &temperatura);

        soma_temperaturas += temperatura;
    }

    printf("Média diária: %.1f°C.\n", soma_temperaturas / 24);

    return 0;
}
```

```
Digite as temperaturas.
Temperatura de  0h: 9.7
Temperatura de  1h: 9.6
Temperatura de  2h: 10.6
Temperatura de  3h: 8.1
Temperatura de  4h: 14.1
Temperatura de  5h: 14.5
Temperatura de  6h: 12.1
Temperatura de  7h: 18.2
Temperatura de  8h: 19.2
Temperatura de  9h: 20.7
Temperatura de 10h: 24.2
Temperatura de 11h: 24.5
Temperatura de 12h: 18.4
Temperatura de 13h: 23.1
Temperatura de 14h: 21.5
Temperatura de 15h: 21.9
Temperatura de 16h: 23.9
Temperatura de 17h: 23.7
Temperatura de 18h: 18.7
Temperatura de 19h: 20.4
Temperatura de 20h: 16.6
Temperatura de 21h: 14.2
Temperatura de 22h: 8.0
Temperatura de 23h: 9.5
Média diária: 16.9°C.
```

12.4.2 Áreas de triângulos

O Algoritmo 12.2 é uma proposta de nível alto para a determinação da área de uma dada quantidade de triângulos. A quantidade de áreas a serem calculadas é a primeira entrada esperada pela solução.

Algoritmo 12.2: Cálculo da área de uma sequência de triângulos.

Descrição: Cálculo da áea de diversos triângulos definidos pelas coordenadas em $\mathbb{R}^2$ de seus vértices
Requer: a quantidade de triângulos seguida pelas coordenadas (x,y) de cada vértice do triângulo
Assegura: a apresentação de cada área calculada

Obtenha a quantidade de triângulos
para cada $i \leftarrow 1$ **até** quantidade de triângulos **faça**
  Obtenha as coordenadas dos vértices
  Calcule a área do triângulo
  Apresente a área calculada
fim para

Para mais detalhes, segue a versão do Algoritmo 12.2 com menor grau de abstração. Nesta versão optou-se por fazer o cálculo da área por semiperímetro.

Algoritmo 12.3: Cálculo da área de uma sequência de triângulos usando o semiperímetro.

Descrição: Cálculo da área de diversos triângulos definidos pelas coordenadas em $\mathbb{R}^2$ de seus vértices
Requer: a quantidade de triângulos seguida pelas coordenadas (x,y) de cada vértice do triângulo
Assegura: a apresentação de cada área calculada

Obtenha a quantidade de triângulos
para cada $i \leftarrow 1$ **até** quantidade de triângulos **faça**
▷ *Obtenção das coordenadas dos vértices*
Obtenha as coordenadas (x_1, y_1), (x_2, y_2) e (x_3, y_3)

▷ *Cálculo da área do triângulo por semiperímetro*
Calcule o lado l_1 como $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ ▷ *distância* $P_1 \rightarrow P_2$
Calcule o lado l_2 como $\sqrt{(x_1 - x_3)^2 + (y_1 - y_3)^2}$ ▷ *distância* $P_1 \rightarrow P_3$
Calcule o lado l_3 como $\sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2}$ ▷ *distância* $P_2 \rightarrow P_3$
Calcule o semiperímetro p como $\frac{l_1+l_2+l_3}{2}$
Calule *área* como $\sqrt{p(p - l_1)(p - l_2)(p - l_3)}$

Apresente *área*
fim para

Uma implementação do Algoritmo 12.3 em C é apresentada na sequência.

```
/*
Cálculo da área de diversos triângulos definidos pelas coordenadas em R^2
    de seus vértices
Requer: a quantidade de triângulos seguida pelas coordenadas $(x,y)$ de
    cada vértice do triângulo
Assegura: a apresentação de cada área calculada
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    printf("Digite a quantidade de triângulos: ");
    fgets(entrada, sizeof entrada, stdin);
    int quantidade;
    sscanf(entrada, "%d", &quantidade);

    for (int i = 1; i <= quantidade; i++) {
        printf("Triângulo %d:\n"
               "Digite x e y do vértice 1: ", i);
        fgets(entrada, sizeof entrada, stdin);
        double x1, y1;
        sscanf(entrada, "%lf%lf", &x1, &y1);  // (x1, y1)
        printf("Digite x e y do vértice 2: ");
        fgets(entrada, sizeof entrada, stdin);
        double x2, y2;
        sscanf(entrada, "%lf%lf", &x2, &y2);  // (x2, y2)
        printf("Digite x e y do vértice 3: ");
        fgets(entrada, sizeof entrada, stdin);
        double x3, y3;
        sscanf(entrada, "%lf%lf", &x3, &y3);  // (x3, y3)

        double lado1 = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
        double lado2 = sqrt(pow(x1 - x3, 2) + pow(y1 - y3, 2));
        double lado3 = sqrt(pow(x2 - x3, 2) + pow(y2 - y3, 2));
        double semiperimetro = (lado1 + lado2 + lado3) / 2;

        double area = sqrt(semiperimetro * (semiperimetro - lado1) *
                           (semiperimetro - lado2) * (semiperimetro - lado3));
```

```
        printf("Área do triângulo %d: %.2f.\n\n", i, area);
    }

    return 0;
}
```

```
Digite a quantidade de triângulos: 2
Triângulo 1:
Digite x e y do vértice 1: 0 0
Digite x e y do vértice 2: 0 1
Digite x e y do vértice 3: 1 0
Área do triângulo 1: 0.50.

Triângulo 2:
Digite x e y do vértice 1: -1.5 -1
Digite x e y do vértice 2: 1.5 -1
Digite x e y do vértice 3: 0 3.8
Área do triângulo 2: 7.20.
```

12.4.3 Desenho de um "círculo"

O programa seguinte mostra o aninhamento de dois laços `for`. Cada um deles vai de -10 a 10 e a saída é um `*` caso a distância ao "centro" seja menor que 10. No programa, ao invés de calcular $\sqrt{x^2 + y^2} \leq 10$, optou-se por usar a expressão equivalente $x^2 + y^2 \leq 10^2$, que não requer calcular a raiz quadrada.

```
/*
Desenho de um círculo usando caracteres
Assegura: apresentação de um círculo tão razoável quanto possível
*/
#include <stdio.h>

int main(void) {
    for (int x = -10; x <= 10; x++) {
        for (int y = -10; y <= 10; y++)
            if (x * x + y * y <= 100)  // distância < 10
                printf("XXX");  // no círculo
            else
                printf("   ");  // fora do círculo
        printf("\n");
    }

    return 0;
}
```

```
                              XXX
                  XXXXXXXXXXXXXXXXXXXXXXXXXXX
            XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
            XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                  XXXXXXXXXXXXXXXXXXXXXXXXXXX
                              XXX
```

Como cada caractere escrito sempre tem altura maior que a largura, optou-se por usar três caracteres para cada posição para tentar compensar essa diferença e obter o "círculo" com menor distorção. A escolha por três caracteres foi feita por tentativa e erro.

13 Repetições com `do while`

Este capítulo contempla a estrutura e aspectos lógicos do `do while`, cuja função é executar comandos repetitivamente.

13.1 A estrutura de repetição `do while`

O `do while` é uma estrutura de repetição condicional e, assim, depende de uma dada condição para determinar se haverá ou não outra repetição. Em oposição ao `while` (Capítulo 11) e ao `for` (Capítulo 12), o teste de continuidade é feito depois da execução do comando.

> `do` *comando* `while (` *condição_de_continuidade* `) ;`

Nessa estrutura de repetição, o primeiro passo é a execução de *comando* que, como nas demais estruturas, pode ser um comando simples ou um bloco de comandos. Terminada a execução, a *condição_de_continuidade* é avaliada, seguindo para nova repetição se for verdadeira ou encerrando `do while` se não for.

Assim, esta estrutura executa *comando* pelo menos uma vez.

Para exemplificar, o trecho de código seguinte ilustra uma leitura de um valor real garantindo que esteja no intervalo de 0 a 10 (inclusive).

```c
/*
Leitura de valor com garantia de estar no intervalo [0, 10], com releitura
    se necessário
Requer: uma sequência de valores terminada por um no intervalo [0, 10]
Assegura: apresentação do último valores
*/
#include <stdio.h>

int main(void) {
    double valor;
    do {
        char entrada[160];
        printf("Digite um valor: ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%lf", &valor);
    } while (valor < 0 || valor > 10);

    printf("\nValor aceito: %g.\n", valor);

    return 0;
}
```

```
Digite um valor: 12.2
Digite um valor: -3.05
Digite um valor: 7.4

Valor aceito: 7.4.
```

A lógica da leitura é direta: faz a leitura e, caso não esteja no intervalo especificado, faz a leitura novamente. Neste caso, a primeira leitura é necessária e seu valor sempre é relevante.

Um detalhe relevante é o escopo das declaraçoes de variáveis: como `entrada` é usada apenas localmente, é declarada dentro do bloco de comandos do `do while`; `valor`, por sua vez, tem que ser declarada fora (antes) do bloco, caso contrário não existiria para ser escrita depois da repetição.

> ⚠ Curiosidade
>
> Em um laço `do while`, as variáveis declaradas dentro do bloco de comandos ficam tão restritas ao próprio bloco que não podem nem ser acessadas na condição da repetição.
>
> ```
> do {
>     bool eh_valido;  // declaração interna
>
>     ...
> } while (eh_valido);  // `eh_valido` não existe aqui...
> ```

13.2 Exemplos

Nesta seção há alguns exemplos de programas que utilizam a estrutura de repetição `do while`.

13.2.1 Leitura de senha

Este exemplo é de um programa que solicita uma senha para autorizar ou não a execução. A senha é *hardcoded* (i.e., fixa no código e nada versátil) e há um limite de três tentativas.

```
/*
Processamento somente mediante senha
Requer: uma sequência de tentativas de senha terminada com a senha correta
    ou com comprimento máximo de três tentativas
Assegura: apresentação de mensagem autorizando ou negando acesso conforme a
    a senha esteja ou não correta
*/
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

int main(void) {
    bool validou_senha = false;
    int numero_tentativas = 0;
    do {
        numero_tentativas++;
        printf("Digite a senha: ");
        char senha[160];
        fgets(senha, sizeof senha, stdin);

        validou_senha = strcmp(senha, "12345678\n") == 0; // é igual?
    } while (!validou_senha && numero_tentativas < 3);

    if (validou_senha)
        printf("\nAcesso autorizado.\n");
    else
        printf("\nAcesso negado.\n*** Este problema será reportado.\n");

    return 0;
}
```

```
Digite a senha: senha
Digite a senha: senha123
Digite a senha: 123456

Acesso negado.
*** Este problema será reportado.
```

Neste código há o uso da função `strcmp`, declarada no arquivo de cabeçalho `string.h`. Ela retorna zero se as duas cadeias de caracteres passadas como parâmetro forem iguais[1]. Uma observação interessante é que `senha` é declarada dentro do bloco do `do while`, uma vez que ela não é necessária fora dele; o mesmo comentário não é válido para `validou_senha`.

13.2.2 Pense em um número de 1 a 15.000!

O programa deste exemplo se propõe a adivinhar um número pensado pelo usuário em no máximo 13 tentativas.

```c
/*
Pense em um número de 1 até 15.000 para eu adivinhar qual é!
Requer: respostas do usuário dizendo se o valor é maior (+) ou menor (-)
    que o número pensado
Assegura: a apresentação do número pensado pelo usuário em até 14 tentativas
*/
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>

int main(void) {
    printf("Pense em um número de 1 a 15000 e responda...\n");
    sleep(8);  // 8s para pensar!

    int inicio_intervalo = 1;
    int fim_intervalo = 15000;
    int numero_tentativas = 0;
    int tentativa;
    bool acertei = false;
    do {
        numero_tentativas++;
        tentativa = (inicio_intervalo + fim_intervalo) / 2;

        printf("É o %d?\n", tentativa);
        printf("Responda + se o número for maior, - se for menor "
               "ou = se eu acertei: ");
        char resposta[160];
        fgets(resposta, sizeof resposta, stdin);
        switch (resposta[0]) {
            case '+':
                inicio_intervalo = tentativa + 1;  // tentarei número maior
                break;
            case '-':
                fim_intervalo = tentativa - 1;  // tentarei número menor
                break;
            case '=':
                acertei = true;  // UFA!
                break;
            default:
                printf("Não entendi a resposta... tente de novo.\n");
                numero_tentativas--;  // essa tentativa não conta!
        }

        if (!acertei) {
            if (numero_tentativas == 13)
                printf("Minha ÚLTIMA CHANCE!!!\n");
            else if (numero_tentativas > 10)
                printf("Só tenho mais %d chances.. :-(\n",
                       14 - numero_tentativas);
            sleep(0.7);  // pausa leve de dramaticidade
        }
    } while (!acertei && fim_intervalo >= inicio_intervalo);

    if (acertei)
        printf("\n\nBeleza! Acertei em %d tentativas!\n", numero_tentativas);
    else
        printf("\n\nHummmm! Algo de errado não está certo aqui...\n"
               "Suas respostas foram corretas?\n");
```

[1] Manipulações de cadeias de caracteres são tratadas em detalhes no Capítulo 16.

```
    return 0;
}
```

```
Pense em um número de 1 a 15000 e responda...
É o 7500?
Responda + se o número for maior, - se for menor ou = se eu acertei: +
É o 11250?
Responda + se o número for maior, - se for menor ou = se eu acertei: +
É o 13125?
Responda + se o número for maior, - se for menor ou = se eu acertei: +
É o 14063?
Responda + se o número for maior, - se for menor ou = se eu acertei: -
É o 13594?
Responda + se o número for maior, - se for menor ou = se eu acertei: -
É o 13359?
Responda + se o número for maior, - se for menor ou = se eu acertei: +
É o 13476?
Responda + se o número for maior, - se for menor ou = se eu acertei: =


Beleza! Acertei em 7 tentativas!
```

14 Arquivos texto

Todos os programas apresentados até o momento escrevem e leem texto, usando o terminal como interface: tela para a apresentação de informações e teclado para receber caracteres digitados.

Na escrita de um programa em C usando a função `printf`, a saída é sempre um texto. Este texto pode ser explícito e direto, como o do comando seguinte.

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

Por outro lado, o texto escrito pode ser o resultante de uma conversão do valor de uma variável, exemplificado na sequência.

```
double d = 1.32143211;
printf("v = %.2f\n", d);  // 1.32
```

Neste último `printf`, a variável `d` contém um dado valor representado internamente com o tipo `double`, mas a escrita é uma representação textual desse valor. De acordo com o formato `%.2f`, a transformação do valor para texto gera a sequência de caracteres `1`, `.`, `3` e `2`. A saída, portanto, é texto.

A questão uso de texto também ocorre nas leituras. Os programas recebem uma sequência de caracteres (texto) obtidas por `fgets`, mas um eventual valor numérico digitado deve ser convertido para um tipo `int` ou `double`, por exemplo. Assim, toda entrada para os programas também tem sido exclusivamente no formato textual. Segue um exemplo destacando a conversão.

```
char entrada[160];  // texto
fgets(entrada, sizeof entrada, stdin);  // obtém texto
double d;
sscanf(entrada, "%lf", &d);  // converte o texto digitado para double
```

14.1 Fluxos de entrada e saída

Um programa em C recebe os caracteres digitados no terminal e produz suas saídas também na tela do terminal. A sequencia de caracteres digitados é denominada fluxo de entrada e o texto gerado é o fluxo de saída.

Esta seção abordada os fluxos usuais do programa e introduz novos fluxos, estes associados a arquivos.

14.1.1 Entrada e saída padrão

Durante a execução de um programa, ele é capaz de escrever e ler de um terminal. O sistema operacional automaticamente associa as leituras a um fluxo chamado de entrada padrão, conhecido por *stdin* (de *standard input*). A saída, por seu turno, é associada à chamada saída padrão, que é o fluxo *stdout* (*standard output*).

Ao executar um programa, *stdout* é associada ao terminal e, assim, todos as chamadas a `printf` produzem textos nesse terminal. De forma similar, *stdin* é também associada a esse terminal, de forma que as digitações realizadas são transferidas para o programa (daí o último argumento de `fgets` ser `stdin`).

Curiosidade

Existe, ainda, um segundo fluxo de saída chamado *stderr* (*standard error*), que também é associado ao terminal e funciona praticamente da mesma forma que *stdout*.

Para concluir, o programa em execução pega dados de um fluxo de entrada e gera dados um fluxo de saída.

14.1.2 Texto plano e outros fluxos de entrada e saída

Nos fluxos de entrada e saída padrão, via de regra, são produzidos o que se convencionou chamar de texto plano (ou texto simples). Esse texto plano se refere apenas ao fluxo de caracteres convencionais, sem atributos associados. Em outras palavras, o nem `fgets` nem `printf` lidam com itálicos, negritos, espaçamento de linhas entre parágrafos ou linhas centralizadas, para citar alguns exemplos de formatação.

Quando um programa em C é escrito em um editor, ele é um texto plano. A maioria dos IDEs coloca cores e negritos para destacar palavras chaves e comentários, mas esse recurso é apenas visual e automático; essas características não são salvas junto como o código fonte. Textos planos são os produzidos em IDEs ou editores simples, como o GEdit ou o Notepad.

Programas em C são capazes de gerenciar outros fluxos além de *stdin* e *stdout*, os quais são manipulados como textos planos. Esses novos fluxos são usualmente associados a arquivos, o que permite que as leituras sejam feitas usando-se os caracteres armazenados em um arquivo e, adicionalmente, que as saídas produzam textos direcionados para outro arquivo.

O termo geral arquivo é usado para qualquer fluxo de dados de entrada ou saída de um programa. Estranhamente, *stdin* e *stdout* são tratados internamente também como arquivos.

A função de escrita `printf` escreve na saída padrão. Há uma outra função similar, a `fprintf`, esta última permitindo especificar o fluxo para o qual texto será enviado.

```c
/*
*/
#include <stdio.h>

int main(void) {
    // Duas saídas equivalente
    printf("Hello, world!\n");  // implicitamente envia para stdout
    fprintf(stdout, "Hello, world!\n"); // explícito para stdout

    return 0;
}
```

```
Hello, world!
Hello, world!
```

A função `fprintf` possui um novo parâmetro obrigatório, que é o fluxo que será usado. Neste programa em particular, as duas instruções possuem o mesmo efeito, pois ambas as funções estão associadas a *stdout*.

14.2 Fluxo de escrita para arquivo

Para a criação de um novo fluxo existe uma função em `stdio.h` chamada `fopen`. Esta é uma função de chamada ao sistema operacional, solicitando o acesso a um determinado arquivo.

Para entender mais facilmente a criação de um novo fluxo dados, o programa seguinte parte do uso do fluxo convencional, ou seja, saída na tela. O programa lê uma sequência de coordenadas em $\mathbb{R}^2$ e apresenta as coordenadas e sua distância à origem. Segue o programa com entradas e saídas convencionais.

```
/*
Criação de um novo arquivo contendo dados digitados pelo usuário
Requer: uma sequência de pares de pontos (x, y), usando o ponto (0, 0) como
    valor sentinela
Ensure: a apresentação de cada ponto e de sua distância à origem
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    printf("Digite x e y do ponto (ou 0 0  para terminar): ");
    fgets(entrada, sizeof entrada, stdin);
    double x, y;
    sscanf(entrada, "%lf%lf", &x, &y);
    while (x != 0 || y != 0) {
        // Cálculo da distância à origem
        double distancia_origem = sqrt(x * x + y * y);

        // Escrita das informações no arquivo
        fprintf(stdout, "(%.1f, %.1f) --> %.1f\n", x, y, distancia_origem);

        // Próximo (x, y)
        printf("Digite x e y do ponto (ou 0 0  para terminar): ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%lf%lf", &x, &y);
    }

    return 0;
}
```

```
Digite x e y do ponto (ou 0 0  para terminar): 3 -2
(3.0, -2.0) --> 3.6
Digite x e y do ponto (ou 0 0  para terminar): 1 -2
(1.0, -2.0) --> 2.2
Digite x e y do ponto (ou 0 0  para terminar): 10 5
(10.0, 5.0) --> 11.2
Digite x e y do ponto (ou 0 0  para terminar): 14.2 -14.2
(14.2, -14.2) --> 20.1
Digite x e y do ponto (ou 0 0  para terminar): 0 0
```

Um ponto a se notar nesse programa é o uso da função `fprintf` para realizar a escrita, mesmo quando o `printf` poderia ser usado. Esse uso é proposital para identificar as alterações para mudar o fluxo para um arquivo.

A nova versão do programa é apresentada na sequência, sendo que a saída produzida pelo programa é gravada em um arquivo chamado `distancias.txt`.

```c
/*
Criação de um novo arquivo contendo dados digitados pelo usuário
Requer: Uma sequência de pares de pontos (x, y), usando o ponto (0, 0) como
    valor sentinela
Ensure: um arquivo texto contendo, linha a linha, as coordenadas do ponto e
    suas distâncias à origem
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    char entrada[160];

    // Criação do arquivo de saída
    FILE *arquivo_destino = fopen("distancias.txt", "w");

    if (arquivo_destino == NULL)
        printf("Erro ao criar novo arquivo.\n");
    else {
        printf("Digite x e y do ponto (ou 0 0  para terminar): ");
        fgets(entrada, sizeof entrada, stdin);
        double x, y;
        sscanf(entrada, "%lf%lf", &x, &y);
        while (x != 0 || y != 0) {
            // Cálculo da distância à origem
            double distancia_origem = sqrt(x * x + y * y);

            // Escrita das informações no arquivo
            fprintf(arquivo_destino, "(%.1f, %.1f) --> %.1f\n", x, y, distancia_origem);

            // Próximo (x, y)
            printf("Digite x e y do ponto (ou 0 0  para terminar): ");
            fgets(entrada, sizeof entrada, stdin);
            sscanf(entrada, "%lf%lf", &x, &y);
        }

        // Encerramento do acesso ao arquivo
        fclose(arquivo_destino);
    }

    return 0;
}
```

```
Digite x e y do ponto (ou 0 0  para terminar): 3 -2
Digite x e y do ponto (ou 0 0  para terminar): 1 -2
Digite x e y do ponto (ou 0 0  para terminar): 10 5
Digite x e y do ponto (ou 0 0  para terminar): 14.2 -14.2
Digite x e y do ponto (ou 0 0  para terminar): 0 0
```

A primeira diferença entre essa versão e a original é especificação de um arquivo, ou seja, um novo fluxo de dados. Essa especificação é feita usando-se uma variável `arquivo_destino` cujo tipo é `FILE *`. A essa variável é atribuído o valor retornado pela função `fopen`.

```c
FILE *arquivo_destino = fopen("distancias.txt", "w");
```

Conforme apresentada no programa, essa função solicita ao sistema operacional para criar um novo arquivo, o qual usará o nome `distancias.txt`. Essa função não é infalível e o sistema operacional pode, assim, negar a criação. Entre as razões para a criação falhar podem ser citadas, a título de exemplo, que o programa tenha autorização de criar um arquivo no diretório corrente ou que já exista um arquivo com esse nome e ele seja protegido para não ser reescrito. No caso de falha, a função retorno o valor `NULL`. O segundo parâmetro, `"w"`, indica que o arquivo deve ser criado.

Assim, essa possibilidade de falha é tratada no programa, que verifica se o valor retornado foi ou não `NULL`.

```
if (arquivo_destino == NULL)
    printf("Erro ao criar novo arquivo.\n");  // aqui houve erro
else {
    // Aqui o arquivo foi criado com sucesso
    ...
}
```

A relação entre o arquivo real (armazenado em um disco rígido, por exemplo) e o programa em C se dá por meio do valor retornado pela função `fopen` e armazenado na variável `arquivo_destino`. O programa usa essa variável para indicar o que é feito e o sistema operacional se encarrega de refletir os efeitos dos comandos no arquivo real.

O arquivo real é chamado usualmente de arquivo físico, enquanto a variável associada a esse arquivo é conhecida como arquivo lógico.

Se o arquivo de saída foi criado com sucesso, o programa segue para o `else`, cujos comandos reproduzem as mesmas leituras do programa original, incluindo as mesmas verificações e conversões. Nessa parte, a primeira diferença está no comando `fprintf`, cujo primeiro argumento é `arquivo_destino`.

```
fprintf(arquivo_destino, "(%.1f, %.1f) --> %.1f\n", x, y, distancia_origem);
```

O texto que seria escrito no terminal é, agora, gravado no arquivo `distancias.txt`, incluindo as formatações (`%.f`) e mudanças de linha (`\n`).

Dessa forma, o arquivo de saída fica com o conteúdo apresentado na sequência.

```
(3.0, -2.0) --> 3.6
(1.0, -2.0) --> 2.2
(10.0, 5.0) --> 11.2
(14.2, -14.2) --> 20.1
```

Há, finalmente, ainda uma diferença importante: depois de encerrada a sequência de entrada, o arquivo é fechado com `fclose`.

```
fclose(arquivo_destino);
```

Fechar um arquivo faz com que o sistema operacional grave todos os bytes enviados para o arquivo, atualize suas informações de armazenamento (tamanho, data de gravação etc.) e libere os recursos que o sistema estava usando para controlar o acesso ao arquivo.

É importante notar que o arquivo somente é fechado caso tenha sido aberto com sucesso. É por essa razão que o `fclose` está no bloco de comandos condicionados no `else`.

Dica

Em geral, quando um programa tem sua execução terminada, os arquivos abertos por ele são automaticamente fechados.
Entretanto, é uma péssima prática não deixar o fechamento dos arquivos explicitamente indicados no código. O uso do `fclose`, portanto, é mandatório para um código de boa qualidade.

Figura 14.1: Três formas armazenadas em arquivo segundo seus vértices.

14.3 Fluxo de leitura

Da mesma forma que a saída textual de um programa pode ser direcionada a um arquivo, criando um fluxo de escrita, também a entrada pode ser feita de um arquivo, criando-se um fluxo de entrada.

Para exemplificar um fluxo de entrada, será considerado um arquivo textual contendo dados sobre triângulos e retângulos. O arquivo conterá, em sua primeira linha, um valor com a quantidade de figuras contidas no arquivo. Para os triângulos há uma linha contendo o caractere T e seis valores reais, cada par deles correspondendo às coordenadas (x, y) dos vértices do triângulo; para os retângulos, o caractere será R e há apenas dois pontos (quatro valores) para referenciá-lo, correspondendo a dois vértices opostos (os lados do retângulo são sempre considerados paralelos aos eixos).

Para um exemplo inicial, pode ser considerado o arquivo seguinte, cujo nome é `figuras.txt`. Esse arquivo pode ser criado com qualquer editor de texto plano.

```
3
R 0.0 0.0 5.5 2.0
T -1.0 1.0 3.0 4.5 4 -0.5
R -1.2 5.2 2.2 3.1
```

Neste exemplo, o `3` indica que há três formas. A primeira é o retângulo R_1 da Figura 14.1; segue-se o triângulo T e a última é o retângulo R_2.

Uma solução para processar os dados desse arquivo e apresentar as áreas de cada figura é o Algoritmo 14.1, apresentado com nível alto de abstração.

Algoritmo 14.1: Processamento de formas em arquivo com apresentação das suas áreas, com prefixação do número de formas.

Descrição: Processamento de um arquivo de formas (triângulos e retângulos) para cálculo das áreas
Requer: arquivo texto com número de formas seguido de uma forma por linha: triângulo (letra T mais coordenadas de seus três vértices) ou retângulo (letra R e as coordenadas de dois vértices opostos)
Assegura: a apresentação da área de cada forma

Obtenha *número_formas* do arquivo
para $i \leftarrow 1$ **até** *número_formas* **faça** ▷ *primeira linha*
    Obtenha uma forma ▷ *uma linha*
    se a forma é um triângulo **então**
        Calcule a área por semiperímetro ▷ *triângulo*
    senão
        Calcule a área como base multiplicada por altura ▷ *retângulo*
    fim se
    apresente a área calculada
fim para

A implementação desse algoritmo em C pode ser feita conforme o programa seguinte. Nessa codificação, o arquivo lógico é dado pela variável `arquivo_formas`, associado ao arquivo físico `figuras.txt`.

```
/*
Processamento de um arquivo de formas (triângulos e retângulos) para cálculo
    das áreas
Requer: arquivo texto com nome 'figuras.txt':
    na linha 1: número de formas
    nas linhas restantes, conforme a quantidade da linha 1:
        - OU a letra T mais coordenadas dos vértices do triângulo
        - OU a letra R e as coordenadas de dois vértices opostos do
          retângulo
Assegura: a apresentação da área de cada forma
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    FILE *arquivo_formas = fopen("figuras.txt", "r");

    if (arquivo_formas == NULL) {
        printf("Falha ao abrir arquivo de formas.\n");
    }
    else {
        char entrada[160];
        fgets(entrada, sizeof entrada, arquivo_formas);
        int numero_formas;
        sscanf(entrada, "%d", &numero_formas);

        // Processamento das formas
        for (int i = 0; i < numero_formas; i++) {
            // Obtenção da linha com a forma
            fgets(entrada, sizeof entrada, arquivo_formas);
            double area;
            if (entrada[0] == 'T') {
                // Triângulo: área por semiperímetro
                double x1, y1, x2, y2, x3, y3;
                sscanf(entrada, "%*c%lf%lf%lf%lf%lf%lf",
                       &x1, &y1, &x2, &y2, &x3, &y3);
                double lado1 = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
                double lado2 = sqrt(pow(x1 - x3, 2) + pow(y1 - y3, 2));
                double lado3 = sqrt(pow(x2 - x3, 2) + pow(y2 - y3, 2));
                double semiperimetro = (lado1 + lado2 + lado3) / 2;

                area = sqrt(semiperimetro * (semiperimetro - lado1) *
```

```
                                (semiperimetro - lado2) * (semiperimetro - lado3));
            }
            else {
                // Retângulo: ára por base x altura
                double x1, y1, x2, y2;
                sscanf(entrada, "%*c%lf%lf%lf%lf", &x1, &y1, &x2, &y2);

                area = fabs(x2 - x1) * fabs(y2 - y1);
            }

            printf("%.2f\n", area);
        }

        fclose(arquivo_formas);
    }

    return 0;
}
```

```
11.00
11.75
7.14
```

A função `fopen` usa como parâmetros o nome do arquivo de formas e o modo `"r"`, que indica acesso para leitura. Todas as leituras são feitas linha a linha, usando a variável genérica `entrada` para armazenamento temporário. Para o caso da primeira linha, a qual contém apenas a quantidade de formas do arquivo, esse valor é obtido pelo `sscanf` usando-se o formato `%d`.

No caso das linhas de cada forma, cada uma delas é armazenada em `entrada`, porém analisadas de forma diferente. Como o primeiro caractere da linha indica sua forma, a decisão entre triângulo e retângulo é feita com base em `entrada[0]`, que será `T` ou `R`.

Com base no tipo da forma, a linha é analisada de forma separada. No caso do triângulo, são esperados seis valores que formam as três coordenadas dos vértices e, desse modo, o `sscanf` usa o padrão de formato `%*c%lf%lf%lf%lf`, que ignora o primeiro caractere e faz a varredura procurando seis valores reais. Para o caso do retângulo, no qual há apenas dois pontos, o padrão usado é `%*c%lf%lf%lf%lf`, o qual ignora o `R` e obtém os quatro valores que formam as duas coordenadas dos vértices opostos.

Naturalmente, terminado o processamento do arquivo, `fclose` é usado para liberar o sistema operacional de gerenciar o arquivo de dados.

14.4 Mais sobre a abertura e o fechamento de arquivos texto

Nesta seção são reapresentadas as funções de manipulação de arquivos com mais detalhamento sobre seu funcionamento e uso dos parâmetros.

14.4.1 A função `fopen`

A função de associação entre o programa e o arquivo real é a `fopen`. Ela sempre possui dois parâmetros, sendo o primeiro a indicação do nome do arquivo e o segundo, o modo de abertura.

O nome do arquivo é uma cadeia de caracteres que pode ser somente o nome do arquivo, como `"figuras.txt"`, por exemplo. Nesse caso, o arquivo físico é considerado em relação ao diretório corrente. O nome pode indicar também o caminho: `"../figuras.txt"` indica o arquivo no diretório pai (isto

é, um acima do atual) e `"/home/user/dados/figuras.txt"` dá o caminho absoluto e independe do diretório corrente[1].

Em relação aos modos, como `"w"` ou `"r"` usados nos exemplos, a Tabela 14.1 sumariza como funcionam.

Tabela 14.1: Modos do `fopen` para manipulação de arquivos texto.

Modo	Significado
`"w"`	Prepara um arquivo somente para escrita, criando um arquivo novo ou truncando o existente para tamanho zero. A posição corrente é ajustada para o início do arquivo.
`"r"`	Prepara um arquivo somente para leitura. A posição corrente é ajustada para o início do arquivo.
`"a"`	Prepara um arquivo somente para escrita. Se o arquivo já existir, a posição corrente é ajustada para o final do arquivo; senão, um novo arquivo é criado.

Arquivos texto são, via de regra, processados sequencialmente, do início ao fim ou do início até um ponto de interesse qualquer. Raramente são abertos para leitura e escrita, de forma que essa opção foi omitida da lista apresentada.

A Tabela 14.1 também introduz um elemento importante, embora intuitivamente fácil de compreender: a posição corrente. Esse termo é aplicado ao local no arquivo onde as operações de leitura e escrita serão realizadas. Por exemplo, quando um arquivo é aberto em modo `"r"`, a posição corrente está no início do arquivo, que é o byte zero. A cada leitura, a posição corrente é ajustada para o próximo byte que será lido.

Embora a posição corrente não seja diretamente usada pelo programador no processamento sequencial do arquivo, um exemplo simples de programa ilustra esse conceito.

```c
/*
Escrita de algumas linhas de texto em um arquivo com acompanhamento da
    posição corrente de Escrita
Assegura: criação de arquivo 'texto.txt' com algum texto e apresentação
    dos bytes usados em cada linha durante a escrita
*/
#include <stdio.h>

int main(void) {
    FILE *arquivo_numeros = fopen("texto.txt", "w");

    if (arquivo_numeros == NULL)
        printf("Falha ao criar arquivo de saída.\n");
    else {
        printf("Primeira linha: bytes de %ld", ftell(arquivo_numeros));
        fprintf(arquivo_numeros, "Primeira linha deste arquivo.\n");
        printf(" a %ld.\n", ftell(arquivo_numeros) - 1);

        printf("Segunda linha: bytes de %ld", ftell(arquivo_numeros));
        fprintf(arquivo_numeros, "Linha 2");  // escrita em duas etapas
        fprintf(arquivo_numeros, " segue aqui...\n");
        printf(" a %ld.\n", ftell(arquivo_numeros) - 1);

        printf("Terceira linha: bytes de %ld", ftell(arquivo_numeros));
        fprintf(arquivo_numeros, "Finalmente o fim do texto! (linha 3)");
        printf(" a %ld.\n", ftell(arquivo_numeros) - 1);

        fclose(arquivo_numeros);
    }
```

[1] No Linux, usado como plataforma para este livro, o separador de diretórios é a barra `/`. O mesmo ocorre para o MacOS. Em sistemas Windows, os caminhos usam a barra reversa `\` e, como essa barra tem significado especial em C, é preciso indicar o caminho com o escape `\\`: `"..\\figuras.txt"` ou `C:\\User\\Documentos\\Dados\\figuras.txt`.

```
    return 0;
}
```

```
Primeira linha: bytes de 0 a 29.
Segunda linha: bytes de 30 a 51.
Terceira linha: bytes de 52 a 87.
```

O conteúdo de `texto.txt` ficou como se segue.

```
Primeira linha deste arquivo.
Linha 2 segue aqui...
Finalmente o fim do texto! (linha 3)
```

Para visualização, a seguir é apresentado o conteúdo do arquivo com 16 bytes por linha. A primeira coluna é a contagem de caracteres (primeira linha do `od` são os bytes de zero a 15), o que ajuda a identificar a posição de cada caractere relativo a sua posição do arquivo. Os valores `\n` são as mudanças de linha usuais[2]. Vale a pena confrontar os valores escritos pelo programa anterior com as posições reais do arquivo.

```
$ od -Ad -tc texto.txt
0000000   P   r   i   m   e   i   r   a       l   i   n   h   a       d
0000016   e   s   t   e       a   r   q   u   i   v   o   .  \n   L   i
0000032   n   h   a       2       s   e   g   u   e       a   q   u   i
0000048   .   .   .  \n   F   i   n   a   l   m   e   n   t   e       o
0000064       f   i   m       d   o       t   e   x   t   o   !       (
0000080   l   i   n   h   a       3   )
0000088
```

O conceito da posição corrente também é válido para as leituras, que se iniciam na posição zero (início do arquivo) e, após cada leitura, é atualizada para o próximo byte que será lido em um próximo `fgets`.

A ideia da posição em que ocorre a próxima operação de entrada ou saída é a usada quando o modo `"a"` é usado no `fopen`. Nesse caso, o arquivo é aberto para escrita, seu conteúdo (se existir) é preservado e a posição corrente é colocada depois do último byte do arquivo. Com isso, uma nova escrita no arquivo acrescentará texto ao seu final.

Um *log* é um arquivo usado para registrar eventos. Por exemplo, sempre que um usuário entra no sistema ou quando um trabalho de impressão é iniciado, cada evento é registrado no *log*. O programa seguinte, cada vez que executado, solicita uma linha de informação e a grava em um arquivo de *log*.

```
/*
Inserção de uma linha no arquivo de log 'registro.log'
Requer: a digitação da linha de informação
Assegura: o acréscimo dessa linha no final do arquivo
*/
#include <stdio.h>

int main(void) {
    FILE *arquivo_log = fopen("registro.log", "a");

    if (arquivo_log == NULL)
        printf("Falha ao criar ou abrir arquivo de log.\n");
    else {
        printf("Digite a informação a ser registrada:\n> ");
        char linha_de_log[160];
        fgets(linha_de_log, sizeof linha_de_log, stdin);

        fprintf(arquivo_log, "%s", linha_de_log);
```

[2] No Windows, a mudança de linha costuma ser indicada pela sequência `\r\n`, de modo que o conteúdo do arquivo pode diferir um pouco caso seja essa a plataforma usada para execução do programa.

```
        fclose(arquivo_log);
    }

    return 0;
}
```

```
Digite a informação a ser registrada:
> Primeiro registro do log.
```

Na primeira execução, o arquivo `registro.log` ainda não existe, o que faz com que ele seja criado vazio. Então o programa solicita o primeiro registro e o grava no arquivo.

A cada nova execução, uma linha é solicitada e adicionada ao final do arquivo.

Curiosidade

Quando um arquivo é aberto apenas para escrita (`"w"` e `"a"`), tentativas de leitura desse arquivo são apenas ignoradas sem gerar qualquer erro. De forma similar, ao se tentar escrever em um arquivo aberto com `"r"`, a escrita não é feita e nenhum erro é indicado.

14.4.2 A função `fclose`

O encerramento do acesso ao arquivo é feito pela função `fclose`, para a qual é preciso passar como parâmetro a variável correspondente a um arquivo aberto com sucesso. Uma vez fechado, nenhuma operação de entrada ou saída para o arquivo é possível.

14.5 A função `fprintf`

A escrita em um arquivo texto com o `fprint` funciona de forma similar ao `print` na tela. A função, inicialmente, processa a cadeia de caracteres de formato e faz as conversões de formato, gerando o texto final. Então, esse texto é gravado no arquivo. Enquanto `print` é sempre direcionado para *stdout*, `fprintf` permite escolher a qual fluxo de saída o texto será enviado.

Como exemplo, pode ser considerado o programa seguinte, que cria um arquivo com algumas operações com raiz quadrada.

```
/*
Criação de arquivo com valores de raízes quadradas
Assegura: criação de arquivo texto contendo valores de raízes
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    FILE *arquivo_raizes = fopen("raizes.txt", "w");

    if (arquivo_raizes == NULL)
        printf("Falha ao criar arquivo.\n");
    else {
        for (double x = 0.0; x <= 100; x += 0.5)
            fprintf(arquivo_raizes, "raiz(%.1f) = %.3f\n", x, sqrt(x));

        fclose(arquivo_raizes);
    }

    return 0;
}
```

O arquivo resultante da execução, `raizes.txt`, segue.

```
raiz(0.0) = 0.000
raiz(0.5) = 0.707
raiz(1.0) = 1.000
raiz(1.5) = 1.225
raiz(2.0) = 1.414

... [191 linhas omitidas]

raiz(98.0) = 9.899
raiz(98.5) = 9.925
raiz(99.0) = 9.950
raiz(99.5) = 9.975
raiz(100.0) = 10.000
```

Executada uma chamada de `fprint`, a escrita é feita a partir da posição corrente, a qual é atualizada para o próximo byte depois do último byte escrito.

14.6 Leitura até o fim do arquivo

Quando um arquivo contém dados, além da estratégia de prefixar os dados com a quantidade (estratégia usada no Algoritmo 14.1), é também possível varrer o arquivo até encontrar seu final. O Algoritmo 14.2 aborda esse novo problema, ainda no contexto de um arquivo com formas triangulares e retangulares.

Algoritmo 14.2: Processamento de formas em arquivo com apresentação das suas áreas, com uma forma por linha.

Descrição: Processamento de um arquivo de formas (triângulos e retângulos) para cálculo das áreas
Requer: arquivo texto contendo uma forma por linha: triângulo (letra T mais coordenadas de seus três vértices) ou retângulo (letra R e as coordenadas de dois vértices opostos)
Assegura: a apresentação da área de cada forma

enquanto existem formas no arquivo **faça**
  Obtenha uma forma ▷ *uma linha*
  se a forma é um triângulo **então**
    Calcule a área por semiperímetro ▷ *triângulo*
  senão
    Calcule a área como base multiplicada por altura ▷ *retângulo*
  fim se
  apresente a área calculada
fim enquanto

A função `fgets` opera da mesma forma para um arquivo aberto quanto para `stdin`, ou seja, os dados do arquivo são lidos para uma variável do programa até que seja encontrado um final de linha (`\n`). O efeito prático é a leitura de uma linha por vez.

Quando não há mais dados no arquivo para serem lidos, situação em que o fim do arquivo foi atingido, o valor de retorno de `fgets` é `NULL`. Assim, pela verificação do valor de retorno, é possível verificar se se chegou ao fim dos dados.

O programa em C seguinte implementa do Algoritmo 14.2, verificando se ainda há formas do arquivo analisando o valor de retorno de `fgets`.

```
/*
Processamento de um arquivo de formas (triângulos e retângulos) para cálculo
    das áreas
Requer: arquivo texto com nome 'figuras.txt' com cada linha contendo:
        - OU a letra T mais coordenadas dos vértices do triângulo
        - OU a letra R e as coordenadas de dois vértices opostos do
            retângulo
Assegura: a apresentação da área de cada forma
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    FILE *arquivo_formas = fopen("figuras.txt", "r");

    if (arquivo_formas == NULL) {
        printf("Falha ao abrir arquivo de formas.\n");
    }
    else {
        char forma[160];
        // Obtenção do número de formas
        while (fgets(forma, sizeof forma, arquivo_formas) != NULL) {
            double area;
            if (forma[0] == 'T') {
                // Triângulo: área por semiperímetro
                double x1, y1, x2, y2, x3, y3;
                sscanf(forma, "%*c%lf%lf%lf%lf%lf%lf",
                       &x1, &y1, &x2, &y2, &x3, &y3);
                double lado1 = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
                double lado2 = sqrt(pow(x1 - x3, 2) + pow(y1 - y3, 2));
                double lado3 = sqrt(pow(x2 - x3, 2) + pow(y2 - y3, 2));
                double semiperimetro = (lado1 + lado2 + lado3) / 2;

                area = sqrt(semiperimetro * (semiperimetro - lado1) *
                            (semiperimetro - lado2) * (semiperimetro - lado3));
            }
            else {
                // Retângulo: ára por base x altura
                double x1, y1, x2, y2;
                sscanf(forma, "%*c%lf%lf%lf%lf", &x1, &y1, &x2, &y2);

                area = fabs(x2 - x1) * fabs(y2 - y1);
            }

            printf("%.2f\n", area);
        }

        fclose(arquivo_formas);
    }

    return 0;
}
```

```
11.00
11.75
7.14
1.00
4.00
0.50
```

O arquivo `figuras.txt` processado por esse programa contém o conteúdo seguinte. Comparativamente ao usado na Seção 14.3, o formado de descrição de cada forma é o mesmo, sendo que apenas a primeira linha contendo o número de formas não existe.

```
R 0.0 0.0 5.5 2.0
T -1.0 1.0 3.0 4.5 4 -0.5
R -1.2 5.2 2.2 3.1
R 0 0 1 1
R -1 -1 1 1
T 0 0 0 1 1 0
```

Durante a execução, o resultado de `fgets` é checado a cada início do `while`, até que retorne `NULL` e encerre a repetição. A cada retorno válido, a área correspondente é calculada.

14.7 A função `rewind`

A palavra *rewind* tem aqui o sentido de rebobinar e vem do tempo em que os arquivos eram mantidos em fitas magnéticas e, para voltar para o início do arquivo, era necessário retroceder fisicamente o carretel. Para o contexto atual, rebobinar significa apenas retornar a posição corrente para o byte zero, ou seja, o primeiro byte do arquivo.

A função `rewind` tem o significado de retornar o arquivo para a primeira posição. Isso significa, por exemplo, que se um arquivo tem que ser processado mais que uma vez, é possível fazer uma varredura até o final e, então, retornar ao início e fazer nova leitura.

Um uso interessante de `rewind` é apresentado no Algoritmo 14.3.

Algoritmo 14.3: Processamento de arquivo de notas.

Descrição: Processamento de um arquivo contendo notas de provas para determinação da quantidade de notas maiores que a média geral de notas
Requer: um arquivo texto contendo uma nota por linha, com mínimo de uma linha
Assegura: a apresentação do número de notas maiores que a média de todas as notas

▷ *Cálculo da média geral*
enquanto existem nota no arquivo **faça**
  Obtenha uma nota
  Acumule a nota em *soma_notas*
  Conte essa nota usando *contador_notas*
fim enquanto
Calcule a *média* como *soma_notas*/*contador_notas*

▷ *Contagem do número de notas maiores que a média calculada*
Retorne ao início do arquivo
enquanto existem nota no arquivo **faça**
  Obtenha uma nota
  Conte essa nota em *contador_maiores* apenas se ela for maior que *média*
fim enquanto

▷ *Apresentação do resultado*
Apresente o valor de *contador_maiores*

Esse problema requer duas passadas pelos valores das notas, uma vez que a contagem dos valores maiores que a média precisa que a média já tenha sido calculada e, para calcular a média, é preciso ter passado por todos as notas. Assim, o arquivo é varrido sequencialmente somando-se as notas e contando quantas são para, em seguida, calcular a média. Em um segundo passo, o arquivo é varrido novamente desde o início, para, então, contar a quantidade de notas maiores que a média.

Como arquivo de notas, segue uma possível versão de `notas.txt`.

```
5.4
3.4
5.6
9.8
5.0
```

```
3.8
4.4
4.9
2.5
4.8
4.9
```

A implementação em C para o processamento do arquivo é apresentada na sequência.

```c
/*
Determinação da quantidade de notas maiores que a média geral das notas
    em um arquivo de dados
Requer: arquivo texto com uma nota por linha
Assegura: a apresentação da quantidade notas maiores que a média geral
*/
#include <stdio.h>

int main(void) {
    FILE *arquivo_notas = fopen("notas.txt", "r");

    if (arquivo_notas == NULL) {
        printf("Falha ao abrir arquivo de notas.\n");
    }
    else {
        // Cálculo da média geral das notas
        char entrada[160];
        double soma_notas = 0;
        int contador_notas = 0;
        while (fgets(entrada, sizeof entrada, arquivo_notas) != NULL) {
            double nota;
            sscanf(entrada, "%lf", &nota);
            soma_notas += nota;
            contador_notas++;
        }
        double media = soma_notas / contador_notas;

        // Contagem da quantidade de notas acima da média calculada
        rewind(arquivo_notas);  // retorna ao início do arquivo
        int contador_maiores = 0;
        while (fgets(entrada, sizeof entrada, arquivo_notas) != NULL) {
            double nota;
            sscanf(entrada, "%lf", &nota);
            if (nota > media)
                contador_maiores++;
        }

        fclose(arquivo_notas);

        // Resultado
        printf("%d notas maiores que %.1f.\n", contador_maiores, media);
    }

    return 0;
}
```

```
4 notas maiores que 5.0.
```

Sem o uso de `rewind`, o segundo `while` terminaria de imediato, visto que a posição corrente já está no fim do arquivo e qualquer nova leitura retornará `NULL` como resultado.

14.8 A traiçoeira função `feof`

O nome "*feof*" é composto pelo *f* inicial, como (quase) todas as funções que trabalham com arquivos (*files*), seguido de *eof*, que significa *end of file*, ou fim de arquivo.

Porém, o uso correto dessa função requer entender tanto o que ela faz e o que ela não faz. Apesar do nome, `feof` não verifica se o arquivo chegou ao fim. Isso é o que ela não faz.

A função `feof` deve ser usada depois que uma operação de leitura (`fgets`, por exemplo) tenta ser executada mas falha. Somente depois dessa falha é que `feof` pode ser usada para verificar porquê a função não foi bem sucedida. Assim, `feof` volta verdadeiro quando a razão da falha foi tentar ler além do último dado do arquivo.

A diferença parece sutil, mas não é. Para entender esse mecanismo, um programa simples é usado para mostrar o que acontece na leitura de um arquivo texto e como `feof` se comporta.

O arquivo usado para exemplo possui apenas duas linhas e é apresentado a seguir.

```
Texto da primeira linha
E material para a segunda linha!
```

Esse arquivo possui duas linhas, ambas terminadas com `\n`, como pode ser observado ao se analisar mais detalhadamente o conteúdo do arquivo.

```
$ od -tc -Ad texto.txt
0000000   T   e   x   t   o       d   a       p   r   i   m   e   i   r
0000016   a       l   i   n   h   a  \n   E       m   a   t   e   r   i
0000032   a   l       p   a   r   a       a       s   e   g   u   n   d
0000048   a       l   i   n   h   a   !  \n
0000057
```

O programa usado para processar esse arquivo está apresentado na sequência.

```
/*
Programa exemplo do comportamento da função feof
Assegura: a apresentação de informações relevantes ao longo da execução
*/
#include <stdio.h>

int main(void) {
    FILE *arquivo = fopen("texto.txt", "r");

    if (arquivo == NULL)
        printf("Falha ao abrir arquivo com texto.");
    else {
        char linha[160];

        // Situação inicial
        printf("Ao abrir o arquivo: feof = %c.\n\n", feof(arquivo) ? 'V' : 'F');

        // Primeira leitura
        fgets(linha, sizeof linha, arquivo);
        printf("Primeira leitura: %s", linha);
        printf("Depois da 1ª leitura: feof = %c.\n\n",
               feof(arquivo) ? 'V' : 'F');

        // Segunda leitura
        fgets(linha, sizeof linha, arquivo);
        printf("Segunda leitura: %s", linha);
        printf("Depois da 2ª leitura: feof = %c.\n\n",
               feof(arquivo) ? 'V' : 'F');

        // Terceira leitura, que deve falhar
        fgets(linha, sizeof linha, arquivo);
        printf("Terceira leitura: %s", linha);
        printf("Depois da 3ª leitura: feof = %c.\n\n",
               feof(arquivo) ? 'V' : 'F');

        fclose(arquivo);
    }
    return 0;
}
```

```
Ao abrir o arquivo: feof = F.

Primeira leitura: Texto da primeira linha
Depois da 1ª leitura: feof = F.

Segunda leitura: E material para a segunda linha!
Depois da 2ª leitura: feof = F.

Terceira leitura: E material para a segunda linha!
Depois da 3ª leitura: feof = V.
```

Como esperado, as duas primeiras execuções de `fgets` fazem as leituras das duas primeiras linhas do arquivo e apresentam seu conteúdo na tela. Como o arquivo `texto.txt` possui apenas essas duas linhas, a terceira chamada a `fgets` falha, sendo o `NULL` retornado ignorado no programa. Como a leitura falha, a variável `linha` não é modificada e é por essa razão que a mensagem de conteúdo da terceira leitura aparece igual ao da segunda.

Quanto à função `feof`, ela retorna falso (zero) logo ao abrir o arquivo e também depois da primeira leitura. Na segunda execução de `fgets`, a segunda linha é copiada para a variável `linha` e efetivamente a posição corrente está no fim do arquivo. Entretanto, como a leitura não foi além do fim do arquivo, `feof` ainda retorna falso. Na terceira leitura, que é mal sucedida por não haver mais o que ler do arquivo, o `fgets` não faz a leitura e, a partir desse momento, `feof` começa a retornar verdadeiro (valor não nulo), indicando que o problema foi tentar ler de um arquivo que já havia terminado.

Cuidado

Por uma tendência em interpretar erroneamente o que `feof` realmente verifica, não é incomum programas como o seguinte, que produz um resultado insatisfatório.
Supondo a existência de um arquvio `inteiros.txt` com o conteúdo seguinte, um programa se propõe a fazer a soma dos valores inteiros contidos nele. Para este arquivo específico, o resultado esperado é 120.

```
20
50
30
20
```

Este arquivo possui um `\n` no final de cada linha. Como ilustra o comando seguinte.

```
$ od -tc -Ad inteiros.txt
0000000   2   0  \n   5   0  \n   3   0  \n   2   0  \n
0000012
```

Segue, agora, o programa que tenta processar esse arquivo.

```c
/*
Tentativa incorreta de somar os valores de um arquivo de inteiros
Requer: um arquivo com valores inteiros, um por linha
Assegura: a soma incorreta dos valores
*/
#include <stdio.h>

int main(void) {
    FILE *arquivo_inteiros = fopen("inteiros.txt", "r");

    if (arquivo_inteiros == NULL)
        printf("Falha ao abrir arquivo_inteiros com os valores inteiros.");
    else {
        int soma = 0;
        while (!feof(arquivo_inteiros)) {
            // Obtenção do inteiro
            char entrada[160];
            fgets(entrada, sizeof entrada, arquivo_inteiros);
            int valor;
            sscanf(entrada, "%d", &valor);

            // Acumulação do valor
            soma += valor;
        }

        fclose(arquivo_inteiros);

        printf("Soma dos valores no arquivo: %d.\n", soma);
    }

    return 0;
}
```

```
Soma dos valores no arquivo: 140.
```

O laço de repetição é baseado na verificação de `feof`. Porém, essa função ainda retorna verdadeiro depois da leitura da última linha, pois o `fgets` não tentou ler depois do fim do arquivo. A consequência disso é que o `while` faz uma repetição a mais e o último valor lido é somado mais uma vez.

Uma alternativa interessante é evitar o uso do `feof`, baseando as decisões no resultado da função de leitura. Segue, assim, uma versão robusta do programa para a soma dos valores do arquivo.

```c
/*
Tentativa incorreta de somar os valores de um arquivo de inteiros
Requer: um arquivo com valores inteiros, um por linha
Assegura: a soma incorreta dos valores
*/
#include <stdio.h>

int main(void) {
    FILE *arquivo_inteiros = fopen("inteiros.txt", "r");

    if (arquivo_inteiros == NULL)
        printf("Falha ao abrir arquivo_inteiros com os valores inteiros.");
    else {
        char entrada[160];

        int soma = 0;
        while (fgets(entrada, sizeof entrada, arquivo_inteiros) != NULL) {
            int valor;
            sscanf(entrada, "%d", &valor);

            soma += valor;
        }

        fclose(arquivo_inteiros);

        printf("Soma dos valores no arquivo: %d.\n", soma);
    }

    return 0;
}
```

```
Soma dos valores no arquivo: 120.
```

15 Desvirtuação das repetições

A estrutura do `for`, embora seja bem objetiva indicando iniciação, condição e incremento (seus três elementos de operação), ela é bastante flexível do ponto de vista de seu uso pelo programador.

Um indicador dessa liberdade dada ao programador é que todos os elementos são opcionais, além de permitirem outros elementos.

15.0.1 Laços infinitos

Em termos de argumentos opcionais, o trecho de código seguinte pode ser considerado.

```
for (;;)
    printf("Faça algo!\n");
```

Para esta repetição, não há iniciação, nem condição, nem incremento. Seu comportamento, assim, funciona como um “repita o comando para sempre”, ou seja, um laço infinito.

Dica

Laços infinitos, embora não se apresentem como uma solução algorítmica em si, não são incomuns em programas. Por exemplo, o programa que gerencia um aparelho eletrônico simples como um forno de micro-ondas não foi feito para terminar: ele inicia ao se ligar o aparelho na tomada e só para quando o plugue for retirado.
Embora o esquema `for (;;)` seja usado, recomenda-se que laços infinitos usem `while (true)` para dar essa ênfase.

Na prática, basta não ter a condição de continuidade especificada para se ter a repetição infinita. Segue outro exemplo simples com repetições infinitas usando o `for`, que faz a contagem cíclica de 0 a 9 (contagem modular).

```
for (int i = 0;; i = (i + 1) % 10)
    printf("%d ", i);
```

15.0.2 Término forçado `for`

Enquanto se escreve um programa, o uso da repetição com `for` intuitivamente indica que um determinado número de repetições vai ocorrer. Muitos programadores deturpam essa percepção artificialmente interrompendo a repetição.

Ao considerar o código seguinte, a expectativa é que a leitura seja feita cinco vezes. Na repetição, porém, há um `if` que altera o valor de `i` de forma a terminar a repetição.

```c
/*
Leitura e apresentação de valores inteiros
Requer: uma sequência de até 5 valores inteiros não negativos ou uma
    sequência encerrada por um valor negativo
Assegura: a apresentação de cada valor lido na tela
*/
#include <stdio.h>

int main(void) {
    printf("Digite até 5 valores inteiros não negativos:\n");
    for(int i = 0; i < 5; i++) {
        char entrada[160];
        printf("> ");
        fgets(entrada, sizeof entrada, stdin);
        int valor;
        sscanf(entrada, "%d", &valor);

        printf("  +-- digitado %d.\n", valor);
        if (valor < 0)
            i = 5;
    }

    return 0;
}
```

```
Digite até 5 valores inteiros não negativos:
> 10
  +-- digitado 10.
> 0
  +-- digitado 0.
> -5
  +-- digitado -5.
```

O programa não está incorreto no sentido de que encerra as leituras ao encontrar um valor negativo. O problema é que esse encerramento prematuro em função do valor lido é mascarado e exige por parte de outro programador que analise os comandos uma atenção extra para entender que um “para i de 1 até 5” não será sempre de 1 a 5.

Em programas mais complexos e com mais variáveis, o uso desse expediente de término forçado pode ficar mascarado o suficiente para que outro programador, ao fazer uma modificação, sequer note essa possibilidade e introduza um erro ou instabilidade no código.

Neste caso, como o número de vezes da repetição é variável, o código ficaria mais claro usando o `do while`, por exemplo. Assim, não há indução no código a um comportamento que o programa não tem e deixa explícitas as duas condições de parada da repetição.

```c
/*
Leitura e apresentação de valores inteiros
Requer: uma sequência de até 5 valores inteiros não negativos ou uma
    sequência encerrada por um valor negativo
Assegura: a apresentação de cada valor lido na tela
*/
#include <stdio.h>

int main(void) {
    printf("Digite até 5 valores inteiros não negativos:\n");
    int i = 0;
    int valor;
    do {
        char entrada[160];
        printf("> ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%d", &valor);

        printf("  +-- digitado %d.\n", valor);
        i++;
    } while (valor >= 0 && i < 5);
```

```
    return 0;
}
```

```
Digite até 5 valores inteiros não negativos:
> 10
  +-- digitado 10.
> 0
  +-- digitado 0.
> -5
  +-- digitado -5.
```

15.1 Laços `while` infinitos, mas nem tanto

Não é incomum encontrar códigos em repositórios que utilizem falsos laços infinitos. Segue um programa exemplo desse recurso.

```
/*
Leitura e apresentação de valores não negativos
Requer: uma sequência de 0 ou mais de inteiros não negativos seguida
    por um valor negativo usado como sentinela
Assegura: a apresentação de cada valor da sequência na tela
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    while (true) {
        char entrada[160];
        printf("> ");
        int valor;
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%d", &valor);
        if (valor < 0)
            break;

        printf("  +-- digitado %d.\n", valor);
    }

    return 0;
}
```

```
> 12
  +-- digitado 12.
> 100
  +-- digitado 100.
> 18
  +-- digitado 18.
> 1
  +-- digitado 1.
> -5
```

Embora o `while (true)` seja um indicador sutil de que talvez não se espere uma repetição infinita, ele requer que o código seja analisado para identificar onde a condição de parada ocorre. No contexto, o comando `break` interrompe sumariamente a repetição, desestruturando o código.

> Dica
>
> Somente se deve usar o `brake` para interromper repetições quando essa for realmente a melhor opção, ou seja, quando for uma exceção! Caso contrário, o emprego de `while` e `do while` são opções melhores.

O programa seguinte é uma versão estruturado com a mesma funcionalidade usando uma condição explícita no `while`.

```
/*
Leitura e apresentação de valores não negativos
Requer: uma sequência de 0 ou mais de inteiros não negativos seguida
    por um valor negativo usado como sentinela
Assegura: a apresentação de cada valor da sequência na tela
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    // Primeira leitura
    printf("> ");
    fgets(entrada, sizeof entrada, stdin);
    int valor;
    sscanf(entrada, "%d", &valor);

    while (valor >= 0) {
        printf("  +-- digitado %d.\n", valor);

        // Próxima leitura
        printf("> ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%d", &valor);
    }

    return 0;
}
```

```
> 12
  +-- digitado 12.
> 100
  +-- digitado 100.
> 18
  +-- digitado 18.
> 1
  +-- digitado 1.
> -5
```

Uma versão com `do while` é apresentada na sequência, com destaque de que a apresentação do valor na tela é feita somente para valores não negativos.

```
/*
Leitura e apresentação de valores não negativos
Requer: uma sequência de 0 ou mais de inteiros não negativos seguida
    por um valor negativo usado como sentinela
Assegura: a apresentação de cada valor da sequência na tela
*/
#include <stdio.h>

int main(void) {
    char entrada[160];

    int valor;
    do {
        printf("> ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%d", &valor);

        if (valor >= 0)
            printf("  +-- digitado %d.\n", valor);
    } while (valor >= 0);

    return 0;
}
```

```
> 12
  +-- digitado 12.
> 100
  +-- digitado 100.
```

```
> 18
  +-- digitado 18.
> 1
  +-- digitado 1.
> -5
```

15.2 O comando `break` nas repetições

O uso do `break` para encerrar repetições não é, em si, um erro de programação ou uma falha em si. Em programas mais complexos, com aninhamento de repetições, a indicação explícita de todas as condições que mantém ou encerram uma repetição pode tornar o código difícil de entender.

O impacto no código das interrupções sumárias com `break` podem ser atenuadas usando-se um recurso de programação interessante: variáveis lógicas. As variáveis lógicas acrescentam ao código uma informação importante, que é o significado da variável dado por seu nome.

Uma versão de um programa para a leitura de inteiros até encontrar um valor negativo como o usado na Seção 15.1 é apresentado na sequência.

```c
/*
Leitura e apresentação de valores não negativos
Requer: uma sequência de 0 ou mais de inteiros não negativos seguida
    por um valor negativo usado como sentinela
Assegura: a apresentação de cada valor da sequência na tela
*/
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char entrada[160];

    int valor;
    bool encontrou_sentinela = false;
    do {
        printf("> ");
        fgets(entrada, sizeof entrada, stdin);
        sscanf(entrada, "%d", &valor);

        if (valor < 0)
            encontrou_sentinela = true;
        else
            printf("  +-- digitado %d.\n", valor);
    } while (!encontrou_sentinela);

    return 0;
}
```

```
> 12
  +-- digitado 12.
> 100
  +-- digitado 100.
> 18
  +-- digitado 18.
> 1
  +-- digitado 1.
> -5
```

Com o nome adequado, uma variável lógica somente acrescenta significado ao código: o `do while`, por exemplo, pode ler entendido como “faça a leitura e apresentação enquanto não encontrar sentinela”.

Para os programadores que evitam novas variáveis para economizar memória (preocupação legítima!), sempre se deve também considerar a clareza do código. Além disso, uma variável `bool` em uma máquina com 8GiB de memória principal consome 0,0000002% do total disponível[1].

[1] O cálculo considera que `sizeof (bool)` é um byte, valor usado pelo compilador durante a escrita desse livro.

Parte V

Cadeias de caracteres

16 Dados textuais em C

A linguagem C não é uma linguagem simples para trabalhar com cadeias de caracteres, as chamadas *strings*. O suporte para textos na linguagem é feito por constantes inseridas diretamente no código ou manipuladas em variáveis simples ou compostas do tipo `char`.

Este capítulo apresenta uma visão inicial das constantes textuais da linguagem e como fazer referências a essas constantes.

16.1 Revisitando as constantes textuais

No Capítulo 3 foram apresentados os principais tipos de dados da linguagem C, incluindo os dados textuais, mas ainda é preciso complementar as informações sobre os tipos literais.

A linguagem C apresenta duas notações para especificar valores textuais. A primeira é para caracteres únicos, como uma letra ou um símbolo de pontuação, e é expressa usando aspas simples, como `'A'` ou `'@'`, por exemplo.

O programa seguinte ilustra três escritas usando o formato `%c` (caractere) para apresentar cada valor.

```
/*
Apresentação de caracteres simples
Assegura: escrita de caracteres únicos
*/
#include <stdio.h>

int main(void) {
    printf("Uma letra simples: %c.\n", 'f');
    printf("Um símbolo de pontuação: %c.\n", ';');
    printf("O espaço: %c.\n", ' ');

    return 0;
}
```

```
Uma letra simples: f.
Um símbolo de pontuação: ;.
O espaço:  .
```

A segunda forma de expressar constantes literais usa aspas duplas e indica uma sequência de caracteres, como `"linguagem"` e `"compilador gcc"`. O `printf` usa a especificação de formato `%s` para substituir cadeias de caracteres.

```
/*
Apresentação de cadeias de caracteres
Assegura: escrita de sequências de caracteres
*/
#include <stdio.h>

int main(void) {
    printf("Um texto: %s.\n", "Era uma vez...");
    printf("Texto com uma tabulação: %s\n", "n =\t20");

    return 0;
}
```

```
Um texto: Era uma vez....
Texto com uma tabulação: n =        20
```

A principal diferença entre um caractere simples e uma cadeia de caracteres é, além do evidente número de caracteres em si, é a representação. Enquanto caracteres simples apenas são representados apenas por um valor, a cadeia usa uma sequência de caracteres simples seguida pelo terminador `\0`, chamado caractere nulo e que é representado com todos os bits iguais a zero. Na prática, quando o compilador encontra uma constante como `"programa"` em um código, ele armazena esse valor como `programa\0`, ou seja, sempre com um byte nulo depois do texto em si. Para explicitar um pouco mais, `'Y'` é só o `Y`, enquanto `"Y"` é `Y\0`.

16.2 Armazenamento de caracteres simples: o tipo `char`

Para criar uma variável para armazenar um único caractere deve ser usado o tipo `char`. Segue um programa simples ilustrando o conceito.

```c
/*
Armazenamento em um char
Assegura: escrita dos valores armazenados
*/
#include <stdio.h>

int main(void) {
    char pontuacao = '?';
    char letra = 'A';

    printf("Qual é a letra%c Resposta: %c!\n", pontuacao, letra);

    printf("Quantidade de bytes em um char: %zu.\n", sizeof letra);

    return 0;
}
```

```
Qual é a letra? Resposta: A!
Quantidade de bytes em um char: 1.
```

A linguagem C usa o tipo `char` para caracteres mas, na prática, trata esse valor como um valor inteiro. Apenas a apresentação com o `printf` é que escolhe usar o caractere como valor apresentado.

Seguem dois programas para mostrar que `char` e `int` são muito próximos (mas não iguais!).

```c
/*
Apresentação do char como caractere e como inteiro
Assegura: apresentação de um char e seu valor associado
*/
#include <stdio.h>

int main(void) {
    char letra = 'Z';
    printf("O caractere %c é armazenado usando o valor decimal %d.\n",
           letra, letra);

    int valor = 90;
    printf("O valor decimal %d pode ser visto como o caractere %c.\n",
            valor, valor);

    return 0;
}
```

```
O caractere Z é armazenado usando o valor decimal 90.
O valor decimal 90 pode ser visto como o caractere Z.
```

Nesse programa, o valor de `letra`, que é do tipo `char`, é apresentado usando-se tanto a interpretação como um caractere, com `%c`, quanto como um valor decimal, com `%d`. O mesmo é feito para a variável inteira `valor`. Nenhum erro ou aviso é emitido pelo compilador.

```c
/*
Apresentação de manipulações curiosas com char
Assegura: apresentação de resultados de manipulação
*/
#include <stdio.h>

int main(void) {
    char letra = 'A';
    printf("A letra atual é %c. ", letra);
    printf("Depois dela vem o %c.\n", letra + 1);

    printf("\nAlfabeto minúsculo: ");
    for (char letra = 'a'; letra <= 'z'; letra++)
        printf("%c", letra);
    printf("\n");

    printf("Alfabeto maiúsculo: ");
    for (char letra = 'A'; letra <= 'Z'; letra++)
        printf("%c", letra);
    printf("\n");

    printf("\nA quantidade de letras de %c até %c é %d.\n",
           'H', 'T', 'T' - 'H' + 1);

    printf("\nA letra que fica no meio de %c e %c é %c.\n",
           'G', 'K', ('G' + 'K') / 2);

    return 0;
}
```

```
A letra atual é A. Depois dela vem o B.

Alfabeto minúsculo: abcdefghijklmnopqrstuvwxyz
Alfabeto maiúsculo: ABCDEFGHIJKLMNOPQRSTUVWXYZ

A quantidade de letras de H até T é 13.

A letra que fica no meio de G e K é I.
```

Reforçando a associação de `char` com um valor inteiro, o programa exemplifica operações como `letra + 1` e incrementos como `letra++`. Adicionalmente há ainda uma expressão interessante: `('G' + 'K') / 2`, que soma `'G'` (71) com de `'K'` (75) e divide o resultado por 2 (divisão inteira), resultando em 73, que é `'I'`.

Curiosidade

Cada letra possui um valor específico associado a ela, assim como cada um dos outros caracteres. Por exemplo, `'A'` é o 65, `'\n'` é o 10 e `' '` (espaço) é o 32.
Como esperado, se `'A'` é 65, `'B'` é 66, `'C'` é 67 e assim por diante. Portanto, é possível verificar se `letra1 <= letra2`, pois uma comparação de inteiros é feita.
Há um "porém" nessa história: `'Z'` é 90, mas `'a'` é 97. Na realidade, as minúsculas se iniciam no 97 e vão até o 122. Isso gera uma situação confusa, pois C tem certeza que `'a' > 'Z'` é verdadeiro.

Dica

Embora haja um prejuízo quanto á clareza, o tipo `char` pode ser usado como um "pequeno inteiro" de um byte de comprimento. Dessa forma, usando o primeiro bit para indicar o sinal, uma variável desse tipo guardaria valores de -128 a 127. Caso necessário, pode-se usar a declaração de um `unsigned char` para ter valores de 0 até 255.
Para se ter programas mais claros, entretanto, mesmo nesse caso é melhor usar os tipos de comprimento fixo definidos em `stdint.h`. Assim, para se ter um inteiro com sinal de oito bits, o tipo seria `int8_t`, e poderia ser usado o `uint8_t` para o mesmo comprimento sem o sinal. O tipo `char` deve ser deixado de fato apenas para caracteres. O uso de `stdint.h` é abordado no **?@sec-arquivos-binarios**.

16.2.1 Caracteres acentuados

Na década de 1960 cada sistema computacional usava sua própria tabela para associar um dado caractere a um valor numérico. Para que os diferentes sistemas pudessem trocar informações, foi elaborada em 1964 uma codificação padronizada denominada *American Standard Code for Information Interchange*, ou tabela ASCII. Essa tabela definia tanto os caracteres de controle (`\n` ou `\t`, por exemplo) quanto os caracteres legíveis (letras, dígitos e pontuação).

Na prática, tanto os caracteres de controle quanto os símbolos legíveis significativos poderiam ser representados com apenas sete bits. Em sistemas com palavras de oito bits, o primeiro bit era usualmente zero, de forma que praticamente metade dos bytes não tinham uso. O ponto em questão é que o conjunto de caracteres era baseado na lingua inglesa, na qual acentuações ou outros símbolos de outras línguas não eram incluídos, como `á`, `ã`, `ç`, `ß` (alemão) ou `č` (esloveno). Os bytes não usados na codificação ASCII (aqueles cujos bits começavam com 1) eram usados para esse fim, cada sistema usando uma codificação específica e particular para atender suas necessidades.

Em grande parte dos sistemas atuais é empregado o Unicode[1], que se propõe a ter representações para todas as línguas do planeta e usa com frequência a codificação UTF-8 para representar os símbolos (letras, ideogramas, emojis) em bytes. Como exemplo, o símbolo monetário do Euro é designado por `U+20AC` no Unicode e usa a sequência de bytes `E282AC` para representação em UTF-8.

O problema que surge dessa representação é que UTF-8 usa uma quantidade de bytes variável conforme o símbolo. Os caracteres ASCII usam um único byte e possuem representação igual, o que mantém a compatibilidade. Outros símbolos, porém usam dois, três ou até quatro bytes, o que torna impossível armazená-los em uma variável `char`.

```c
/*
Incompatibilidade de símbolos Unicode/UTF-8
*/
#include <stdio.h>

int main(void) {
    char c = 'é';
    printf("c = %c.\n", c);

    return 0;
}
```

```
main.c: In function ´main´:
main.c:7:14: warning: multi-character character constant [-Wmultichar]
    7 |     char c = 'é';
      |              ^~~
main.c:7:14: warning: overflow in conversion from ´int´ to ´char´
changes value from ´50089´ to ´-87´ [-Woverflow]
```

[1] Unicode Consortium: https://home.unicode.org.

```
c = � .
```

A falha na compilação acima é que o caractere `é` (`U+00E9`) é codificado em dois bytes, `C3` e `A9`, dos quais apenas o valor `A9` (decimal 169) é guardado na variável `c`. Esse valor 169, sozinho, não representa um caractere UTF-8 válido.

Esse código fonte, armazenado no arquivo `acentuacao.c`, está codificado com UTF-8, como se indica com o comando `file`.

```
$ file acentuacao.c
acentuacao.c: C source, Unicode text, UTF-8 text
```

Existe a codificação de caracteres ISO-8859, conhecida como *latin1*, que inclui os caracteres latinos acentuados e que usa apenas um byte por caractere. O comando `iconv` pode ser usado para criar um novo arquivo fonte (`acentuacao-latin1.c`) com essa codificação, conforme segue.

```
$ iconv -f utf8 -t latin1 acentuacao.c > acentuacao-latin1.c
$ file acentuacao-latin1.c
acentuacao-latin1.c: C source, ISO-8859 text
```

Como o caractere `é` usado no código fonte agora possui um único byte, ele pode ser guardado em um `char` e a compilação ocorre sem problemas.

```
$ gcc -Wall -pedantic -std=c17 acentuacao-latin1.c
```

Ao executar o programa, como a saída produzida é ISO-8859, ela tem que ser convertida de volta para UTF-8 para que o terminal a exiba corretamente.

```
$ ./a.out | iconv -f latin1 -t utf8
c = é.
```

Nos programas exemplificados neste texto, simplesmente são evitados os casos em que um `char` armazenará um caractere Unicode com mais que um byte, pois todas as codificações de caractere usam UTF-8. Para efetivamente usar caracteres de múltiplos bytes, C disponibiliza uma série de funções em `wchar.c`, as quais lidam com os "caracteres largos" (*wide characters*). Porém, o uso dessas funções não é tratado neste livro.

16.3 Acesso às cadeias de caracteres constantes

Nesta seção é apresentada uma visão básica sobre cadeias de caracteres em C, sendo que a manipulação de variáveis com conteúdo textual é coberta pelo Capítulo 17.

Em programas escritos em C, cadeias de caracteres são indicadas entre aspas duplas e, internamente, é acrescido um byte nulo para indicar seu fim. Quando uma constante literal é parte de um programa, ela é incluída na seção de dados do arquivo executável.

Um exemplo trivial dessa inclusão pode ser vista com um código simples, como o seguinte.

```
/*
Apresentação de mensagens simples
Assegura: apresentação de duas mensagens na tela
*/
#include <stdio.h>

int main(void) {
    printf("Bom dia!\n");
    printf("Boa noite.\n");

    return 0;
}
```

```
Bom dia!
Boa noite.
```

Esse programa gera um executável denominado `a.out`, como observado pelos comandos que seguem.

```
$ ls -l a.out
-rwxr-xr-x 1 jander jander 16144 ago 24 11:40 a.out
$ file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=911559c75a4cd72916890eee142aaa2bdb16bb7c, for GNU/Linux 3.2.0,
not stripped
```

O comando `strings` usado na sequência mostra as cadeias de caracteres detectadas no arquivo `a.out` (o `egrep` é usado para remover outras linhas, de forma a reduzir o tamanho da saída).

```
$ strings a.out | egrep -v '(^\.|^_)'
/lib64/ld-linux-x86-64.so.2
puts
printf
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
PTE1
u+UH
%c%s%c
Bom dia!
Boa noite.
;*3$"
GCC: (Debian 12.2.0-14) 12.2.0
Scrt1.o
crtstuff.c
deregister_tm_clones
completed.0
frame_dummy
main.c
puts@GLIBC_2.2.5
printf@GLIBC_2.2.5
user_input_time
print_user_input
main
```

Como pode ser observado, `"Bom dia!"` e `"Boa noite."` fazem parte dos literais fisicamente presentes no arquivo.

Com base no fato de que as constantes fazem parte do arquivo executável e que quando o programa é colocado em execução elas são também carregadas para a memória, o programa seguinte mostra como fazer referência a essas cadeias.

```c
/*
Exemplificação de referência a constantes literais presentes em um programa
Assegura: apresentação de textos
*/
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *texto1 = "Um primeiro texto";
    char *texto2 = "Texto número 2";

    char *texto_selecionado;
    if (rand() % 100 < 50)
        texto_selecionado = texto1;
    else
        texto_selecionado = texto2;

    printf("1) %s\n", texto1);
    printf("2) %s\n", texto2);
    printf("\nSorteado: %s\n", texto_selecionado);

    return 0;
}
```

```
1) Um primeiro texto
2) Texto número 2

Sorteado: Texto número 2
```

Neste programa há dois textos importantes: `"Um primeiro texto"` e `"Texto número 2"`. Essas duas constantes textuais ficam em algum lugar do programa. Existem também as duas variáveis `texto1` e `texto2`, que são declaradas como sendo do tipo `char *`. O asterisco, nesse contexto, indica que as variáveis são referências aos textos e não que sejam um `char` comum. Dessa forma, `texto1` está referenciando a constante `"Um primeiro texto"`, por exemplo.

Essas variáveis que guardam referências às coisas recebem, em programação, o nome de ponteiros[2]. Como elas são usadas apenas como referência a um texto já existente, não podem ser diretamente usadas para outras manipulações, como para guardar um valor digitado pelo usuário ou tentar modificar a constante referenciada.

No jargão computacional, diz-se que "`texto1` aponta para a constante `"Um primeiro texto"`", da mesma forma que `texto2` aponta para a constante `"Texto número 2"`.

No código há ainda a variável `texto_selecionado`, também do tipo `char *`, cujo valor inicialmente é indefinido (lixo). A expressão `rand() % 100` resulta em um valor (pseudo)aleatório de 0 a 99. Assim, a referência ser guardada em `texto_selecionado` depende desse valor aleatório, com praticamente 50% de chance para cada caso. Se o valor aleatório for menor que 50, `texto_selecionado` guarda a mesma referência guardada em `texto1`, ou seja, ela também aponta para a constante `"Um primeiro texto"`

Nas chamadas a `printf`, a especificação de formato `%s` apresenta o texto apontado por cada variável. Desse modo, `printf("2) %s\n", texto2);` significa mostre o texto que `texto2` está apontando.

O uso desse recurso é relativamente simples de der feito, porém muito limitado, visto que apenas aceita mudar para qual constante cada variável aponta.

Para finalizar, é interessante ver que esse recurso foi empregado na implementação do Algoritmo 7.3.

[2]Esse tema é tratado em mais detalhes no Capítulo 20.

17 Manipulação de dados textuais em C

O uso de variáveis para guardar valores textuais tem sido usado em praticamente todos os programas até o momento e seu uso para leitura está descrito na Seção 4.5.1. Este capítulo apresenta com mais detalhes as cadeias de caracteres e introduz recursos para sua manipulação.

17.1 As cadeias de caracteres em variáveis

Variáveis declaradas como `char *` armazenam, como visto na Seção 16.3, apenas uma referência a uma cadeia de caracteres já existentes. Para que se tenha uma variável para guardar dados textuais de forma genérica, é preciso ter espaço para armazenar tanto os caracteres quanto o terminador `\0`.

Como C não dispões de um tipo nativo para o armazenamento de textos, a alternativa é o uso de um vetor de caracteres. A declaração seguinte ilustra uma declaração para armazenamento de um valor literal. Ela cria um espaço na memória capaz de armazenar 50 valores do tipo `char`, ou seja, 50 caracteres.

```
char texto[50];
```

Esse espaço reservado na declaração é fixo a partir da declaração da variável, ou seja, uma vez escolhido um comprimento, não é possível aumentá-lo ou diminuí-lo.

O que se faz nos programas é ter um espaço máximo, que é o tamanho declarado para o vetor de caracteres, porém usá-lo apenas parcialmente. Como exemplo, se na variável `texto` declarada com 50 posições estiver armazenado o texto "grandes responsabilidades", os caracteres do *g* inicial até o *s* final ocupam os primeiros 25 `char` dela, sendo o 26º caractere necessariamente um `\0`. Restam na variável 24 posições sem uso e seu conteúdo é ignorado.

Com essa estratégia de uso, o conteúdo textual pode ser aumentado pelo uso dos caracteres sobressalentes, deslocando-se o `\0` mais para o final do vetor, ou encolhido, quando o terminador ocupa uma posição mais para o início. O limite evidente é o espaço total reservado para o armazenamento, sempre se lembrando que deve haver um espaço para o terminador. Assim, a variável texto pode ter comprimento máximo de 49 `char`, pois tem que haver espaço para o `\0` final.

Desse modo, pelo uso desse mecanismo de armazenamento, mantém-se uma compatibilidade com as constantes usadas no código: se existe uma constante `"bom dia!"` ocupando nove bytes no programa, esse mesmo valor pode ser guardado em uma variável, ocupando as nove primeiras posições do vetor de caracteres.

17.2 Manipulação de conteúdo

Uma dificuldade imediata é que os vetores de caracteres não são uma variável simples, mas uma coleção de variáveis individuais tratadas como uma coisa só. Como não são variáveis como as declaradas como `int`, `double` ou mesmo `char` simples, não admitem atribuições ou manipulações diretas. Para tratar as variáveis textuais C provê uma série de funções, as quais podem ser usadas pela importação do cabeçalho `string.h`.

Dica

O arquivo de cabeçalho é `string.h`.

```
#include <string.h>
```

Existe, porém, outro arquivo de cabeçalhos chamado `strings.h` (no plural) e eles não podem ser confundidos.

Para que se tenha utilidade prática na manipulação de variáveis textuais, é preciso que se possa ver o comprimento do texto e fazer atribuições de valores, realizar concatenações e comparações.

17.2.1 Comprimento de cadeias de caracteres

Uma função que já foi usada anteriormente em diversos programas é a `strlen` (*string length*), que retorna o comprimento da cadeia de caracteres sem o `\0` final.

```
/*
Comprimento de cadeias de caracteres
Assegura: a apresentação de textos e respectivos comprimentos
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    char *texto1 = "C: a linguagem";
    printf("\"%s\" tem comprimento %zu.\n", texto1, strlen(texto1));

    char *texto2 = "";
    printf("\"%s\" tem comprimento %zu.\n", texto2, strlen(texto2));

    char *texto3 = "Programas estruturados";
    printf("\"%s\" tem comprimento %zu.\n", texto3, strlen(texto3));

    return 0;
}
```

```
"C: a linguagem" tem comprimento 14.
"" tem comprimento 0.
"Programas estruturados" tem comprimento 22.
```

17.2.2 Atribuições de cadeias de caracteres

O programa seguinte foca na questão da atribuição. Porém, esse código serve apenas para mostrar o que não funciona.

```
/*
Tentativa malsucedida de atribuir um valor a uma variável
Assegura: absolutamente nada (infelizmente)
*/
#include <stdio.h>

int main(void) {
    char texto[50];

    texto = "Bom dia!";
    printf("%s\n", texto);

    return 0;
}
```

```
main.c: In function ´main´:
main.c:10:11: error: assignment to expression with array type
   10 |     texto = "Bom dia!";
      |           ^
```

O humilde autor desse texto tem convicção que praticamente a totalidade dos programadores C do planeta desejariam que esse programa funcionasse. A realidade, porém, não é essa: o erro principal do código é exatamente a atribuição. Basicamente, `texto` é o nome para a coleção de caracteres e, na linguagem, não faz sentido armazenar nela uma constante.

O programa seguinte mostra como a função `strncpy` (*string copy*) pode ser usada para obter o mesmo efeito da atribuição.

```c
/*
Atribuição e apresentação de valor textual
Assegura: apresentação do valor guardado em uma string
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    char texto[50];

    strncpy(texto, "Bom dia!", sizeof texto - 1);
    printf("%s\n", texto);

    return 0;
}
```

```
Bom dia!
```

O processo da atribuição é copiar um dado valor para uma variável.

```c
strncpy(texto, "Bom dia!", sizeof texto - 1);
```

O primeiro parâmetro de `strncpy` é a variável que receberá o valor. O segundo parâmetro é o valor que será atribuído, que pode ser uma constante ou outra variável. Há, ainda, um terceiro parâmetro, que é o tamanho da variável, que é importante para garantir que o limite de espaço reservado para a variável será respeitado. Ultrapassar o tamanho da variável pode trazer consequências imprevisíveis à execução.

> **Dica**
>
> Existem funções de manipulação de cadeias de caracteres que não verificam o tamanho da variável destino. É importante nunca usá-las.

17.2.3 Concatenação de cadeias de caracteres

Além da cópia de uma sequência de caracteres para uma variável, é possível estender seu conteúdo, fazendo a concatenação. A função `strncat` (*string catenation*) faz a cópia de uma cadeia anexando-a a outra.

```c
/*
Criação de uma única linha a partir de leituras separadas
Requer: Uma sequência de linhas de texto
Assegura: apresentação da concatenação de todas as linhas em uma só
*/
#include <stdio.h>
```

```c
#include <string.h>
#include <stdbool.h>

int main(void) {
    char texto[160];
    char linha_unica[5000];

    // Inicia uma cadeia vazia
    strncpy(linha_unica, "", sizeof linha_unica - 1);

    // Acrescenta cada linha digitada à linha única
    printf("Digite linhas, usando Ctrl-D para encerrar:\n");
    bool primeira_palavra = true;
    while (fgets(texto, sizeof texto, stdin) != NULL) {

        texto[strlen(texto) - 1] = '\0';
        if (primeira_palavra)
            primeira_palavra = false;
        else
            strncat(linha_unica, " ",
                    sizeof linha_unica - strlen(linha_unica) - 1);

        strncat(linha_unica, texto,
                sizeof linha_unica - strlen(linha_unica) - 1);
    }

    // Texto total
    printf("\n\"%s\"\n", linha_unica);

    return 0;
}
```

```
Digite linhas, usando Ctrl-D para encerrar:
Era
uma
vez
uma linda menina
que possuia uma capa vermelha com
capuz.

"Era uma vez uma linda menina que possuia uma capa vermelha com capuz."
```

A função de concatenação acrescenta à variável do primeiro parâmetro (`linha_unica`) uma cópia dos caracteres contidos no segundo parâmetro (`texto`). O terceiro parâmetro é a quantidade máxima de caracteres a serem copiados, a qual é usada para respeitar o tamanho do espaço da variável destino. No código, a quantidade máxima para cada concatenação corresponde ao cálculo do espaço livre existente em `linha_unica`, respeitando um byte para ter espaço para o `\0` final.

17.2.4 Comparações de cadeias de caracteres

A comparação entre duas cadeias de caracteres é feita no sentido alfabético usual, sempre comparando os primeiros caracteres das duas cadeias e, sendo iguais, passando para os segundos, terceiros e assim por diante. Nas funções de `string.h`, a função `strncmp` (*string comparison*) tem a finalidade realizar as comparações.

A função `strncmp` retorna sempre um valor inteiro, como apresentado na Tabela 17.1.

Tabela 17.1: Resultados possíveis para a função `strncmp`.

Resultado	Significado
zero	Iguais
menor que zero	Anterior
maior que zero	Posterior

Resultado	Significado

Segue um programa ilustrando o uso de `strncmp`.

```
/*
Comparações de cadeias de caracteres
Requer: a digitação de pares de palavras em cada linha
Assegura: a apresentação da comparação entre as palavras de cada par
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    char linha[160];
    printf("Digite duas palavras por linha, usando Ctrl-D para encerrar:\n");
    while (fgets(linha, sizeof linha, stdin) != NULL) {
        char palavra1[80], palavra2[80];
        sscanf(linha, "%79s%79s", palavra1, palavra2);  // palavras simples

        if (strncmp(palavra1, palavra2, sizeof palavra1) == 0)
            printf("%s é igual a %s.\n", palavra1, palavra2);
        else if (strncmp(palavra1, palavra2, sizeof palavra1) < 0)
            printf("%s é anterior a %s.\n", palavra1, palavra2);
        else
            printf("%s é posterior a %s.\n", palavra1, palavra2);
    }

    return 0;
}
```

```
Digite duas palavras por linha, usando Ctrl-D para encerrar:
abacaxi abacaxi
abacaxi é igual a abacaxi.
marca marcas
marca é anterior a marcas.
vermelho azul
vermelho é posterior a azul.
azul vermelho
azul é anterior a vermelho.
contraponto contraindicado
contraponto é posterior a contraindicado.
```

Uma discussão sobre comparações de palavras acentuadas é apesentada na Seção 17.5.

17.3 Mais sobre declaração de variáveis textuais

Assim como variáveis aritméticas e lógicas, variáveis literais também podem ser iniciadas na declaração. Há formatos diferentes para essa iniciação.

O programa seguinte ilustra os diferentes formatos, os quais serão abordados individualmente na sequência.

```
/*
Exemplificação de declaração de cadeias de caracteres juntamente com iniciação
Assegura: apresentação dos textos atribuídos e do tamanho de cada variável
*/
#include <stdio.h>

int main(void) {
    // Referência a uma constante
    char *texto1 = "Um texto constante e fixo";
    printf("texto1 = '%s'.\n", texto1);

    // Variável com comprimento explícito
```

```
    char texto2[100] = "Uma variável de até 99 posições";
    printf("texto2 = '%s'.\n", texto2);

    // Variável com comprimento automático
    char texto3[] = "Variável com comprimento dependente do valor atribuído";
    printf("texto3 = '%s'.\n", texto3);

    // Comprimentos das variáveis
    printf("\ntexto1: %zu bytes, pois é apenas referência a uma constante.\n",
           sizeof texto1);
    printf("texto2: %zu bytes, conforme tamanho especificado.\n",
           sizeof texto2);
    printf("texto3: %zu bytes, pois depende do texto de iniciação.\n",
           sizeof texto3);

    return 0;
}
```

```
texto1 = 'Um texto constante e fixo'.
texto2 = 'Uma variável de até 99 posições'.
texto3 = 'Variável com comprimento dependente do valor atribuído'.

texto1: 8 bytes, pois é apenas referência a uma constante.
texto2: 100 bytes, conforme tamanho especificado.
texto3: 57 bytes, pois depende do texto de iniciação.
```

O primeiro formato é o utilizado na Seção 16.3, que declara uma variável que é apenas uma referência a um texto em outro lugar. No caso, esse texto é uma constante que o compilador insere no código fonte. Ao longo do código, a variável `texto1` pode ser alterada de forma a apontar para outro texto.

```
char *texto = "Texto";  // a variável é uma referência à constante "Texto"
```

Outra possibilidade é a mais usual, criando-se um vetor de caracteres com um tamanho predeterminado e, juntamente a isso, fazer a cópia de um valor para ele. Para o caso da variável `texto2` do programa, há a alocação de um espaço total de 100 bytes, permitindo um texto de até 99 bytes. Ao fazer a declaração, um cópia da constante textual é feita para a variável. Como `texto2` recebe uma cópia, os dados da variável podem ser alterados.

```
char *texto[20] = "Texto";  // a variável tem 20 posições, usando as 6
                            // primeiras para guardar "Texto" e \0
```

Finalmente, `texto3` é uma variável declarada como `texto2`, porém omitindo seu comprimento. Neste caso, a variável é criada com capacidade para guardar exatamente o valor atribuído, ou seja, o espaço para os caracteres e para o `\0` terminal.

```
char *texto[] = "Texto";  // a variável tem exatamente as 6 posições
                          // necessárias para guardar "Texto" e \0
```

É importante lembrar que as declarações que usam os colchetes, seja com tamanho explícito ou omitido, o espaço reservado para a variável deve sempre ser respeitado.

17.4 Acesso direto ao conteúdo

Uma forma prática de usar cadeias de caracteres é fazendo acesso a cada posição individual do vetor. Para isso, é possível indicar um `char` qualquer armazenado especificando sua posição no vetor.

O programa seguinte mostra como cada posição de uma cadeia de caracteres pode ser acessada.

```c
/*
Apresentação de um texto caractere a caractere
Requer: um texto digitado pelo usuário
Assegura: apresentação de cada caractere, posição a posição
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    printf("Digite um texto qualquer:\n");
    char texto[160];
    fgets(texto, sizeof texto, stdin);

    for (int posicao = 0; posicao < strlen(texto) - 1; posicao++)
        printf("%2d: '%c'\n", posicao, texto[posicao]);

    return 0;
}
```

```
Digite um texto qualquer:
Subi no ombro de gigantes!
 0: 'S'
 1: 'u'
 2: 'b'
 3: 'i'
 4: ' '
 5: 'n'
 6: 'o'
 7: ' '
 8: 'o'
 9: 'm'
10: 'b'
11: 'r'
12: 'o'
13: ' '
14: 'd'
15: 'e'
16: ' '
17: 'g'
18: 'i'
19: 'g'
20: 'a'
21: 'n'
22: 't'
23: 'e'
24: 's'
25: '!'
```

A variável `texto` possui um espaço de armazenamento para 160 caracteres simples. Cada uma dessas posições pode ser indica por um índice, que vai de zero a 159. Assim, o primeiro caractere do texto está em `texto[0]`, o segundo em `texto[1]` e o último em `texto[strlen(texto) - 1]`. No programa, o laço de repetição claramente se inicia na posição zero, mas é relevante ressaltar que a última posição apresentada na tela é a `strlen(texto) - 2`, o que evita escrever o `\n` que está na última posição.

> **Dica**
>
> Uma das grandes fontes de erro no acesso aos índices de uma cadeia de caracteres é errar nos limites dos índices. Não é incomum, em uma repetição, faltar uma posição ou passar uma posição. Atenção é sempre necessária nesse ponto.

Além das funções em `string.h`, há algumas interessantes em `ctype.h`, que permitem facilmente verificar caracteres. A Tabela 17.2 apresenta uma lista de várias funções interessantes.

Tabela 17.2: Funções para verificação de caracteres simples (`ctype.h`). Todas as funções retornam um inteiro não nulo em caso de sucesso ou zero, caso contrário.

Função	Significado
`isalpha(c)`	Verifica se `c` é um caractere alfabético (i.e., uma letra).
`isdigit(c)`	Verifica se `c` é um dígito de 0 a 9.
`isalnum(c)`	Verifica se `c` é alfanumérico, ou seja se é alfabético ou dígito.
`isascii(c)`	Verifica se `c` é um caractere ASCII do escopo de 7 bits.
`iscntrl(c)`	Verifica se `c` é um caractere de controle.
`islower(c)`	Verifica se `c` é uma letra minúscula.
`isupper(c)`	Verifica se `c` é uma letra maiúscula.
`isspace(c)`	Verifica se `c` é qualquer caractere de “espaço branco” (*white-space*), ou seja espaço, `\f`, `\n`, `\r`, ,͡ e `\v`.
`isblank(c)`	Verifica se `c` é um caractere “branco” (*blank*), o seja, se é espaço ou tabulação.
`isprint(c)`	Verifica se `c` é um caractere imprimível, incluindo o espaço.
`isgraph(c)`	Verifica se `c` é um caractere imprimível, exceto o espaço.
`ispunct(c)`	Verifica se `c` é um caractere imprimível que não seja um espaço ou um caractere alfanumérico.
`isxdigit(c)`	Verifica se `c` for hexadecimal digits, that is, one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.

Para exemplificar o uso de algumas funções, segue um programa que conta alguns itens de interesse em um texto digitado pelo usuário.

```c
/*
Contagem de maiúsculas, minúsculas, pontuação, dígitos e caracteres de
    controle em um texto
Requer: um texto digitado pelo usuário
Assegura: apresentação das contagens de maiúsculas, minúsculas, pontuação,
    dígitos e caracteres de controle presentes no texto
*/
#include <stdio.h>
#include <ctype.h>

int main(void) {
    printf("Digite um texto qualquer:\n");
    char texto[160];
    fgets(texto, sizeof texto, stdin);

    int contador_minusculas = 0;
    int contador_maiusculas = 0;
    int contador_pontuacao = 0;
    int contador_digitos = 0;
    int contador_controle = 0;

    int i = 0;
    while (texto[i] != '\0') {
        if (islower(texto[i]))
            contador_minusculas++;
        else if (isupper(texto[i]))
            contador_maiusculas++;
        else if (ispunct(texto[i]))
            contador_pontuacao++;
        else if (isdigit(texto[i]))
            contador_digitos++;
        else if (iscntrl(texto[i]))
            contador_controle++;

        i++;
    }

    printf("Maiúsculas: %d.\n", contador_maiusculas);
    printf("Minúsculas: %d.\n", contador_minusculas);
```

```
    printf("Pontuação: %d.\n", contador_pontuacao);
    printf("Dígitos: %d.\n", contador_digitos);
    printf("Caracteres de controle: %d.\n", contador_controle);

    return 0;
}
```

```
Digite um texto qualquer:
Moro na rua XV de Novembro, 2605
Maiúsculas: 4.
Minúsculas: 17.
Pontuação: 1.
Dígitos: 4.
Caracteres de controle: 1.
```

Esse programa usa algumas das funções da Tabela 17.2 para fazer a contagem de alguns tipos de caracteres. O resultado para o texto `Moro na rua XV de Novembro, 2605` pode ser visto diretamente. Um destaque é feito para o número de caracteres de controle, que incluiu o `\n` existente no fim do texto de entrada.

17.4.1 A remoção do `\n` incluído pelo `fgets`

Uma ação bastante comum quando a leitura de um valor textual é feito com a função `fgets` é a remoção do `\n` que ela deixa na cadeia de caracteres lida. Essa remoção é importante para ficar somente com o texto digitado. Em mais detalhes, essa remoção parte de dois princípios: primeiro, que a leitura com o `fgets` sempre terá um `\n` no seu final e, segundo, que tudo que estiver na variável a partir do `\0` é ignorado.

Dessa forma, a eliminação do `\n` é feita pela simples atribuição do `\0` sobre ele. Como consequência, a cadeia de caracteres contém um novo final.

```
texto[strlen(texto) - 1] = '\0';  // elimina o último caractere sobrepondo
                                  // a ele um novo terminador de cadeia
```

O programa seguinte tem o objetivo de apresentar essa remoção em detalhes, mostrando o conteúdo da variável antes e depois da eliminação do `\n`.

```
/*
Apresentação de todos os valores contidos em uma variável literal
Requer: um texto de até 24 caracteres digitados pelo usuário
Assegura: apresentação detalhada do texto posição a posição antes e
    depois da remoção do \n remanescente do fgets
*/
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int main(void) {
    // Leitura do texto
    char texto[25];
    printf("Digite um texto qualquer:\n");
    fgets(texto, sizeof texto, stdin);

    // Apresentação do vetor inteiro
    printf("\nAntes:\n");
    for (int i = 0; i < (int)sizeof texto; i++)
        printf("%3d", i);
    printf("\n");
    for (int i = 0; i < (int)sizeof texto; i++) {
        if (texto[i] == '\0')
            printf(" \\0");
        else if (texto[i] == '\n')
            printf(" \\n");
```

```
        else if (texto[i] == ' ')
            printf("   ");
        else if (isprint(texto[i]))
            printf("%3c", texto[i]);
        else
            printf(" ??");
    }
    printf("\n");

    // Remoção do \n final
    texto[strlen(texto) - 1] = '\0';  // remoção do \n

    // Reapresentação do vetor depois da alteração
    printf("\nDepois:\n");
    for (int i = 0; i < (int)sizeof texto; i++)
        printf("%3d", i);
    printf("\n");
    for (int i = 0; i < (int)sizeof texto; i++) {
        if (texto[i] == '\0')
            printf(" \\0");
        else if (texto[i] == '\n')
            printf(" \\n");
        else if (texto[i] == ' ')
            printf("   ");
        else if (isprint(texto[i]))
            printf("%3c", texto[i]);
        else
            printf(" ??");
    }
    printf("\n");

    return 0;
}
```

```
Digite um texto qualquer:
Teste

Antes:
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
  T  e  s  t  e \n \0  ~  ) ?? ?? ??  @ ?? ?? \0  | ??  T ?? ?? \0 ?? ??  v

Depois:
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
  T  e  s  t  e \0 \0  ~  ) ?? ?? ??  @ ?? ?? \0  | ??  T ?? ?? \0 ?? ??  v
```

O objetivo desse programa é apresentar todas as posições da variável, incluindo as inválidas depois do `\0` final. Na saída, os valores `\n` e `\0` são verificados para serem apresentados de forma coerente, enquanto quaisquer caracteres não imprimíveis são apresentados como `??`. Para visualização, o vetor foi definido com tamanho 25. Tudo que está depois do `\0` é, naturalmente, lixo.

Pode-se observar que depois da leitura, a posição 5 contém o `\n` e a 6 o `\0` final. Depois da eliminação do `\n`, há um `\0` na posição 5, fazendo com que o conteúdo válido seja somente as posições de 0 a 4.

17.5 Acentuações nas cadeias de caracteres

Como abordado na Seção 16.2.1, o uso de Unicode como tabela de caracteres e sua a codificação UTF-8 são bastante comuns nos sistemas computacionais atuais e seu suporte em C é muito limitado. Há, entretanto, uma boa notícia. Da forma em que foi desenvolvido o sistema de representação de caracteres com múltiplos bytes, as funções especificadas em `string.h` continuam funcionando a contento. Há, claro, exceções.

A função `strncpy` apenas copia todos os bytes até que o terminador `\0` seja encontrado. Assim, independentemente da codificação adotada, a cópia é fiel ao original. O mesmo raciocínio se aplica a

`strncat`. Na comparação por meio de `strncmp`, a comparação pela igualdade funciona perfeitamente, pois se o texto é igual, também é sua codificação UTF-8, e a comparação ocorre entre os bytes e não entre os caracteres representados em si.

A comparação por desigualdade com `strncmp`, como abordado na Seção 17.2.4, tem suas limitações, uma vez que, por exemplo, `Y` vem antes de `b` na ordenação das tabelas de caracteres. Com UTF-8, essas limitações são agravadas. Por exemplo, `"aí"` é codificado para a sequência de bytes 97, 195 e 173 (`a` usa um byte, enquanto `í` necessita de dois) e `"abc"` tem codificação 97, 98 e 99 (um byte para cada letra), de modo que na comparação, `b` é comparado com uma parte de `í` e `c` é emparelhado com o seu segundo byte. Nada disso é coerente.

A pior situação fica com `strlen`, que retorna sempre o número de bytes e não o de caracteres. Ela é prática para determinar o número de caracteres simples do início até o `\0`, mas não considera múltiplos bytes por caractere.

Quando o espaço total na variável comporta o conteúdo que está sendo atribuído ou concatenado, o comportamento descrito até agora prevalece. O problema ocorre quando não há espaço e somente parte de um caractere de múltiplos bytes é copiada. O resultado, então, é um caractere inválido.

O programa seguinte ilustra o uso de caracteres acentuados nas strings.

```c
/*
Ilustração do uso de acentuação em cadeias de caracteres e algumas de suas
    limitações
Assegura: apresentação de atribuições bem sucedidas, da incoerência no
    comprimento da string e falha na atribuição parcial de um caractere
    longo
*/
#include <stdio.h>
#include <string.h>


int main(void) {
    char texto[160];

    // Situação normal
    strncpy(texto, "Acentuação e Unicode são assim", sizeof texto - 1);
    printf("%s.\n", texto);
    strncat(texto, ": hvaða tungumál er þetta?", sizeof texto - strlen(texto) - 1);
    printf("%s\n", texto);

    // Comprimento é infiel ao número de caracteres
    strncpy(texto, "Ímã", sizeof texto - 1);
    printf("'%s' tem comprimento %zu.\n", texto, strlen(texto));

    // Problema com espaço insuficiente
    char texto_pequeno[5];  // cabem 4 bytes
    strncpy(texto_pequeno, texto, sizeof texto_pequeno - 1);
    printf("Atribuição parcial: '%s'.\n", texto_pequeno);

    return 0;
}
```

```
Acentuação e Unicode são assim.
Acentuação e Unicode são assim: hvaða tungumál er þetta?
'Ímã' tem comprimento 5.
Atribuição parcial: 'Ím� '.
```

Nesse programa, a atribuição e uso de textos em Unicode funcionam normalmente no geral. O comprimento, como é o número de bytes, pode não corresponder ao número de caracteres. Um problema mais grave ocorre quando um caractere é copiado de forma incompleta, como ocorre quando apenas o primeiro byte que forma o `ã` é transferido por não haver espaço na variável `texto_pequeno`.

A conclusão é que, havendo espaço suficiente para o armazenamento, a codificação de caracteres pode ser usada com as funções convencionais sem uma implicação negativa significativa. Isso ocorre

para a maioria dos casos e, em geral, não é uma preocupação muito grande. Ainda vale apontar que `wchar.h` contém funções específicas para trabalhar com caracteres de múltiplos bytes.

Dica

Declarar variáveis texto com espaço um pouco superdimensionado ajuda a evitar problemas, principalmente quando se espera uma variabilidade grande no comprimento dos textos. Esse é o caso de uma variável para guardar um nome.
Variáveis mais "apertadas" se justificam quando a variabilidade de comprimento é muito pequena. Para exemplificar, o armazenamento de um CPF pode usar comprimento 15, pois o comprimento é sempre fixo com 14 caracteres (dígitos, pontos e hífen). Neste caso, não há necessidade de folga.

Parte VI

Modularização e memória

18 Funções em C

As funções são parte essencial do desenvolvimento de qualquer código. Não somente evitam que trabalho extra tenha que ser feito mas também melhoram a qualidade geral do código como um todo. As estruturas e estratégias das funções serão abordadas ao longo deste capítulo.

18.1 O que é uma função

O conceito de função nas linguagens de programação é bastante similar ao das funções matemáticas, estas últimas mapeando um conjunto domínio em um conjunto chamado contradomínio. A função $y = \cos x$, por exemplo, é usualmente definida para mapear valores $x \in \mathbb{R}$ para $y \in \mathbb{R}$, sendo domínio e contradomínio iguais; a função $y = \lfloor x \rfloor$ (função piso), por sua vez, tem domínio real e contradomínio inteiro.

Basicamente, uma função é aplicada em um argumento[1] do domínio e tem como resultado um valor do contradomínio.

Um exemplo de programa que inclui uma função matemática é apresentado a seguir.

```
/*
Apresentação de valores de raiz quadrada
Assegura: apresentação do valor e sua raiz, de 0 a 1, de 0,1 em 0,1
*/
#include <stdio.h>
#include <math.h>

int main(void) {
    for (double valor = 0; valor <= 1.0; valor += 0.1)
        printf("A raiz quadrada de %g é %g.\n", valor, sqrt(valor));

    return 0;
}
```

```
A raiz quadrada de 0 é 0.
A raiz quadrada de 0.1 é 0.316228.
A raiz quadrada de 0.2 é 0.447214.
A raiz quadrada de 0.3 é 0.547723.
A raiz quadrada de 0.4 é 0.632456.
A raiz quadrada de 0.5 é 0.707107.
A raiz quadrada de 0.6 é 0.774597.
A raiz quadrada de 0.7 é 0.83666.
A raiz quadrada de 0.8 é 0.894427.
A raiz quadrada de 0.9 é 0.948683.
A raiz quadrada de 1 é 1.
```

Em C, a função `sqrt` retorna a raiz quadrada de um valor, sendo seu domínio e contradomínio `double`.

Como domínio e contradomínios não precisam ser iguais, segue exemplo em C que ilustra uma função nessa condição.

[1] As funções, tanto matemáticas quanto computacionais, podem ter mais que um argumento (funções de múltiplas variáveis).

```c
/*
Apresentação de uma cadeia de caracteres e seu tamanho em bytes
Assegura: apresentação de um texto e o número de bytes que ele ocupa
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    char *texto = "Modularização";

    printf("\"%s\" usa %zu bytes.\n", texto, strlen(texto));
    return 0;
}
```

```
"Modularização" usa 15 bytes.
```

A função `strlen` faz o mapeamento de cadeias de caracteres (seu domínio) para valores inteiros (contradomínio).

As funções em C podem mapear os mais diferentes tipos de dados para outros tantos tipos, como de `double` para `int`, `int` para `double` ou cadeias de caracteres para `double`, por exemplo.

18.2 Declaração e implementação das funções

Todo programa em C contém pelo menos uma função. Desde o primeiro código apresentado nesse livro (o programa que não faz nada, na Seção 2.2.1), fica patente que todo executável gerado a partir de um código em C precisa de `main`, e `main` é uma função. Como usado em todos os programas até o momento, a função principal tem domínio vazio (indicado por `void`) e seu contradomínio é o tipo `int`.

```c
int main(void) {
    ...
    return 0;
}
```

Toda função tem um nome. Neste caso, o nome é `main`, assim como seriam logaritmo (log) ou cosseno (cos) para citar algumas funções matemáticas. Entre os parênteses estão especificados seus parâmetros e, para `main`, não há efetivamente nenhum parâmetro. O contradomínio é indicado antes do nome da função: `int`.

Para citar um exemplo já conhecido, a função `sqrt` tem a seguinte declaração em `math.h`.

```c
double sqrt(double x);
```

Colocando-se isso mais explicitamente, `sqrt` aceita um `double` como parâmetro e, como resultado, devolve um valor também `double`.

18.2.1 Declaração e implementação de funções próprias

O programador tem como uma grande vantagem poder escrever suas próprias funções e, para isso, precisa fazer sua declaração.

Declaração de função

tipo_de_retorno nome_da_função (*lista_de_parâmetros*) ;

As funções precisam ter um *nome_da_função* (identificador válido), especificar seu *tipo_de_retorno* e seus parâmetros (*lista_de_parâmetros*). Declarações de função nesse formato são conhecidos como protótipos da função.

Os conceitos envolvidos na escrita de funções podem ser ilustrados por uma sequência de tentativas de escrever um programa que usa uma função. Esse programa tem o objetivo de ler um texto qualquer e apresentar o número de vogais presentes (ignorando acentuações).

Segue a versão do programa sem nenhuma função.

```c
/*
Determinação da quantidade de vogais em um texto digitado pelo usuário
Requer: um texto qualquer
Assegura: apresentação do número de vogais contidas no texto
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    printf("Digite um texto: ");
    char texto[160];
    fgets(texto, sizeof texto, stdin);
    texto[strlen(texto) - 1] = '\0';

    int numero_vogais = 0;
    for (int i = 0; i < (int)strlen(texto); i++)
        if (texto[i] == 'a' || texto[i] == 'e' || texto[i] == 'i' ||
             texto[i] == 'o' || texto[i] == 'u' || texto[i] == 'A' ||
             texto[i] == 'E' || texto[i] == 'I' || texto[i] == 'O' ||
             texto[i] == 'U')
            numero_vogais++;

    printf("\"%s\" possui %d %s.\n", texto, numero_vogais,
           numero_vogais > 1 ? "vogais" : "vogal");

    return 0;
}
```

```
Digite um texto: Comece pelo começo, disse a Rainha a Alice
"Comece pelo começo, disse a Rainha a Alice" possui 18 vogais.
```

A linguagem provê funções de verificação do tipo dos caracteres, como `isdigit` ou `isspace` (Tabela 17.2), entre outras. Porém, entre elas não se inclui uma que verifica as vogais.

Assim, o objetivo do programa é ter uma função para verificar se um dado caractere é ou não uma vogal. Essa função será nomeada `eh_vogal`. O código seguinte tenta usar essa função, que até o momento não existe.

```c
/*
Determinação da quantidade de vogais em um texto digitado pelo usuário
Requer: um texto qualquer
Assegura: apresentação do número de vogais contidas no texto
*/
#include <stdio.h>
#include <string.h>

int main(void) {
    printf("Digite um texto: ");
    char texto[160];
    fgets(texto, sizeof texto, stdin);
    texto[strlen(texto) - 1] = '\0';

    int numero_vogais = 0;
    for (int i = 0; i < (int)strlen(texto); i++)
        if (eh_vogal(texto[i]))
            numero_vogais++;

    printf("\"%s\" possui %d %s.\n", texto, numero_vogais,
```

```
        numero_vogais > 1 ? "vogais" : "vogal");

    return 0;
}
```

```
main.c: In function ´main´:
main.c:17:13: warning: implicit declaration of function ´eh_vogal´
[-Wimplicit-function-declaration]
   17 |         if (eh_vogal(texto[i]))
      |             ^~~~~~~~
/usr/bin/ld: /tmp/cc4fm92w.o: na função "main":
main.c:(.text+0x7d): referência não definida para "eh_vogal"
collect2: error: ld returned 1 exit status
```

Há dois problemas dignos de destaque no resultado da compilação. O primeiro se refere ao uso de `eh_vogal`, dando um alerta (*warning*) de declaração implícita, o que significa que a função não possui declaração e o compilador assumiu um formato padrão. O segundo ponto é o que diz "referência não definida para `eh_vogal`", que significa que a função não foi implementada e não existe nenhum código para ela.

O primeiro ponto a se resolver, portanto, é declarar a função antes de ela ser usada, o que é feito antes da função `main`. Essa nova versão do programa é a que segue.

```
/*
Determinação da quantidade de vogais em um texto digitado pelo usuário
Requer: um texto qualquer
Assegura: apresentação do número de vogais contidas no texto
*/
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

bool eh_vogal(char caractere);

int main(void) {
    printf("Digite um texto: ");
    char texto[160];
    fgets(texto, sizeof texto, stdin);
    texto[strlen(texto) - 1] = '\0';

    int numero_vogais = 0;
    for (int i = 0; i < (int)strlen(texto); i++)
        if (eh_vogal(texto[i]))
            numero_vogais++;

    printf("\"%s\" possui %d %s.\n", texto, numero_vogais,
        numero_vogais > 1 ? "vogais" : "vogal");

    return 0;
}
```

```
/usr/bin/ld: /tmp/ccVYlaQN.o: na função "main":
main.c:(.text+0x78): referência não definida para "eh_vogal"
collect2: error: ld returned 1 exit status
```

A declaração é feita por meio de seu protótipo, que indica o tipo, o nome e os parâmetros que a função possui. A prototipagem da função requereu também incluir `stdbool.h` para poder especificar o tipo de retorno `bool`.

```
bool eh_vogal(char caractere);  // protótipo da função
```

Vale observar que a compilação do programa foi bem sucedida e todo o código objeto para ele foi criado com êxito. O problema que persiste é que, ao criar o executável, notou-se a ausência da implementação.

Esse é o problema que a versão final do programa resolve.

```
/*
Determinação da quantidade de vogais em um texto digitado pelo usuário
Requer: um texto qualquer
Assegura: apresentação do número de vogais contidas no texto
*/
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

bool eh_vogal(char caractere);

int main(void) {
    printf("Digite um texto: ");
    char texto[160];
    fgets(texto, sizeof texto, stdin);
    texto[strlen(texto) - 1] = '\0';

    int numero_vogais = 0;
    for (int i = 0; i < (int)strlen(texto); i++)
        if (eh_vogal(texto[i]))
            numero_vogais++;

    printf("\"%s\" possui %d %s.\n", texto, numero_vogais,
           numero_vogais > 1 ? "vogais" : "vogal");

    return 0;
}

/*!
 * Verificação se um caractere é uma vogal (maiúscula ou minúscula).
 * @param caractere: caractere simples a ser verificado
 * @return true se for vogal, false caso contrário
 */
bool eh_vogal(char caractere) {
    bool se_eh_vogal =
            (caractere == 'a' || caractere == 'e' || caractere == 'i' ||
             caractere == 'o' || caractere == 'u' || caractere == 'A' ||
             caractere == 'E' || caractere == 'I' || caractere == 'O' ||
             caractere == 'U');

    return se_eh_vogal;
}
```

```
Digite um texto: Comece pelo começo, disse a Rainha a Alice
"Comece pelo começo, disse a Rainha a Alice" possui 18 vogais.
```

A função declarada é lógica, retornando um valor `bool`, o nome escolhido para ela é `eh_vogal` e ela tem um único parâmetro que é um `char` denominado `caractere`.

Para sua implementação, a função define uma variável booliana local `se_eh_vogal` que recebe `true` ou `false` conforme o resultado da comparação. O valor resultante da função é determinado pela instrução `return` que, nesse caso, volta o valor booliano. Sempre que `return` é executado, a função é encerrada e comandos escritos depois dele nunca serão executados.

O uso da variável local `se_eh_vogal` é meramente didático pois estabelece dois passos: o cálculo do resultado da função e o retorno desse resultado. Na prática, é mais interessante escrever essa função conforme se segue, sem o uso da variável local e já retornando o valor da expressão.

```
/*!
 * Verificação se um caractere é uma vogal (maiúscula ou minúscula).
 * @param caractere: (char) caractere a ser verificado
 * @return true se for vogal, false caso contrário
 */
bool eh_vogal(char caractere) {
    return (caractere == 'a' || caractere == 'e' || caractere == 'i' ||
            caractere == 'o' || caractere == 'u' || caractere == 'A' ||
```

```
            caractere == 'E' || caractere == 'I' || caractere == 'O' ||
            caractere == 'U');
}
```

Para concluir esta seção é preciso rever dois pontos importantes:

- Toda função tem que ser declarada antes de ser usada;
- A implementação da função (seu código) tem que ter sido escrito para que executável possa ser criado.

Assim, para o exemplo do contador de vogais, a declaração foi feita na forma de um protótipo, o qual indica para o compilador o tipo de retorno e os parâmetros e respectivos tipos. Com essas informações, as devidas verificações de consistência podem ser realizadas durante a compilação, como verificar se o tipo do parâmetro passado é compatível com o tipo indicado na declaração. A implementação, por sua vez, foi inserida no código fonte logo abaixo de `main`, fornecendo condições para que `eh_vogal` possuísse sua implementação e viabilizasse o que se deseja: o executável completo.

18.2.2 Onde declarar a função?

Há duas estratégias usuais para declarar funções em C. A primeira é usando protótipos para a declaração e ter a implementação feita posterioriormente. Outra forma é pela inserção direta da implementação, a qual tanto implementa quanto declara a função ao mesmo tempo.

A segunda forma é apresentada na sequência, como uma versão alternativa do mesmo programa de contagem de vogais.

```
/*
Determinação da quantidade de vogais em um texto digitado pelo usuário
Requer: um texto qualquer
Assegura: apresentação do número de vogais contidas no texto
*/
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

/*!
 * Verificação se um caractere é uma vogal (maiúscula ou minúscula).
 * @param caractere: (char) caractere a ser verificado
 * @return true se for vogal, false caso contrário
 */
bool eh_vogal(char caractere) {
    return (caractere == 'a' || caractere == 'e' || caractere == 'i' ||
            caractere == 'o' || caractere == 'u' || caractere == 'A' ||
            caractere == 'E' || caractere == 'I' || caractere == 'O' ||
            caractere == 'U');
}

int main(void) {
    printf("Digite um texto: ");
    char texto[160];
    fgets(texto, sizeof texto, stdin);
    texto[strlen(texto) - 1] = '\0';

    int numero_vogais = 0;
    for (int i = 0; i < (int)strlen(texto); i++)
        if (eh_vogal(texto[i]))
            numero_vogais++;

    printf("\"%s\" possui %d %s.\n", texto, numero_vogais,
           numero_vogais > 1 ? "vogais" : "vogal");

    return 0;
}
```

```
Digite um texto: Comece pelo começo, disse a Rainha a Alice
"Comece pelo começo, disse a Rainha a Alice" possui 18 vogais.
```

Não há uma regra para usar uma forma ou outra, sendo assim uma opção pessoal (exceto quando a empresa ou o cliente possuírem regras específicas).

Como regra geral, este livro adota o uso das declarações de função com protótipos e cuja implementação fica posterior à função `main` quando estiverem no mesmo arquivo.

Curiosidade

É interessante notar que, embora a implementação da função `eh_vogal` tenha sido feita no mesmo arquivo em que estava a função `main`, ela poderia estar em outro arquivo.
Quando programas se tornam maiores, é uma prática comum e muito positiva separar as implementações de funções em arquivos separados. Assim, se um programa deve lidar com o processamento de um texto, pode haver um arquivo para as funções que manipulam arquivos, outro para as que manipulam as cadeias de caracteres e assim por diante. Tudo se reflete em organização.
Esse assunto é abordado no **?@sec-organizacao-do-codigo-separado**.

18.2.3 Escopo da declaração das funções

Uma discussão prévia sobre declarações e escopo está apresentada na Seção 9.1, tratando da validade de cada variável em um programa. Com a introdução do assunto da modularização, o conceito de escopo se amplia. Qualquer variável usada nos diversos programas possuem sua validade definida internamente a `main`, ou seja, são locais à função principal e não existem fora dela.

Ao se criar uma nova função, ela é declarada fora da função `main` e não é, portanto, local. As declarações das funções são sempre globais. A consequência para esse tipo de declaração é que ela é válida da linha do protótipo (ou do cabeçalho de uma função implementada) até o fim do arquivo do código fonte.

O programa seguinte é apenas um exemplo para o qual comentários pertinentes sobre escopo serão apresentados. Nenhuma função é comentada, visto que são apenas exemplos simples (documentação de funções está na Seção 18.2.5).

```c
/*
Programa exemplo com funções simples
*/
#include <stdio.h>

double dobro(double valor);

double maximo(double valor1, double valor2);

int somatorio(int n);

double um_double(void);

int um_int(void);

int main(void) {
    printf("Quádruplo de 1.77 = %.2lf.\n", dobro(dobro(1.77)));
    printf("max(1.2, 7.8) = %.1lf.\n", maximo(1.2, 7.8));
    printf("Somatório de 1 até 100 = %d.\n", somatorio(100));
    printf("Um: %d e %.1f.\n", um_int(), um_double());

    return 0;
}

double dobro(double valor) {
```

```
26	    double valor_dobrado = 2 * valor;
27	
28	    return valor_dobrado;
29	}
30	
31	double maximo(double valor1, double valor2) {
32	    return (valor1 > valor2) ? valor1 : valor2;
33	}
34	
35	int somatorio(int n) {
36	    int soma = 0;
37	    for (int i = 1; i <= n; i++)
38	        soma += i;
39	
40	    return soma;
41	}
42	
43	double um_double(void) {
44	    return (double)um_int();
45	}
46	
47	int um_int(void) {
48	    return 1;
49	}
```

```
Quádruplo de 1.77 = 7.08.
max(1.2, 7.8) = 7.8.
Somatório de 1 até 100 = 5050.
Um: 1 e 1.0.
```

No programa exemplificado há cinco funções definidas pelo programador: `dobro`, `maximo`, `somatorio`, `um_int` e `um_double`. Cada uma delas é declarada por seu respectivo protótipo e, em consequência, possuem validade da linha em que houve a declaração até o final do arquivo do código fonte. Como exemplos, a função `dobro` é conhecida e pode ser usada da linha 6 até a 49, enquanto `somatorio` tem validade a partir da linha 10 e também encerrando na linha 49.

Assim, toda função pode ser usada a partir da linha de sua declaração. Quando se chega à função `main`, todas as funções já são conhecidas e seu uso é perfeitamente possível.

Um destaque relevante vai para as funções `um_double` e `um_int`, que apenas retornam o valor unitário, um do tipo `double`, outro do tipo `int`. No exemplo, a função `um_double` retorna uma chamada para `um_int`, a qual já foi declarada por seu protótipo.

Segue um exemplo para o caso em que as funções têm suas declarações sem prototipagem, ou seja, são declaradas e implementadas antes da função `main`. A regra de escopo aqui é a mesma e, a título de exemplo, `maximo` é declarado na linha 12 e pode ser usado desta linha até a 39.

```
1	/*
2	Programa exemplo com funções simples
3	*/
4	#include <stdio.h>
5	
6	double dobro(double valor) {
7	    double valor_dobrado = 2 * valor;
8	
9	    return valor_dobrado;
10	}
11	
12	double maximo(double valor1, double valor2) {
13	    return (valor1 > valor2) ? valor1 : valor2;
14	}
15	
16	int somatorio(int n) {
17	    int soma = 0;
18	    for (int i = 1; i <= n; i++)
19	        soma += i;
20	
```

```
21      return soma;
22  }
23
24  int um_int(void) {
25      return 1;
26  }
27
28  double um_double(void) {
29      return (double)um_int();
30  }
31
32  int main(void) {
33      printf("Quádruplo de 1.77 = %.2lf.\n", dobro(dobro(1.77)));
34      printf("max(1.2, 7.8) = %.1lf.\n", maximo(1.2, 7.8));
35      printf("Somatório de 1 até 100 = %d.\n", somatorio(100));
36      printf("Um: %d e %.1f.\n", um_int(), um_double());
37
38      return 0;
39  }
```

```
Quádruplo de 1.77 = 7.08.
max(1.2, 7.8) = 7.8.
Somatório de 1 até 100 = 5050.
Um: 1 e 1.0.
```

Uma ressalva importante é feita à função `um_double`, a qual necessariamente tem que ser declarada depois de `um_int`, pois essa inversão implicaria no problema de uma função ser desconhecida antes de ser usada. Nesses casos ou a correta ordenação ou a inclusão de um protótipo são soluções válidas.

> **Cuidado**
>
> A especificação da linguagem C não admite que funções sejam declaradas dentro de outras funções, o chamado aninhamento de declarações. Uma vantagem dessa estratégia é ter uma função que é reconhecida e usada apenas dentro de outra função.
> Muitos compiladores dão, porém, suporte a esse recurso e, assim, torna-se importante ressaltar que o código fonte pode não ser compatível com outros compiladores, outros sistemas ou mesmo com versões futuras.
> Aderir aos recursos padronizados oficiais da linguagem é sempre uma boa opção de conduta.

18.2.4 Funções com vários parâmetros

Não é incomum que uma função precise de mais que um único parâmetro. Por exemplo, pode-se ter uma função que calcule o peso (massa) ideal de uma pessoa conforme seu sexo biológico e sua altura. Essa função poderia ser escrita com a declaração seguinte.

```
/*!
 * Determina a massa ideal a partir do sexo biológico e da altura de um
 * indivíduo.
 * @param sexo: o sexo biológio ('F' para feminino, 'M' para masculino)
 * @param altura: a altura da pessoa em metros
 * @return a massa ideal em quilogramas
 */
double massa_ideal(char sexo, double altura);
```

Quando há mais que um parâmetro, eles são separados por vírgulas. Na sequência, são apresentadas declarações genéricas e suas interpretações.

```
double funcao(int a, double b);  // a é int, b é double; retorna double
```

```
double funcao(int a, int b);  // a e b são int; retorna double
```

```
bool funcao(bool a, bool b, bool c);  // a, b e c são bool; retorna bool
```

```
int funcao(char *a, char *b, int c, int d);  // a e b são strings; c e d são
                                             // int; retorna int
```

> **Cuidado**
>
> Cada um dos parâmetros devem ter seu tipo especificado. Um exemplo interessante é uma função declarada como na sequência.
>
> ```
> double funcao(double a, b, c);
> ```
>
> Nessa função, o parâmetro `a` é do tipo `double`. Porém `b` e `c` não possuem tipo e, em consequência, haverá um erro de compilação.

18.2.5 Documentação

Neste livro as funções são documentadas usando um padrão de formato aceito pelo Doxygen[2], que é uma ferramenta para gerar documentação para grandes projetos. No caso dos programas e funções apresentados como exemplos ao longo do texto, o termo "grande projeto", entretanto, não se aplica, nem tampouco é necessária uma ferramenta para gerar a documentação. Apesar disso, o formato escolhido para a documentação é simples e atende às necessidades.

A documentação de uma função envolve:

- A descrição do que ela faz;
- A apresentação contextualizada dos parâmetros;
- O que a função retorna como resultado.

Como exemplo, segue a declaração da função para o cálculo do fatorial de um valor inteiro.

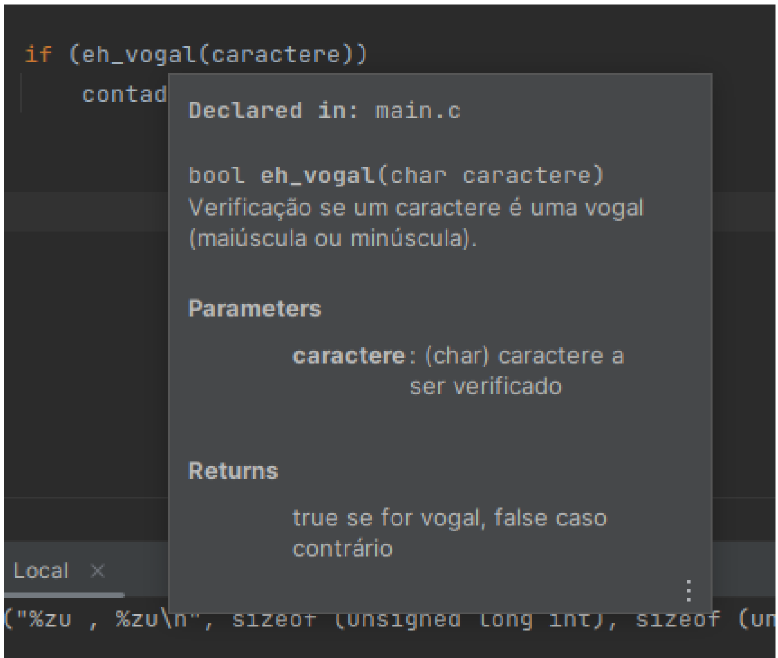
```
/*!
 * Retorna, para um dado n, o valor de n!
 * @param n: valor para o qual seu fatorial é calculado, n >= 0
 * @return n!
 */
unsigned long int fatorial(unsigned int n);
```

Nessa declaração, a descrição cobre o que a função faz: retorna o fatorial de um valor n indicado. na lista de parâmetros há um só, denominado `n`, além de indicar que a função atende corretamente apenas valores naturais. O retorno da função, por sua vez, é o que a função promete devolver, ou seja, $n!$. Os tipos envolvidos tanto no parâmetro quanto para o retorno estão indicados no cabeçalho da função.

Quando se escreve a documentação, é importante que esteja indicado o que função faz e não como ela o faz. Por exemplo, a descrição não deve conter algo como "por meio de multiplicações sucessivas", pois isso já se refere ao como. A descrição pode ser curta ou longa, dependendo da necessidade, sempre com foco de indicar como outro programador poderá usar essa função corretamente. Para os parâmetros formais, ou seja, os que ficam entre os parênteses, todos devem ser elencados dentro do contexto da função (indicados com `@param`). Assim, para o fatorial de um $n \in \mathbb{N}$, o único parâmetro é o n a ser

[2] Doxygen: https://www.doxygen.nl.

Figura 18.1: Auxílio ao programador em IDEs quando as funções são devidamente documentadas. IDE: CLion.



usado, sendo que `n >= 0` deixa claro para quais valores a função funciona corretamente[^ressalva-dominio-incorreto]. Para o retorno da função (`@return`), é indicado tão claramente quanto possível o que é resultado da função.

A documentação de uma uma função é sempre um elemento importante. Entretanto, documentar absolutamente todas as funções pode ter um efeito inverso ao de proporcionar clareza a um dado código. Funções extremamente simples, que sejam praticamente autoexplicativas, podem ficar sem uma documentação explícita. O mesmo ocorre para funções que são apenas auxiliares para outras funções e não são destinadas para que outros programadores as usem. Nessa última categoria, é interessante ver que `sscanf` é uma função importante e, portanto, deva ter sua documentação bem estabelecida (o que é verdade, pois possui sua página de manual), mas é possível supor que ela chame outras funções menores para cada conversão de texto para um determinado tipo; essas funções não requerem, necessariamente, uma documentação, já que não foram feitas para serem usadas abertamente.

Nos exemplos dados ao longo deste texto, é possível que funções simples ou auxiliares deixem de ter documentação explícita.

Os IDEs em geral reconhecem a documentação das funções e auxiliam o programador durante a codificação. Na Figura 18.1 há um exemplo de como a documentação é mostrada quando se posiciona o mouse sobre o nome da função `eh_vogal`. Outros auxílios, como verificações dos tipos dos parâmetros também são recursos comuns.

18.3 Criação de funções

Quando uma função é escrita em um programa, é importante que se tenha clareza do seu objetivo. A função deve, em consequência, ater-se somente a ele e não realizar outra tarefa qualquer. Por exemplo, se uma função é escrita para converter graus Celsius para Fahrenheit, é somente a conversão que deve ser feita; a função não deve ler nada nem escrever nada.

Segue um exemplo dessa função.

```
/*!
 * Retorna a temperatura em Fahenheit dado seu valor em graus Celsius
 * @param celsius: a temperatura Celsius
 * @return a temperatura em Fahrenheit
 */
double celsius_para_fahrenheit(double celsius) {
    return 1.8 * celsius + 32;
}
```

Todos os dados necessários devem ser passados como parâmetros e o resultado retornado pela função.

Não se está dizendo aqui que função não podem nem ler nem escrever. Se uma função tenha como objetivo ler um valor digitado pelo usuário, é isso que ela tem que fazer. O programa exemplo seguinte inclui uma função de interface para ler um valor inteiro digitado pelo usuário, verificando erros e garantindo que um valor válido seja entrado.

```
/*
Leitura da idade de uma pessoa e escrita desse valor na tela
Requer: digitação de uma idade
Assegura: apresentação da idade na tela
*/
#include <stdio.h>

int leia_inteiro(char *mensagem);

int main(void) {
    int idade = leia_inteiro("Digite sua idade: ");
    printf("Você tem %d anos de idade.\n", idade);

    return 0;
}

/*!
 * Retorna um valor inteiro válido digitado pelo usuário.
 * A leitura é repetida caso o valor digitado não corresponda a
 * um inteiro válido, sendo uma mensagem de aviso apresentada. Qualquer
 * texto depois do inteiro é descartado.
 * @param mensagem: mensagem solicitando a leitura
 * @return um valor inteiro lido
 */
int leia_inteiro(char *mensagem) {
    printf("%s", mensagem);
    int valor_lido, quantidade_itens_lidos;
    do {
        char entrada[160];
        fgets(entrada, sizeof entrada, stdin);
        quantidade_itens_lidos = sscanf(entrada, "%d", &valor_lido);
        if (quantidade_itens_lidos != 1)
            printf("> Valor inválido. Esperado um inteiro.\n"
                   "%s", mensagem);
    } while (quantidade_itens_lidos != 1);

    return valor_lido;
}
```

```
Digite sua idade: abc
> Valor inválido. Esperado um inteiro.
Digite sua idade: *
> Valor inválido. Esperado um inteiro.
Digite sua idade: #10
> Valor inválido. Esperado um inteiro.
Digite sua idade: 25
Você tem 25 anos de idade.
```

Há que se notar que a função deve ler e retornar um valor inteiro e digitações erradas são detectadas e contornadas. Para ser uma função genérica, ela lê inteiros e não idades (tanto que valores negativos

seriam aceitos nesse caso). Essa função pode ser usada em qualquer outro programa para a leitura de um inteiro e ela não atende exclusivamente o problema do programa em questão.

Caso fosse interessante garantir que o valor fosse maior que zero, outra função poderia ser escrita para ler valores apenas em $\mathbb{N}$. Segue um exemplo de implementação de uma função que atende a esses requisitos, a qual aproveita a já escrita `leia_inteiro`, apenas repassando a mensagem para o usuário. No programa anterior, bastaria `main` executar `leia_natural` no lugar de `leia_inteiro`.

```
/*!
 * Retorna um valor inteiro natural digitado pelo usuário.
 * A leitura é repetida para valores inválidos e o restante da linha
 * depois do inteiro é ignorado
 * @param mensagem: mensagem solicitando a leitura
 * @return valor inteiro lido, maior ou igual a zero
 */
int leia_natural(char *mensagem) {
    int valor_lido;
    do {
        valor_lido = leia_inteiro(mensagem);
        if (valor_lido < 0)
            printf("Valor inválido. Esperado maior ou igual a zero.\n");
    } while (valor_lido < 0);
    return valor_lido;
}
```

18.4 Escolha dos tipos de retorno e dos parâmetros

Os tipos dos parâmetros devem satisfazer as necessidades das funções. Por exemplo, na conversão de graus Celsius para Fahrenheit é esperado que a temperatura em Celsius seja `double`, assim como o valor retornado em Fahrenheit.

```
double celsius_para_fahrenheit(double celsius);
```

Para uma função do cálculo do fatorial de um número, entretanto, as escolhas podem não ser tão elementares. Uma primeira tentativa seria, por exemplo, ter a função com o seguinte protótipo.

```
int fatorial(int n);
```

Um primeiro ponto é que, nos sistemas atuais, um `int` ocupa geralmente quatro bytes e permite valores de -2.147.483.648 a 2.147.483.647. Metade dos valores representados são negativos e o resultado do fatorial nunca será negativo. Assim, pode-se optar pela prototipagem seguinte.

```
unsigned int fatorial(int n);
```

Nessa nova versão, ainda considerando os tamanhos típicos para inteiros, os valores de retorno podem ser de zero a 4.294.967.295. Esse intervalo comportaria valores até 12! (479.001.600), pois 13! (6.227.020.800) já extrapolaria.

A versão seguinte expande o tipo de retorno para `unsigned long int`, o qual usa oito bytes na versão do `gcc` usada nos programas. Desse modo, a gama de possíveis resultados vai para 18.446.744.073.709.551.615, o que comportaria até 20![3].

```
unsigned long int fatorial(int n);
```

[3]Embora 20! não pareça muito, vale lembrar seu valor é 2.432.902.008.176.640.000, ou seja, aproximadamente $2{,}4 \times 10^{18}$.

O mesmo raciocínio se aplicaria ao parâmetro `n`. Para $n!$, sempre se tem $n \geq 0$, e, assim, a função poderia ter como protótipo final o formato que segue.

```
unsigned long int fatorial(unsigned int n);
```

Sua implementação seria como se segue.

```
/*!
 * Retorna, para um dado n, o valor de n!
 * @param n: valor para o qual seu fatorial é calculado, n >= 0
 * @return n!
 */
unsigned long int fatorial(unsigned int n) {
    unsigned long int multiplicador = 1;
    for (unsigned int i = 2; i <= n; i++)
        multiplicador *= i;

    return multiplicador;
}
```

A vantagem na escolha dos tipos é que, até certo limite, o compilador consegue verificar consistência nos valores passados para a função. Por exemplo, se houver uma variável `i` do tipo `int` (que possui valores negativos), o compilador pode avisar (não é erro) que pode haver algum problema. A mensagem de compilação seguinte mostra que pode haver um problema de conversão caso os valores sejam negativos e o resultado, portanto, pode não ser confiável.

```
main.c:18:31: warning: conversion to ´unsigned int´ from ´int´ may change
the sign of the result [-Wsign-conversion]
   18 |     printf("%lu.\n", fatorial(i));
      |                               ^
```

A boa escolha dos tipos sempre proporciona uma camada adicional de controle na escrita de programas.

19 Regras de escopo com a modularização

Este capítulo estende os conceitos de escopo de validade das declarações na linguagem C. A Seção 9.1 abordou as declarações de variáveis internas (locais) a `main`, enquanto na Seção 18.2.3. Esses assuntos são revisitados e integrados.

Além de variáveis e funções, há outros elementos em C que podem ser declarados. Esses não serão cobertos diretamente neste texto e, em grande parte, a discussão exposta aqui também se aplica a eles.

19.1 Local × global

Um código fonte escrito em C contém declarações, sejam de funções ou de variáveis. Esse código fonte está contido em um arquivo texto, usualmente com extensão `.c`.

Qualquer função declarada no arquivo tem escopo global, o que quer dizer que sua validade vai desde a linha em que ocorre a declaração até a última linha do arquivo. Em outras palavras, essa função é conhecida e pode ser usada dentro de seu escopo de declaração. Essa regra aplica-se tanto à declaração simples, na forma de protótipo de função, quanto às implementações sem o protótipo (Seção 18.2.3).

Outra forma de olhar para essa questão é considerar global qualquer declaração feita fora de uma função.

Como existe o conceito de "fora de uma função", também há o de "dentro de uma função". Assim, variáveis declaradas no corpo da implementação de uma função estão dentro da função e são chamadas de declarações locais. O termo se aplica também aos parâmetros formais da função.

19.1.1 Validade das declarações globais e locais

Para exemplificar tanto declarações globais quanto locais quanto suas validades, segue um programa para simplificação de números racionais, o qual emprega uma função para o cálculo do máximo divisor comum (MDC) entre dois números inteiros e outra para o cálculo do valor absoluto (módulo, $|n|$) de um inteiro. O objetivo do programa é simplificar um número racional, lembrando que $q \in \mathbb{Q}$ é um valor expresso na forma a/b, sendo $a \in \mathbb{Z}$ com $b \in \mathbb{Z}^*$.

A lógica de modificação do número racional é apresentada no Algoritmo 19.1.

Algoritmo 19.1: Leitura e apresentação de números racionais.

Descrição: Leitura de um valor racional na forma $\frac{a}{b}$ e sua apresentação, padronizando para mostrar o sinal (se negativo) e sua forma fracionária simplificada
Requer: número racional $\frac{a}{b}$
Assegura: apresentação do racional em forma padronizada

Obtenha o racional $r = \frac{a}{b}$
Determine o valor do sinal de r ▷ *1 ou −1*
Calcule o MDC entre a e b, guardando em *fator*
Calcule a' e b' com, respectivamente, $|a|$ e $|b|$
Modifique o valor de a' para ter o sinal de r e o valor $\frac{a'}{fator}$ ▷ *simplifica o numerador*
Modifique o valor de b' para $\frac{b'}{fator}$ ▷ *simplifica o denominador*
Apresente $r' = \frac{a'}{b'}$ ▷ *o sinal já está presente em a'*

A codificação em C é apresentada na sequência.

```c
/*
 * Leitura e escrita de um número racional na forma de fração
 * Requer: a digitação de um valor a/b, a,b inteiros, b != 0
 * Assegura: apresentação do mesmo valor em forma simplificada e padronizada
 */
#include <stdio.h>

/*!
 * Retorna o MDC de dois inteiros quaisquer (máximo divisor comum).
 * @param n1: primeiro valor
 * @param n2: segundo valor
 * @return MDC(n1, n2)
 */
unsigned int mdc(int n1, int n2);

/*!
 * Retorna o valor absoluto de um inteiro
 * @param n: valor inteiro
 * @return o valor absoluto do número, |n|
 */
int valor_absoluto(int n);

/*
 * Main
 */
int main() {
    // Leitura
    printf("Digite um número racional a/b (a e b inteiros e b não nulo): ");
    char entrada[160];
    fgets(entrada, sizeof entrada, stdin);
    int numerador, denominador;
    sscanf(entrada, "%d/%d", &numerador, &denominador);

    // Simplificação da fração e colocação do sinal no numerador
    if (numerador == 0)
        denominador = 1;  // o valor zero é padronizado para 0/1
    else {
        int fator_divisao = mdc(numerador, denominador);
        int sinal_da_fracao = ((double)numerador / denominador >= 0) ? 1 : -1;
        numerador = valor_absoluto(numerador);
        denominador = valor_absoluto(denominador);
        numerador /= sinal_da_fracao * fator_divisao;
        denominador /= fator_divisao;
    }

    // Resultado
    printf("O racional digitado foi: %d/%d.\n", numerador, denominador);

    return 0;
```

```
50  }
51
52  // Máximo divisor comum: MDC(n1, n2)
53  unsigned int mdc(int n1, int n2) {
54      // Converte n1 e n2 para valores positivos
55      n1 = valor_absoluto(n1);
56      n2 = valor_absoluto(n2);
57
58      // Resolução pelo método de Euclides
59      int resto;
60      do {
61          resto = n1 % n2;
62          n1 = n2;
63          n2 = resto;
64      } while (resto != 0);
65
66      return n1;  // Contém o MDC no final
67  }
68
69  // Valor absoluto de um inteiro
70  int valor_absoluto(int n) {
71      return (n >= 0) ? n : -n;
72  }
```

```
Digite um número racional a/b (a e b inteiros e b não nulo): -3553/-627
O racional digitado foi: 17/3.
```

Para esse programa, há declarações tanto de variáveis quanto de funções. A Tabela 19.1 apresenta as declarações de interesse no programa e destaca o escopo e validade (linhas do código) de cada uma.

Tabela 19.1: Declarações relevantes feitas no programa de apresentação de números racionais, seu tipo, escopo e linhas em que são válidas.

Declaração	Tipo	Escopo	Início	Fim
`mdc`	função	global	14	68
`valor_absoluto`	função	global	21	68
`entrada`	variável	local (`main`)	29	46
`numerador`, `denominador`	variável	local (`main`)	31	46
`fator_divisao`	variável	local (`main`)	35	46
`sinal_da_fracao`	variável	local (`main`)	36	46
`n1`, `n2`	parâmetro	local (`mdc`)	49	63
`resto`	variável	local (`mdc`)	55	63
`n`	parâmetro	local (`valor_absoluto`)	66	68

Curiosidade

É interessante apontar que `n1` e `n2` na linha 14, assim como `n` na linha 21 do programa apresentado não possuem validade, pois o protótipo de uma função é apenas sua declaração e, como tal, seus parâmetros não são realmente criados, mas apenas informados ao compilador.
Na prática, a linha 14 do código do programa poderia ser escrita como segue, mas com o prejuízo de reduzir as informações de documentação do programa ao não informar semanticamente a que cada parâmetro se refere.

```
unsigned int mdc(int, int);  // o nome dos parâmetros é irrelevante, mas sua
                             // omissão prejudica a documentação e legibilidade
```

19.2 Reuso de identificadores

Nos programa em C é possível usar um mesmo identificador desde que eles suas declarações atendam escopos diferentes. Dessa forma, um identificador `p` pode ser parâmetro para diversas funções diferentes. Da mesma forma, uma variável local a uma função não conflita com qualquer outra declaração. O compilador reclamará, assim, apenas de duas declarações globais com mesmo identificador ou então o uso de um mesmo nome em duplicidade no mesmo contexto local.

A seguir é apresentado um código fonte genérico, não realiza qualquer processamento útil, O objetivo é indicar como identificadores iguais são tratados.

```
/*
 * Programa exemplo de declarações
 */
#include <stdio.h>

int f1(int a, int b);

double f2(double a);

int f3(double f1, double f2);

int main(void) {
    int a, b, c;
    double d = 1.25;

    a = (int)f2(d);
    b = f3(d, 0.5 * a);
    c = f1(a, b);

    printf("%d %d %d %g\n", a, b, c, d);

    return 0;
}

int f1(int a, int b) {
    int n = f3(a, b);
    return a + b + n;
}

double f2(double a) {
    int n = 1;
    double b = f1(a, n);
    return b;
}

int f3(double f1, double f2) {
    int main = (int)f1 + (int)f2;
    return main;
}
```

```
4 3 14 1.25
```

Os pontos que requerem atenção neste programa são os seguintes:

- As funções `f1`, `f2` e `f3` têm validade em praticamente todo o programa;
- Ambas as funções `f1` e `f2` possuem parâmetro com identificador `a`, mas eles são independentes pois estão em escopos diferentes;
- A função principal `main` também possui variáveis locais `a` e `b`, também disjuntas dos parâmetros e outras declarações locais;
- Tanto `f1` quanto `f2` possuem variáveis locais chamadas `n`, sendo elas completamente separadas dado seu escopo local diferente;
- A função `f1` chama `f3`, que é conhecida dado o escopo global, sendo que o mesmo ocorre com a chamada de `f1` em `f2`;

- `f3` possui parâmetros `f1` e `f2`, cujos nomes se sobrepõem às funções globais `f1` e `f2`, significando que, dentro de `f3`, essas funções não estão acessíveis pois são obscurecidas pelas declarações locais;
- `f3` possui uma variável local `main`, que não conflita com a função global com mesmo nome.

Dessa forma, seguem algumas orientações gerais de escopo:

- Declarações globais valem em todo o código fonte a partir de sua declaração;
- Parâmetros e declarações em funções são locais, têm apenas validade no escopo da função e são independentes de qualquer outra declaração com mesmo nome em escopo maior;
- Declarações locais podem se sobrepor e ocultar declarações globais;
- Blocos de comandos podem ter declarações cujo escopo é o próprio bloco apenas (Seção 9.1).

19.3 Variáveis globais

Assim como funções, também variáveis podem ser globais. Uma variável declarada fora do escopo de qualquer função é uma variável global e, como tal, tem sua validade definida e conhecida desde a linha em que é declarada até o final do arquivo.

Variáveis declaradas como globais possuem duas diferenças importantes em relação às locais, sejam variáveis ou parâmetros de funções:

- Local e momento de criação;
- Iniciação automática.

A primeira diferença, portanto, é que as variáveis globais são criadas juntamente com a execução do programa e possuem um espaço de memória específico para elas. As variáveis locais e os parâmetros são criados apenas no momento em que a função é chamada e, dependendo de quando isso ocorre, podem ser criadas em diferentes locais da memória dependendo da chamada.

Variáveis globais são consideradas estáticas enquanto as locais são criadas dinamicamente em função do momento em que as funções são chamadas

O segundo ponto é que as variáveis locais nunca possuem lixo, ou seja, são sempre iniciadas com valores nulos. Assim, se uma variável global `int i` é criada sem atribuição, seu valor será necessariamente zero. Caso exista um `double d` global, o valor de `d` será 0,0, exceto se houver outro valor inicial. De forma similar, se uma cadeia de caracteres for criada em escopo global com `char s[100]`, ela terá comprimento zero, pois todas suas posições terão `\0`.

19.3.1 Declaração de variáveis globais

Para que uma variável seja global, basta que sua declaração seja feita fora de uma função. Segue um exemplo simples em que foi criado um contador global para monitorar o número de vezes que uma

```
/*
 * Programa exemplo com variável global
 * O código cria duas funções, volta_igual e volta_negativo, mantendo
 *      controle sobre o número de vezes que elas são chamadas
 * Assegura: apresentações diversas do uso da função e do número de vezes
 *      em que foram chamadas
 */
#include <stdio.h>

//! Contador para uso global
int numero_chamadas;
```

```
/*!
 * Retorna igual ao que foi passado
 * @param n
 * @return n
 */
int volta_igual(int n);

/*!
 * Retorna o oposto do que foi passado
 * @param n
 * @return -n
 */
int volta_negativo(int n);

/*
 * Main
 */
int main(void) {
    printf("%d = %d.\n", 10, volta_igual(10));
    printf("%d = -1 * %d.\n\n", -10, volta_negativo(-10));
    for (int n = -5; n <= 5; n++)
        printf("%d = %d = -1 * %d.\n", n, volta_igual(n), volta_negativo(n));

    printf("\nAs funções volta_igual e volta_negativo foram chamadas %d "
           "vezes no total.\n", numero_chamadas);
    return 0;
}

// Retorna igual
int volta_igual(int n) {
    numero_chamadas++;  // conta a chamada à função
    return n;
}

// Retorna negativo
int volta_negativo(int n) {
    numero_chamadas++;  // conta a chamada à função
    return -n;
}
```

```
10 = 10.
-10 = -1 * 10.

-5 = -5 = -1 * 5.
-4 = -4 = -1 * 4.
-3 = -3 = -1 * 3.
-2 = -2 = -1 * 2.
-1 = -1 = -1 * 1.
0 = 0 = -1 * 0.
1 = 1 = -1 * -1.
2 = 2 = -1 * -2.
3 = 3 = -1 * -3.
4 = 4 = -1 * -4.
5 = 5 = -1 * -5.

As funções volta_igual e volta_negativo foram chamadas 24 vezes no total.
```

A variável `numero_chamadas` é global e seu escopo de validade se inicia na linha de declaração, de forma que ela pode ser usada em todas as funções seguintes, incluindo `main`. Como ela é global e não há iniciação explícita, seu valor inicial é zero. Cada vez que as funções `volta_igual` e `volta_negativo` são chamadas, essa variável é incrementada.

19.3.2 Quando usar variáveis globais?

A resposta rápida e curta para a pergunta do título da seção é simples: nunca. Claro que "nunca" é um exagero, pois há exceções. O ponto é sempre que qualquer variável global deve ser evitada, pois pode induzir a erros no código extremamente difíceis de serem localizados.

O exemplo do contador de chamadas pode, talvez, ser caracterizado como uma exceção. O programa é pequeno e o uso da variável global para o contador oculta a contagem separando-a do uso da função. Se outro programador fizer modificações no programa, ele poderia até ignorar a contagem e usar as duas funções definidas sem problemas.

O problema do uso de variáveis locais, entretanto, é exatamente alguém fazer uma modificação no programa e, por um descuido simples, interferir inadvertidamente no valor de uma variável que ele nem sabia que existia.

Para exemplificar, suponha que seja solicitado a outro programador uma pequena modificação na função `main`: a inclusão de uma série de exemplos de chamadas à função `volta_negativo` antes dos exemplos já existentes. O programa seguinte mostra a solução feita rapidamente pelo novo programador.

```
/*
 * Programa exemplo com variável global
 * O código cria duas funções, volta_igual e volta_negativo, mantendo
 *      controle sobre o número de vezes que elas são chamadas
 * Assegura: apresentações diversas do uso da função e do número de vezes
 *      em que foram chamadas
 */
#include <stdio.h>

//! Contador para uso global
int numero_chamadas;

/*!
 * Retorna igual ao que foi passado
 * @param n
 * @return n
 */
int volta_igual(int n);

/*!
 * Retorna o oposto do que foi passado
 * @param n
 * @return -n
 */
int volta_negativo(int n);

/*
 * Main
 */
int main(void) {
    // Exemplos novos para volta_negativo
    int numero_chamadas;
    for (numero_chamadas = 10; numero_chamadas >= 0; numero_chamadas--)
        printf("> volta_negativo(%d) = %d.\n", numero_chamadas,
               volta_negativo(numero_chamadas));

    // Código com os exemplos orginais
    printf("%d = %d.\n", 10, volta_igual(10));
    printf("%d = -1 * %d.\n\n", -10, volta_negativo(-10));
    for (int n = -5; n <= 5; n++)
        printf("%d = %d = -1 * %d.\n", n, volta_igual(n), volta_negativo(n));

    printf("\nAs funções volta_igual e volta_negativo foram chamadas %d "
           "vezes no total.\n", numero_chamadas);
    return 0;
}

// Retorna igual
int volta_igual(int n) {
    numero_chamadas++;  // conta a chamada à função
    return n;
}

// Retorna negativo
int volta_negativo(int n) {
    numero_chamadas++;  // conta a chamada à função
```

```
    return -n;
}
```

```
> volta_negativo(10) = -10.
> volta_negativo(9) = -9.
> volta_negativo(8) = -8.
> volta_negativo(7) = -7.
> volta_negativo(6) = -6.
> volta_negativo(5) = -5.
> volta_negativo(4) = -4.
> volta_negativo(3) = -3.
> volta_negativo(2) = -2.
> volta_negativo(1) = -1.
> volta_negativo(0) = 0.
10 = 10.
-10 = -1 * 10.

-5 = -5 = -1 * 5.
-4 = -4 = -1 * 4.
-3 = -3 = -1 * 3.
-2 = -2 = -1 * 2.
-1 = -1 = -1 * 1.
0 = 0 = -1 * 0.
1 = 1 = -1 * -1.
2 = 2 = -1 * -2.
3 = 3 = -1 * -3.
4 = 4 = -1 * -4.
5 = 5 = -1 * -5.

As funções volta_igual e volta_negativo foram chamadas -1 vezes no total.
```

O programa foi compilado com sucesso, sem erros e sem avisos. Porém o resultado agora está incorreto. Uma vez criada a variável local `numero_chamadas`, ela se sobrepõe à global. No último `printf`, o valor mostrado para a contagem é o da nova variável local e não mais o do contador global, produzindo um resultado inconsistente, provavelmente de fácil detecção, já que o novo resultado é incongruente.

Um erro mais sutil poderia ser introduzido, gerando uma situação em que o programa afirma que 11 = -10.

```c
/*
 * Programa exemplo com variável global
 * O código cria duas funções, volta_igual e volta_negativo, mantendo
 *      controle sobre o número de vezes que elas são chamadas
 * Assegura: apresentações diversas do uso da função e do número de vezes
 *      em que foram chamadas
 */
#include <stdio.h>

//! Contador para uso global
int numero_chamadas;

/*!
 * Retorna igual ao que foi passado
 * @param n
 * @return n
 */
int volta_igual(int n);

/*!
 * Retorna o oposto do que foi passado
 * @param n
 * @return -n
 */
int volta_negativo(int n);

/*
 * Main
 */
```

```
int main(void) {
    // Exemplo adicional
    numero_chamadas = 10;
    printf("> volta_negativo(%d) = %d.\n", numero_chamadas,
           volta_negativo(numero_chamadas));

    // Código com os exemplos originais
    printf("%d = %d.\n", 10, volta_igual(10));
    printf("%d = -1 * %d.\n\n", -10, volta_negativo(-10));
    for (int n = -5; n <= 5; n++)
        printf("%d = %d = -1 * %d.\n", n, volta_igual(n), volta_negativo(n));

    printf("\nAs funções volta_igual e volta_negativo foram chamadas %d "
           "vezes no total.\n", numero_chamadas);
    return 0;
}

// Retorna igual
int volta_igual(int n) {
    numero_chamadas++;  // conta a chamada à função
    return n;
}

// Retorna negativo
int volta_negativo(int n) {
    numero_chamadas++;  // conta a chamada à função
    return -n;
}
```

```
> volta_negativo(11) = -10.
10 = 10.
-10 = -1 * 10.

-5 = -5 = -1 * 5.
-4 = -4 = -1 * 4.
-3 = -3 = -1 * 3.
-2 = -2 = -1 * 2.
-1 = -1 = -1 * 1.
0 = 0 = -1 * 0.
1 = 1 = -1 * -1.
2 = 2 = -1 * -2.
3 = 3 = -1 * -3.
4 = 4 = -1 * -4.
5 = 5 = -1 * -5.

As funções volta_igual e volta_negativo foram chamadas 35 vezes no total.
```

Em programas mais longos, mais complexos e com muitas funções, diagnosticar problemas com variáveis globais pode ser uma tarefa árdua e desmotivante.

20 Endereçamento de memória e ponteiros nos programas

Este capítulo aborda alguns detalhes sobre como os diversos elementos se relacionam à memória do dispositivo onde um programa é executado. Esse tema pode parecer desconexo do conteúdo de todos os capítulos anteriores, mas os conceitos descritos aqui são muito importantes para capítulos seguintes. Para capítulos anteriores este material complementa informações já apresentadas de forma mais superficial. Nos capítulos seguintes este conteúdo será relevante, pois são tratados mecanismos para modificar, dentro de funções, variáveis declaradas em outros escopos (Capítulo 21, Seção 22.2) e também meios para requerer dinamicamente espaços para armazenamento de dados (**?@sec-alocacao-dinamica-de-memoria**).

Nesta parte do texto apenas os conceitos de armazenamento e uso da memória são abordados. As aplicações desses recursos para resolver problemas práticos da linguagem são abordados em outros lugares.

20.1 Endereçamento de memória

Quando uma variável é declarada, um espaço na memória é reservado para guardar seu valor. Por exemplo, ao se criar uma variável `i` do tipo `int`, alguns bytes da memória precisam ser reservados para guardar o valor da variável.

```
int i;  // criação de uma variável inteira
```

Ao se fazer uma atribuição, como `i = 10`, os bytes da variável `i` são modificados para representar o valor inteiro 10. Se uma chamada `printf("%d", i)` é feita, os bytes da memória reservados para `i` são consultados e convertidos para um texto (valor decimal, `%d`) e apresentado na tela.

```
i = 10;  // os bytes de i são modificados para representar o valor inteiro 10
printf("%d", i);  // os bytes de i são consultados e convertidos para "10"
```

Até este momento, os bytes reservados para a variável `i` foram irrelevantes. O compilador, apenas tendo o nome da variável (identificador `i`), sabe onde e quantos são os bytes usados e como os valores devem ser representados. Para usar a memória para os dados, basta usar seu identificador e todo o resto é gerenciado automaticamente. E isso é ótimo para o programador, tanto que essa necessidade pelos detalhes nunca apareceu.

Para ilustrar esses detalhes ocultos, segue um programa que apresenta mais informações sobre as variáveis do programa.

```
/*
 * Apresentação simples de endereços de memória
 * Assegura: apresentação do valor de variáveis e suas localizações na
 *  memória de execução do programa
 */
#include <stdio.h>
```

```
int main(void) {
    int i = 100;
    printf("i = %d e está no endereço %p e tem %zu bytes.\n", i, (void *)&i,
           sizeof i);

    double d = -17.2;
    printf("d = %g e está no endereço %p e tem %zu bytes.\n", d, (void *)&d,
           sizeof d);

    return 0;
}
```

```
i = 100 e está no endereço 0x7ffe51267d3c e tem 4 bytes.
d = -17.2 e está no endereço 0x7ffe51267d30 e tem 8 bytes.
```

Não há novidades na atribuição de valores tanto à variável `i` quanto `d`, nem na apresentação de seus valores com o `printf`. O que este programa introduz é o operador `&`, o qual significa "endereço de". Assim, `&i` é o endereço de memória da variável `i`, da mesma forma que `&d` corresponde ao endereço de `d`. O modificador de tipo (*cast*) `(void *)` serve apenas para indicar que o endereço é genérico e desprovido de tipo. Ao longo do texto esse assunto voltará a ser tratado. O operador `sizeof` também já foi utilizado e indica quantos bytes cada variável usa.

Quando um programa é colocado em execução, o sistema operacional cria um processo e compartilha com o programa o uso do processador e também uma porção da memória principal. A memória do programa vista por ele como um bloco contínuo de bytes, cada com seu endereço. É usual que endereços de memória sejam apresentados em valores hexadecimais (formato `%p` do `printf`).

Por exemplo, $7FFE51267D3C_{16}$ (endereço de `i` no programa) corresponde ao valor decimal 140.730.259.897.660, mas esse valor, por si só, não é relevante. Dado que a variável está no endereço $7FFE51267D3C_{16}$ e possui quatro bytes, os endereços $7FFE51267D3C_{16}$, $7FFE51267D3D_{16}$, $7FFE51267D3E_{16}$ e $7FFE51267D3F_{16}$ são usados pela variável. Um raciocínio similar se aplica aos oito bytes da variável `d`.

Para os objetivos desta seção, é apenas relevante saber, que cada variável está em algum lugar e que o compilador sabe seu endereço. Desse modo, atribuições triviais como `d = -17.2` podem ser feitas, pois o compilador sabe o tipo (`double`), a quantidade de bytes que serão usados (`sizeof d`) e quais são esses bytes (os oito bytes começando em $7FFE51267D30_{16}$).

20.2 Armazenamento de endereços

Endereços de memória podem ser guardados em variáveis, as quais recebem genericamente o nome de ponteiros. Quando um ponteiro guarda um endereço, diz-se que ele guarda uma referência àquele endereço e, portanto, ao seu conteúdo.

```
/*
 * Armazenamento de endereços de memória
 * Assegura: apresentação do endereço de uma variável
 */
#include <stdio.h>

int main(void) {
    double d = 1.125;
    double *endereco_de_d = &d;

    printf("A variável d usa %zu bytes começando em %p.\n", sizeof d,
           (void *)endereco_de_d);

    return 0;
}
```

```
A variável d usa 8 bytes começando em 0x7ffc3e6187b0.
```

Este programa cria uma variável chamada `endereco_de_d`, à qual é atribuído o valor `&d` (que é o endereço de `d`). O tipo de uma variável que guarda endereços deve ser sempre um ponteiro e sua declaração usa o `*` para indicar isso.

```
double *endereco_de_d;  // variável que guarda um endereço
```

Entrando em mais detalhes, a variável é declarada com o tipo `double *` e isso significa que a variável guarda o endereço de algo que ela sabe que é um `double`. Na prática, uma declaração de ponteiro como a usada significa que o valor armazenado será o endereço primeiro dos oito bytes que estão guardando um valor do tipo `double`.

Os ponteiros são criados com tipos associados à referência que vão armazenar e, assim, o compilador têm o controle do que é apontado. Seguem alguns exemplos adicionais de declarações.

```
int *pi;  // endereço de um inteiro
char *pc;  // endereço de um char
unsigned long int *puli;  // endereço de um unsigned long int
```

20.3 Ponteiros nulos

Não custa lembrar que uma variável do tipo ponteiro é como qualquer outra variável e, para ser usada, precisa ter um valor válido atribuído a ela. O programa que segue mostra o conteúdo de um ponteiro para `double` que não foi iniciado e, portanto, contém lixo.

```
/*
 * Uso de um ponteiro sem valor atribuído
 * Assegura: apresentação do endereço de uma variável
 */
#include <stdio.h>

int main(void) {
    double *ponteiro_para_double;
    printf("%p.\n", (void *)ponteiro_para_double);  // lixo

    return 0;
}
```

```
main.c: In function ´main´:
main.c:9:5: warning: ´ponteiro_para_double´ is used uninitialized
[-Wuninitialized]
    9 |     printf("%p.\n", (void *)ponteiro_para_double);  // lixo
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.c:8:13: note: ´ponteiro_para_double´ was declared here
    8 |     double *ponteiro_para_double;
      |             ^~~~~~~~~~~~~~~~~~~~
```

```
0x7f28a83b9ad0.
```

Porém, há uma diferença entre ter uma variável com valor inválido (nada foi atribuído a ela) e ter uma variável que "não aponta para nada". Em C, o valor `NULL` é usado para indicar explicitamente que uma variável não é referência para um endereço real.

```
/*
 * Uso de um ponteiro sem valor atribuído
 * Assegura: apresentação do endereço de uma variável
 */
#include <stdio.h>

int main(void) {
    double *ponteiro_para_double = NULL;  // endereço explicitamente inválido
    printf("%p.\n", (void *)ponteiro_para_double);

    return 0;
}
```

```
(nil).
```

Na computação em geral, termos como *null*, *nil* ou *nulo* são usados para se referir a um endereço sabidamente inválido. Em C esse valor é expresso por `NULL` e permite comparações, como `if (p != NULL)`, por exemplo.

Para recordar essa situação é possível citar o acesso a arquivos. Uma variável para guardar um arquivo lógico é um ponteiro do tipo `FILE *`, ou seja, guarda uma referência (endereço) de um objeto do tipo `FILE`. Quando uma chamada à função `fopen` não consegue acessar o arquivo, ela retorna `NULL`, Em outras palavras, `fopen` retorna o endereço de algo válido em caso de sucesso ou o endereço especial `NULL` para indicar que o endereço não pode ser usado. Todas as demais funções (`fprint`, `fgets`, `fclose`) apenas usam o endereço válido quando são chamadas.

20.4 Manipulação da memória com uso de ponteiros

Uma aplicação importante de ponteiros é a possibilidade de, tendo em mãos um endereço, modificar o que há naquele local. Assim, se um ponteiro contém o endereço de um inteiro, é viável ver e alterar o valor apontado.

Um exemplo inicial simples é apresentado na sequência, ilustrando como o ponteiro pode ser usado para acessar uma posição de memória.

```
/*
 * Uso de ponteiro para ter acesso a um valor armazenado na memória
 * Assegura: Apresentação do valor de duas variáveis usando um ponteiro
 */
#include <stdio.h>

int main(void) {
    int n1 = 75;
    int n2 = -3;
    int *ponteiro;

    // Uso do ponteiro com n1
    ponteiro = &n1;  // guarda em ponteiro a referência para n1
    printf("Valor apontado: %d.\n", *ponteiro);

    // Uso do ponteiro com n2
    ponteiro = &n2;  // altera a referência para n2
    printf("Valor apontado: %d.\n", *ponteiro);

    return 0;
}
```

```
Valor apontado: 75.
Valor apontado: -3.
```

Neste programa são criadas duas variáveis `int`: `n1` e `n2`. À primeira é atribuído o valor 75 e à segunda, -3. Uma variável `ponteiro` é criada para guardar o endereço de um valor `int` e o endereço de `n1` é armazenado, conforme destacado na sequência.

```
int *ponteiro;  // criação de uma variável ponteiro
ponteiro = &n1;  // armazenamento do endereço de n1
```

Nesse momento, com `ponteiro` contendo o endereço de `n1`, a expressão `*ponteiro` se refere ao valor inteiro guardado nesse endereço. Em outras palavras, `ponteiro` aponta para `n1` e, em consequência, `*ponteiro` dá o valor apontando, que é o valor de `n1`. Assim, o `printf` usa essa valor para "espiar" em `n1`.

O programa então dá instruções para que `ponteiro` aponte para `n2` para, em seguida, usar `*ponteiro` para acessar seu valor, conforme destaque seguinte.

```
ponteiro = &n2;  // altera a referência para n2
printf("Valor apontado: %d.\n", *ponteiro);
```

É importante notar que, nesse programa, `ponteiro` é do tipo `int *` e guarda endereços de elementos inteiros e, por sua vez, `*ponteiro` é do tipo `int`, pois olha o conteúdo apontado naquele endereço. A Tabela 20.1 mostra alguns casos e os tipos associados à notação sem e com o operador `*`.

Tabela 20.1: Algumas declarações de ponteiros e os tipos associados ao identificador e ao uso do operador `*`.

Declaração	Tipos associados	Exemplos
`char *pc`	`pc` é `char *` `*pc` é `char`	`pc = &c` `prinf("%c", *pc)`
`int *pi`	`pi` é `int *` `*pi` é `int`	`pi = &i` `printf("%d", *pi)`
`double *pd`	`pd` é `double *` `*pd` é `double`	`pd = &d` `printf("%g", *pd)`
`unsigned int *pui`	`pui` é `unsigned int *` `*pd` é `unsigned int`	`pui = &ui` `printf("%u", *pui)`

De forma similar ao se obter o conteúdo referenciado por um ponteiro, também é possível modificar o valor apontado. Segue um programa para exemplificar essa situação.

```
/*
 * Modificação de um valor inteiro com uso de um ponteiro
 * Assegura: Apresentação do valor do inteiro antes e depois da modificação
 */
#include <stdio.h>

int main(void) {
    int valor_inteiro = 123;
    printf("Valor da variável: %d.\n", valor_inteiro);

    int *ponteiro = &valor_inteiro;
    *ponteiro = 98765;
    printf("Valor da variável: %d.\n", valor_inteiro);

    return 0;
}
```

```
Valor da variável: 123.
Valor da variável: 98765.
```

A variável `valor_inteiro` é declarada, tem o valor 123 atribuído a ela e esse valor é apresentado. Então a variável `ponteiro` é criada e o endereço de `valor_inteiro` é armazenado nela. A modificação de valor é feita pelo comando destacado.

```
*ponteiro = 98765;  // modifica o valor do endereço guardado em ponteiro
```

Essa instrução, basicamente, diz "coloque o valor 98765 no endereço armazenado em `ponteiro`". Nesse caso, como `ponteiro` aponta para `valor_inteiro`, os bytes dessa última variável serão alterados. O resultado é que, em última instância, `valor_inteiro` tem seu conteúdo atualizado.

Desse modo, `*ponteiro` pode ser tanto usado para obter o valor apontado quanto para modificá-lo.

```
int i, j, *pi;  // i e j inteiros, pi ponteiro para inteiro

i = 10;
pi = &i;

j = *pi;  // copia o valor 10 (de i) para j
*pi = 1;  // coloca o valor 1 em i
```

20.5 Os ponteiros têm tipos

Os ponteiros são sempre declarados usando um tipo (`char`, `int`, `double` etc.) e especificando um asterisco antes do identificador, conforme os exemplos que seguem.

```
int *pi;  // ponteiro para int
char *pc;  // ponteiro para char
long double *pld;  // ponteiro para long double
unsigned char *puc;  // ponteiro para um unsigned char
```

Os tipos são importantes para o compilador lidar com as diversas operações. Assim, atribuições usando `*pi` do lado esquerdo do operador de atribuição tratarão uma atribuição para inteiro; ao se escrever `*pi + *pld`, as promoções de tipo serão feitas segundo as regras; ou para usar o `printf` para mostrar `*puc` deve ser usado o formato `%u`, pois seu tipo é `unsigned char`.

Segue um exemplo em que há mistura de tipos e, dada a mistura, os resultados divergem dos esperados.

```
/*
 * Uso do tipo incorreto para um ponteiro
 * Assegura: Apresentação do valor do inteiro antes e depois da modificação
 */
#include <stdio.h>

int main(void) {
    float f = -1.1;

    int *p = (int *)&f;  // usa int* para apontar para float
    *p = 1000;  // altera o conteúdo de d

    printf("d = %g.\n", f);

    return 0;
}
```

```
d = 1.4013e-42.
```

Como as representações de `float` e `int` são diferentes, a variável `f` tenta extrair um valor real a partir de um conjunto de bits que, na realidade, representa um inteiro. A conclusão é simples: não funciona.

20.6 Ponteiros para cadeias de caracteres

Em C, uma cadeia de caracteres é uma sequência de bytes contínuos na memória (os caracteres) seguida por um byte nulo `\0`. É dessa forma, por exemplo, que a função `printf`, quando usa o formato `%s`, interpreta a memória e decide o que apresentar na tela.

Como está introduzido na Capítulo 16, há cadeias de caracteres constantes e também em variáveis. Em particular, a Seção 16.3 mostra como usar ponteiros para referenciar as constantes literais existentes em um programa.

Da mesma forma que é possível ter uma variável do tipo `char *` apontando para uma constante, também é comum que esse ponteiro seja usado para apontar para uma variável.

Para os ponteiros lidarem com cadeias de caracteres há dois pontos principais: o ponteiro mantém o endereço do primeiro byte da cadeia e o fim da cadeia é indicado pelo terminador `\0`.

O programa seguinte mostra o uso de ponteiros tanto para constantes quanto para variáveis.

```c
/*
 * Ponteiros para cadeias de caracteres
 * Assegura: apresentação de algumas cadeias de caracteres
 */
#include <stdio.h>

int main(void) {
    char *texto_ponteiro = "Texto constante";  // ponteiro para constante
    printf("Constante: %s.\n", texto_ponteiro);

    char texto_variavel[100] = "Texto de iniciação da variável";  // variável
    printf("Variável: %s.\n", texto_variavel);

    char *outro_ponteiro;
    outro_ponteiro = texto_ponteiro;  // também aponta para a constante
    printf("Constante de novo: %s.\n", outro_ponteiro);

    outro_ponteiro = texto_variavel;  // aponta para a variável
    printf("Variável via ponteiro: %s.\n", outro_ponteiro);

    return 0;
}
```

```
Constante: Texto constante.
Variável: Texto de iniciação da variável.
Constante de novo: Texto constante.
Variável via ponteiro: Texto de iniciação da variável.
```

A variável `texto_ponteiro` é um ponteiro e contém o endereço do primeiro byte da constante `"Texto constante"`. Por sua vez, `texto_variavel` já é um espaço para uma cadeia de até 99 bytes, ao qual também é copiado um valor inicial.

Uma terceira variável `outro_ponteiro` é usada, em um primeiro momento, para apontar para a constante, o que é feito copiando-se o endereço armazenado em `texto_ponteiro`.

```
outro_ponteiro = texto_ponteiro;  // copia a referência para outro_ponteiro
```

Essa mesma variável é usada para, em um segundo momento, apontar para o primeiro byte da variável `texto_variavel`.

```
outro_ponteiro = texto_variavel;  // aponta para a variável
```

Aqui, é importante observar que existe uma variável literal denominada `texto_variavel` e o uso deste identificador não se refere ao texto armazenado nela, mas ao seu endereço. Na prática, `texto_variavel` equivale a `&texto_variavel[0]`, ou seja, ao endereço do primeiro byte da variável. Esta é uma das razões pelas quais a atribuição direta a variáveis textuais em C não funciona, como exemplificado na sequência.

```
char nome[100] = "Cervantes"  // variável
char outro_nome[100];

outro_nome = nome;  // não funciona, pois 'nome' é um endereço e não
                    // o texto "Cervantes"
```

20.7 Exemplos

Nesta seção são apresentados alguns programas relativamente simples e genéricos que usam ponteiros, tendo como objetivo reforçar os conceitos e indicar alguns de seus usos.

20.7.1 Selecionando o menor valor

O exemplo seguinte usa um ponteiro para modificar o valor de uma entre duas variáveis. A variável a ser modificada é sempre a de valor mínimo.

```
/*
 * Duplicando o valor mínimo com uso de ponteiro
 * Requer: dois valores reais quaisquer
 * Assegura: apresentação do dobro do valor mínimo e do valor máximo original
 */
#include <stdio.h>

int main(void) {
    // Entrada
    printf("Digite dois valores reais: ");
    char entrada[160];
    fgets(entrada, sizeof entrada, stdin);
    double valor1, valor2;
    sscanf(entrada, "%lf%lf", &valor1, &valor2);

    // Duplicação do valor mínimo usando um ponteiro
    double *ponteiro_minimo;
    if (valor1 < valor2)
        ponteiro_minimo = &valor1;
    else
        ponteiro_minimo = &valor2;
    *ponteiro_minimo = *ponteiro_minimo * 2;  // dobra o valor mínimo

    // Apresentação do resultado
    printf("valor1 = %g e valor2 = %g.\n", valor1, valor2);

    return 0;
}
```

```
Digite dois valores reais: 10.7 18.2
valor1 = 21.4 e valor2 = 18.2.
```

Com `valor1` e `valor2` lidos, a variável `ponteiro_minimo` pode apontar tanto para a primeira quanto para a segunda, a depender de seus valores.

A estrutura `if` usada poderia ser substituída por uma atribuição com o condicional ternário:

```
ponteiro_minimo = (valor1 < valor2) ? &valor1 : &valor2;
```

20.7.2 Vários ponteiros para um mesmo local

Assim como nada impede que duas ou mais variáveis `double` tenham um mesmo valor, também é possível que vários ponteiros armazenem o mesmo endereço. Nesse caso, diz-se que vários ponteiros apontam para o mesmo local.

O programa seguinte define uma variável `c` do tipo `char` e usa três ponteiros para referenciá-la.

```
/*
 * Uso de vários ponteiros para um mesmo local
 * Assegura: apresentação dos valores apontados dadas algumas modificações
 */
#include <stdio.h>

int main(void) {
    char c = 'A';

    char *p1, *p2, *p3;
    p1 = p2 = p3 = &c;  // todos os ponteiros apontam para c
    printf("c = %c; *p1 = %c; *p2 = %c; *p3 = %c.\n", c, *p1, *p2, *p3);

    c = 'B';
    printf("c = %c; *p1 = %c; *p2 = %c; *p3 = %c.\n", c, *p1, *p2, *p3);

    *p1 = 'C';
    printf("c = %c; *p1 = %c; *p2 = %c; *p3 = %c.\n", c, *p1, *p2, *p3);

    *p2 = 'D';
    printf("c = %c; *p1 = %c; *p2 = %c; *p3 = %c.\n", c, *p1, *p2, *p3);

    *p3 = 'E';
    printf("c = %c; *p1 = %c; *p2 = %c; *p3 = %c.\n", c, *p1, *p2, *p3);

    return 0;
}
```

```
c = A; *p1 = A; *p2 = A; *p3 = A.
c = B; *p1 = B; *p2 = B; *p3 = B.
c = C; *p1 = C; *p2 = C; *p3 = C.
c = D; *p1 = D; *p2 = D; *p3 = D.
c = E; *p1 = E; *p2 = E; *p3 = E.
```

Como todas as variáveis (`c`, `*p1`, `*p2` e `*p3`) estão se referenciando ao mesmo `char` na memória, esse caractere pode ser modificado por qualquer uma delas.

20.7.3 Copiando referências

Ao se atribuir o endereço de uma variável a um ponteiro, o valor armazenado é o endereço de memória do primeiro byte dessa variável. Se houver outra variável do tipo ponteiro, esse endereço pode ser copiado para ela com uma atribuição simples. Dessa forma, como no exemplo da Seção 20.7.2, o resultado é que se tem mais de um ponteiro referenciando uma mesma posição.

```
/*
 * Cópia de referência entre ponteiros
 * Assegura: apresentação dos valores apontados dadas algumas modificações
 */
#include <stdio.h>

int main(void) {
    char c = 'A';

    char *p1 = &c;
    char *p2 = p1;
    printf("c = %c; *p1 = %c; *p2 = %c.\n", c, *p1, *p2);

    c = 'X';
    printf("c = %c; *p1 = %c; *p2 = %c.\n", c, *p1, *p2);

    return 0;
}
```

```
c = A; *p1 = A; *p2 = A.
c = X; *p1 = X; *p2 = X.
```

Neste programa, `p1` aponta para `c` ao receber `&c`. Para `p2` é atribuído o valor de `p1`, o qual, nesse momento, é igual ao `&c`.

21 Procedimentos em C

A modularização envolve as funções, como apresentado no Capítulo 18. Há casos, porém, em que há necessidades de um módulo para realizar uma ação, mas ele não precisa retornar um valor. Módulos que não retornam valores são chamados procedimentos e, em C, são funções que não voltam valor.

Este capítulo trata das funções sem retorno de valor.

21.1 Funções sem retorno

Um exemplo simples de função que não retorna nada é a função (procedimento) `perror`, cuja função é apresentar uma mensagem de erro, como o programa seguinte que mostra as mensagens referentes aos códigos de erro de zero até nove.

```c
/*
 * Mensagens de erro padrão da biblioteca C
 * Assegura: apresentação das 10 primeiras mensagens de erro
 */
#include <stdio.h>
#include <errno.h>

int main(void) {
    for (int i = 0; i < 10; i++) {
        errno = i;  // ajusta o erro para o valor de i
        perror(NULL);  // mensagem referente ao erro i
    }

    return 0;
}
```

```
Success
Operation not permitted
No such file or directory
No such process
Interrupted system call
Input/output error
No such device or address
Argument list too long
Exec format error
Bad file descriptor
```

O objetivo de `perror` é apresentar uma mensagem na tela de acordo com o valor de `errno`, artificialmente ajustado com o valor de `i` a cada repetição. Essa função não retorna nenhum valor. Essa função tem a declaração seguinte.

```c
void perror(const char *s);
```

O tipo de retorno é especificado como `void`, que poderia ser entendido como "nada", o que indica que essa função somente pode ser usada como um comando e nunca em uma expressão.

Outro exemplo de procedimento da biblioteca padrão é `bzero`[1], que preenche todos os bytes de uma cadeia de caracteres com o byte nulo e não retorna qualquer valor.

[1] A função `bzero`, embora seja um bom exemplo para este texto, é obsoleta e não deve ser usada.

```
void bzero(void s[.n], size_t n);
```

> **Nota**
>
> Embora o nome procedimento se refira claramente a um conjunto de instruções que não retornam valor, é comum na linguagem C o uso do nome função. Assim, de forma geral, algumas vezes uma rotina será chamada de procedimento, mas no geral, retornando ou não valores, os módulos com rotinas serão chamados genericamente de funções.
> O leitor, portanto, não deve assumir que o nome função se refere necessariamente à rotina que retorna um valor.

21.2 Criação de procedimentos

Para escrever um procedimento, basta que seu tipo de retorno seja `void`. Os parâmetros podem ser quantos e de quais tipos forem necessários.

Segue um exemplo de um procedimento que apresenta uma mensagem (bom dia, boa tarde ou boa noite) dependendo da hora do dia.

```
/*!
 * Apresenta na tela uma saudação dependente da hora do dia
 * @param hora: hora do dia, de 0 a 23
 */
void faca_uma_saudacao(int hora);
```

A implementação dessa função pode ser observada no programa exemplo que segue.

```
/*
 * Saudação cordial
 * Assegura: apresentação de uma mensagem de acordo com o horário (bom dia,
 *  boa tarde ou boa noite)
 */
#include <stdio.h>
#include <time.h>

/*!
 * Retorna a hora atual
 * @return hora segundo o sistema operacional, de 0 a 24
 */
int obtenha_hora_atual();

/*!
 * Apresenta na tela uma saudação dependente da hora do dia (bom dia,
 *  boa tarde ou boa noite)
 */
void faca_uma_saudacao(int hora);

/*
 * Main
 */
int main(void) {
    printf("Às 8 horas: ");
    faca_uma_saudacao(8);

    printf("Às 17 horas: ");
    faca_uma_saudacao(17);

    printf("Às 22 horas: ");
    faca_uma_saudacao(22);

    int hora_atual = obtenha_hora_atual();
    printf("Agora, %d hora(s): ", hora_atual);
```

```
    faca_uma_saudacao(hora_atual);

    return 0;
}

// Apresenta saudação dependente da hora
void faca_uma_saudacao(int hora) {
    if (hora <= 11)
        printf("Bom dia!\n");
    else if (hora <= 18)
        printf("Boa tarde!\n");
    else
        printf("Boa noite!\n");
}

// Retorna a hora atual do sistema
int obtenha_hora_atual() {
    time_t tempo = time(NULL);
    struct tm horario = *localtime(&tempo);
    return horario.tm_hour;
}
```

```
Às 8 horas: Bom dia!
Às 17 horas: Boa tarde!
Às 22 horas: Boa noite!
Agora, 11 hora(s): Bom dia!
```

O procedimento `faca_uma_saudacao` escolhe qual saudação apresentar, havendo em `main` três exemplos fixos e um dependente da hora real. É relevante notar que os procedimentos dispensam o `return`, apenas terminando naturalmente depois de executar a última instrução interna.

A função `obtenha_hora_atual` retorna um valor de zero a 23 dependendo da hora atual do sistema operacional. Os detalhes das funções envolvidas não são relevantes e não são discutidos.

Como talvez tenha já sido constado pelo leitor, o uso de procedimentos costuma ser menor que o de funções (que voltam valores).

22 Parâmetros das funções na linguagem C

Neste capítulo são retomados os conceitos de parâmetros de funções, detalhando seu funcionamento interno e dando alternativas para algumas necessidades reais dos programas.

22.1 Passagem de parâmetros por valor

A ideia de se ter parâmetros nas funções é bastante intuitivo. Considerando-se uma função simples, pode ser escrito o código seguinte para o cálculo do quadrado de um valor real.

```
/*!
 * Retorna x^2
 * @param x
 * @return x^2
 */
double quadrado(double x);
```

A implementação dessa função pode ser dada como se segue.

```
// Retorna o quadrado de x
double quadrado(double x) {
    return x * x;
}
```

Dessa forma, um comando usando essa função é bastante direto.

```
double x = 32.2;
double y = quadrado(x);

printf("%g^2 = %g.\n", x, y);
```

Como não existe uma única forma de se implementar uma função, uma nova versão pode ser usada, como a que é apresentada na sequência.

```
// Retorna o quadrado de x
double quadrado(double x) {
    x = x * x;  // transforma x em x^2
    return x;
}
```

Para essa implementação, não há dúvidas de que o valor retornado sempre será igual ao da primeira implementação. A diferença aqui é que o parâmetro `x` é modificado dentro da função.

O que se pode colocar em dúvida aqui é, ao se escrever `y = quadrado(x)`, o valor de `x` não é também alterado? Da discussão apresentada no Capítulo 19 já é possível deduzir que o parâmetro `x` é uma declaração local e, portanto, independente de qualquer outro `x` que exista no programa. A execução do programa exemplo seguinte demonstra isso.

```
/*
 * Cálculo do quadrado de um valor real
 * Assegura: apresentação do quadrado de um valor exemplo
 */
#include <stdio.h>

/*!
 * Retorna x^2
 * @param x
 * @return x^2
 */
double quadrado(double x);

/*
 * Main
 */
int main(void) {
    double x = 32.2;
    double y = quadrado(x);

    printf("%g^2 = %g.\n", x, y);

    return 0;
}

// Retorna o quadrado de x
double quadrado(double x) {
    x = x * x;
    return x;
}
```

```
32.2^2 = 1036.84.
```

No `printf` usado em `main` é possível verificar que o valor de `x` local continua com o valor inicialmente atribuído.

Para um segundo exemplo, um outro programa é apresentado, o qual contém uma função para apresentar uma cadeia de caracteres entre aspas.

```
/*
 * Cálculo do quadrado de um valor real
 * Assegura: apresentação do quadrado de um valor exemplo
 */
#include <stdio.h>
#include <string.h>

/*!
 * Escreve o texto na tela entre aspas
 * @param texto: referência ao texto que será escrito
 */
void escreva_entre_aspas(char *texto);

/*
 * Main
 */
int main(void) {
    // Constante textual
    char *texto_constante = "Programação em C é legal!";
    escreva_entre_aspas(texto_constante);
    printf("\n");

    // Variável textual
    char texto_variavel[100];
    strncpy(texto_variavel, "Outras linguagens também são!",
            sizeof texto_variavel - 1);
    escreva_entre_aspas(texto_variavel);
    printf("\n");

    return 0;
}
```

```
// Escreve texto entre aspas
void escreva_entre_aspas(char *texto) {
    printf("\"%s\"", texto);
}
```

```
"Programação em C é legal!"
"Outras linguagens também são!"
```

Neste caso, como cadeias de caracteres não podem ser copiadas diretamente em C, a opção foi usar as referências às cadeias, como está apresentado na Seção 20.6. O “truque” é passar o endereço do que se deseja apresentar na tela, lembrando que, em `main`, nas duas chamadas para o procedimento `escreva_entre_aspas` o argumento é sempre o endereço do primeiro byte da cadeia de caracteres de interesse.

Para especificar claramente as passagens de parâmetros, no exemplo da função `quadrado`, o parâmetro `x` recebe uma cópia do valor do `x`. Para o caso da função `escreva_entre_aspas`, o parâmetro `texto` (que é um ponteiro), recebe uma cópia do valor do endereço de `texto_constante` (na primeira chamada) e de `texto_variavel` (na segunda).

Em programação, quando um parâmetro recebe a cópia do valor de seu argumento, essa passagem de parâmetros é chamada passagem por valor.

22.2 Passagem de referências

Para introduzir uma nova necessidade, um novo programa é apresentado. Nele, dois valores são dados pelo usuário e eles devem ser trocados se estiverem em ordem decrescente. Esta implementação é uma nova versão para o Algoritmo 7.1, agora usando um procedimento.

```
/*
Apresentação de dois valores em ordem não decrescente
Requer: dois valores reais v1 e v2
Assegura: apresentação de v1 e v2, v1 <= v2
*/
#include <stdio.h>

/*!
 * Troca o valor do primeiro com o do segundo
 * @param valor1: o primeiro valor
 * @param valor2: o segundo valor
 */
void troque_valores(double valor1, double valor2);

/*
 * Main
 */
int main(void) {
    printf("Digite dois valores reais: ");
    char entrada[160];
    fgets(entrada, sizeof entrada, stdin);
    double v1, v2;
    sscanf(entrada, "%lf%lf", &v1, &v2);

    if (v2 < v1)
        troque_valores(v1, v2);

    printf("Valores em ordem não decrescente: %g e %g.\n", v1, v2);

    return 0;
}

// Troca os valores de valor1 e valor2 (versão incorreta)
void troque_valores(double valor1, double valor2) {
    double temporario = valor1;
```

```
    valor1 = valor2;
    valor2 = temporario;
}
```

```
Digite dois valores reais: 120 28
Valores em ordem não decrescente: 120 e 28.
```

Há uma óbvia frustração ao se executar o programa, visto que os valores não são apresentados na ordem esperada. Isso se deve à passagem por valor de `v1` e `v2`, pois efetivamente os conteúdos de `valor1` e `valor2` são trocados, porém as variáveis usadas nas chamadas são mantidas intactas.

Muitas linguagens dispõe de uma forma de passagem de parâmetros conhecida como passagem por referência, de forma que modificações feitas nos parâmetros se refletem nas variáveis usadas para chamar a função. Seguem exemplos de implementações com passagem por referência em Pascal e C++.

Em Pascal, a palavra chave `var` indica que os dados serão alterados externamente ao procedimento.

```
{ Troca os valores de Valor1 e Valor2 }
procedure TroqueValores(var Valor1, Valor2: Real);
var Temporario: Real;
begin
    Temporario := Valor1;
    Valor1 := Valor2;
    Valor2 := Temporario;
end;
```

Em C++, o `var` do Pascal é substituído por `&`, com o mesmo significado.

```
// Troca os valores de valor1 e valor2
void troque_valores(double &valor1, double &valor2) {
    double temporario = valor1;
    valor1 = valor2;
    valor2 = temporario;
}
```

Em C, porém, há um limitante importante para códigos similares: a linguagem não possui passagem por referência. A solução, portanto, é contornar essa restrição.

Conforme se discute no Capítulo 20, é possível alterar o valor de uma variável se houver um ponteiro que a referencie. É essa a estratégia usada nos programas em C, passando os endereços das variáveis que devem ser modificadas e usando os ponteiros para fazer as alterações desejadas.

Segue a versão funcional do programa para a troca dos valores das variáveis.

```
/*
Apresentação de dois valores em ordem não decrescente
Requer: dois valores reais v1 e v2
Assegura: v1 <= v2
*/
#include <stdio.h>

/*!
 * Troca o valor do primeiro com o do segundo
 * @param valor1: referência ao primeiro valor
 * @param valor2: referência ao segundo valor
 */
void troque_valores(double *valor1, double *valor2);

/*
 * Main
 */
```

```
int main(void) {
    char entrada[160];

    printf("Digite dois valores reais: ");
    fgets(entrada, sizeof entrada, stdin);
    double v1, v2;
    sscanf(entrada, "%lf%lf", &v1, &v2);

    if (v2 < v1)
        troque_valores(&v1, &v2);

    printf("Valores em ordem não decrescente: %g e %g.\n", v1, v2);

    return 0;
}

// Troca os valores de valor1 e valor2
void troque_valores(double *valor1, double *valor2) {
    double temporario = *valor1;
    *valor1 = *valor2;
    *valor2 = temporario;
}
```

```
Digite dois valores reais: 120 28
Valores em ordem não decrescente: 28 e 120.
```

Inicialmente, é preciso notar que os parâmetros para chamar as funções não são mais os valores das variáveis, mas seus endereços.

```
troque_valores(&v1, &v2);  // passagem dos endereços das variáveis
```

Para receber os endereços, os parâmetros formais da função agora são ponteiros.

```
void troque_valores(double *valor1, double *valor2);  // parâmetros: ponteiros
```

Na implementação da função, `temporario` é uma variável local do tipo `double`, sem nada de especial. Para realizar as trocas, o conteúdo apontado pelos parâmetros é usado.

```
double temporario = *valor1;  // copia o que está no endereço passado para o
                              // primeiro parâmetro (valor1) para temporario
*valor1 = *valor2;  // copia o conteúdo de um endereço para o outro
*valor2 = temporario;  // recupera o temporario e o copia para o endereço
                       // do segundo parâmetro (valor2)
```

Na linguagem C, como não há passagem por referência, as referências têm que ser passadas explicitamente.

22.2.1 Cadeias de caracteres são sempre passagem por referência

Para as variáveis que armazenam cadeias de caracteres, tem-se que seu identificador já representa seu endereço. Em consequência, quando a função `fgets` é usada, seu primeiro parâmetro é uma passagem da referência. Isso é ilustrado a seguir.

```
char texto[100];
fgets(texto, sizeof texto, stdin);  // o parâmetro texto é o endereço onde
                                    // os dados da variável são guardados
```

Dessa forma a função `fgets` consegue, por meio do ponteiro passado, modificar o conteúdo da variável `texto`. A declaração de `fgets` é equivalente à apresentada na sequência[1].

```
char *fgets(char s*, int size, FILE *stream);
```

O primeiro parâmetro da função espera que seja passado um endereço de um `char`, que é para onde os bytes lidos serão transferidos.

A consequência é que, na linguagem C, sempre são passadas as referências (endereços) das cadeias de caracteres.

Dica

Quando se desejar passar uma cadeia de caracteres por valor (o que não é tecnicamente possível), pode-se contornar o problema especificando que a função não tem autorização para mudar o conteúdo da variável. O valor passado como parâmetro continua sendo a referência, mas o compilador gerará um erro caso o programador tente modificar o valor.
Segue um exemplo simples.

```
/*
 * Exemplo de erro ao tentar mudar uma cadeia de caracteres
 */
#include <stdio.h>

/*!
 * Apresenta uma mensagem na tela
 * @param mensagem: texto a ser apresentado
 */
void apresente_mensagem(char const *mensagem);

/*
 * Main
 */
int main(void) {
    char minha_mensagem[] = "Olá a todos!";
    apresente_mensagem(minha_mensagem);

    return 0;
}

// Apresenta a mensagem
void apresente_mensagem(const char *mensagem) {
    mensagem[0] = 'X';  // tentativa de alteração
    printf("%s\n", mensagem);
}
```

```
main.c: In function ´apresente_mensagem´:
main.c:24:17: error: assignment of read-only location ´*mensagem´
   24 |     mensagem[0] = 'X';  // tentativa de alteração
      |                 ^
```

O modificador `const` adicionado ao parâmetro alerta o compilador que o identificador `mensagem` (parâmetro) não pode ser usado para alterar o valor.

22.3 Exemplos

Nesta seção alguns exemplos interessantes são apresentados.

[1] A declaração foi levemente modificada para maior clareza, porém mantendo a equivalência.

22.3.1 “Já usei passagem por referência muitas vezes”

Ao longo dos inúmeros exemplos usados neste livro, a passagem por referência já vinha sendo utilizada. Por exemplo, na leitura de um valor real, a conversão é feita pelo `sscanf`, que pode ter o formato seguinte.

```
sscanf(entrada, "%lf", &valor);
```

É importante notar que o endereço da variável `valor` foi passado para `sscanf`. Depois de interpretar a cadeia de caracteres em `entrada` procurando por um `%lf` presente, o resultado dessa conversão é guardado como um `double` na posição de memória passada, ou seja, exatamente onde está a variável `valor`.

Sem o operador `&`, a função `sscanf` não conseguiria ter acesso à variável `valor`, que tem escopo externo a ela, e usa seu endereço para conseguir modificá-la.

E, naturalmente, também `fgets` usa referência para fazer a leitura.

22.3.2 Simplificação de números racionais

Um número racional ($\mathbb{Q}$) é aquele que pode ser escrito na forma a/b, sendo $a \in \mathbb{Z}$ com $b \in \mathbb{Z}^*$. O Algoritmo 19.1 apresenta uma solução de como fazer a simplificação de um valor racional, padronizando sua apresentação.

A seguir é apresentada uma versão de implementação desse algoritmo, agora com o emprego de um procedimento para fazer a simplificação.

```
/*
 * Leitura e escrita de um número racional na forma de fração
 * Requer: a digitação de um valor a/b, a,b inteiros, a, b != 0
 * Assegura: apresentação do mesmo valor em forma simplificada e padronizada
 */
#include <stdio.h>

/*!
 * Simplificação de um número racional dados numerador e denominador
 * @param numerador: numerador
 * @param denominador: denominador (não nulo)
 */
void simplifique_racional(int *numerador, int *denominador);

/*!
 * Retorna o MDC de dois inteiros quaisquer (máximo divisor comum).
 * @param n1: primeiro valor
 * @param n2: segundo valor
 * @return MDC(n1, n2)
 */
unsigned int mdc(int n1, int n2);

/*!
 * Retorna o valor absoluto de um inteiro
 * @param n: valor inteiro
 * @return o valor absoluto do número, |n|
 */
int valor_absoluto(int n);

/*
 * Main
 */
int main() {
    // Leitura
    printf("Digite um número racional a/b (a e b inteiros e b não nulo): ");
    char entrada[160];
    fgets(entrada, sizeof entrada, stdin);
```

```
    int numerador, denominador;
    sscanf(entrada, "%d/%d", &numerador, &denominador);

    // Simplificação e apresentação da razão
    simplifique_racional(&numerador, &denominador);
    printf("O racional digitado foi: %d/%d.\n", numerador, denominador);

    return 0;
}

// Máximo divisor comum: MDC(n1, n2)
unsigned int mdc(int n1, int n2) {
    // Converte n1 e n2 para valores positivos
    n1 = valor_absoluto(n1);
    n2 = valor_absoluto(n2);

    // Resolução pelo método de Euclides
    int resto;
    do {
        resto = n1 % n2;
        n1 = n2;
        n2 = resto;
    } while (resto != 0);

    return (unsigned int)n1;  // Contém o MDC no final
}

// Valor absoluto de um inteiro
int valor_absoluto(int n) {
    return (n >= 0) ? n : -n;
}

// Simplificação de um racional
void simplifique_racional(int *numerador, int *denominador) {
    if (*numerador == 0) {
        *denominador = 1;  // padroniza zero para 0/1
    }
    else {
        int sinal_da_fracao = ((double)*numerador / *denominador >= 0) ? 1 : -1;
        int fator_divisao = (int)mdc(*numerador, *denominador);
        *numerador = valor_absoluto(*numerador);
        *denominador = valor_absoluto(*denominador);
        *numerador /= sinal_da_fracao * fator_divisao;
        *denominador /= fator_divisao;
    }
}
```

```
Digite um número racional a/b (a e b inteiros e b não nulo): 18/-26
O racional digitado foi: -9/13.
```

A função `simplifique_racional` determina o sinal (-1 ou 1, o qual será associado ao numerador) e qual o MDC entre o numerador e o denominador. Como são passadas referências às variáveis que estão em `main`, dentro da função são usados `*numerador` e `*denominador` para acesso às variáveis externas. A chamada para a simplificação, também, requer que os endereços das variáveis sejam passados.

```
simplifique_racional(&numerador, &denominador);  // endereços das variáveis
```

Um ponto positivo de ter funções para realizar as diversas tarefas é que a função `main` fica mais simples e mais legível.

Parte VII

Estruturação de dados

23 C: Dados com `struct`

Este capítulo discute como variáveis compostas heterogêneas, comumente chamadas de registros, são declaradas e usadas em C. São cobertos os aspectos de declaração, atribuição e seu uso com funções.

23.1 Variáveis compostas

A exploração de variáveis compostas se inicia com a apresentação de um problema envolvendo pontos em $\mathbb{R}^3$. Se três pontos no espaço são especificados, então um triângulo é formado e sua área pode ser calculada. O Algoritmo 23.1 apresenta a estratégia para esse cálculo.

Algoritmo 23.1: Cálculo da área de um triângulo no espaço $\mathbb{R}^3$.

Descrição: Determinação da área de um triângulo dados seus três vértices em $\mathbb{R}^3$
Requer: três pontos em $\mathbb{R}^3$
Assegura: apresentação da área do triângulo formado

Obtenha os pontos P_1, P_2 e P_3
Apresente ÁreaTriângulo(P_1, P_2, P_3)

A função ÁreaTriângulo está apresentada no Algoritmo 23.2, assim como uma função DistânciaPontos necessária à sua execução.

Algoritmo 23.2: Função para cálculo da área de um triângulo dados seus vértices.

Descrição: Cálculo da área de um triângulo dados seus três vértices em $\mathbb{R}^3$
Requer: três pontos em $\mathbb{R}^3$
Assegura: retorno área do triângulo formado

função ÁreaTriângulo(P_1, P_2, P_3)
  Calcule l_1 como DistânciaPontos(P_1, P_2)
  Calcule l_2 como DistânciaPontos(P_1, P_3)
  Calcule l_3 como DistânciaPontos(P_2, P_3)
  Calcule o semiperímetro p como $\dfrac{l_1 + l_2 + l_3}{2}$
  retorne $\sqrt{p(p - l_1)(p - l_2)(p - l_3)}$
fim função

Descrição: Cálculo da distância entre dois pontos em $\mathbb{R}^3$
Requer: pontos P_1 e P_2
Assegura: distância entre os pontos

função DistânciaPontos(P_1, P_2)
  retorne $\sqrt{(P_1.x - P_2.x)^2 + (P_1.y - P_2.y)^2 + (P_1.z - P_2.z)^2}$
fim função

Com base nessas especificações, é possível escrever um programa em C para implementar essa solução.

```c
/*
 * Determinação da área de um triângulo dados seus vértices em R^3
 * Requer: três pontos, cada um composto por suas coordenadas x, y e z
 * Assegura: a apresentação da área do triângulo definido pelos vértices
 */
#include <stdio.h>
#include <math.h>

/*!
 * Retorna a área de um triângulo dados seus vértices
 * @param x1: x de P1
 * @param y1: y de P1
 * @param z1: z de P1
 * @param x2: x de P2
 * @param y2: y de P2
 * @param z2: z de P2
 * @param x3: x de P3
 * @param y3: y de P3
 * @param z3: z de P3
 * @return área do triângulo
 */
double area_triangulo(double x1, double y1, double z1,
                      double x2, double y2, double z2,
                      double x3, double y3, double z3);

/*!
 * Retorna a distância entre P1 e P2
 * @param x1: x de P1
 * @param y1: y de P1
 * @param z1: z de P1
 * @param x2: x de P2
 * @param y2: y de P2
 * @param z2: z de P2
 * @return a distância
 */
```

```
double distancia_pontos(double x1, double y1, double z1,
                        double x2, double y2, double z2);

/*!
 * Leitura de um ponto em R^3
 * @param mensagem: mensagem solicitando a digitação dos dados
 * @param x: referência a x
 * @param y: referência a y
 * @param z: referência a z
 */
void leia_ponto(char *mensagem, double *x, double *y, double *z);

/*
 * Main
 */
int main(void) {
    double x1, y1, z1;
    leia_ponto("Digite as coordenadas de P1: ", &x1, &y1, &z1);  // P1

    double x2, y2, z2;
    leia_ponto("Digite as coordenadas de P2: ", &x2, &y2, &z2);  // P2

    double x3, y3, z3;
    leia_ponto("Digite as coordenadas de P3: ", &x3, &y3, &z3);  // P3

    double area = area_triangulo(x1, y1, z1, x2, y2, z2, x3, y3, z3);
    printf("Área do triângulo: %.2f.\n", area);
}

// Retorna a área do triângulo dados os vértices
double area_triangulo(double x1, double y1, double z1,
                      double x2, double y2, double z2,
                      double x3, double y3, double z3) {
    double lado1 = distancia_pontos(x1, y1, z1, x2, y2, z2); // P1 e P2
    double lado2 = distancia_pontos(x1, y1, z1, x3, y3, z3); // P1 e P3
    double lado3 = distancia_pontos(x2, y2, z2, x3, y3, z3); // P2 e P3
    double semiperimetro = (lado1 + lado2 + lado3) / 2;

    return sqrt(semiperimetro * (semiperimetro - lado1) *
                (semiperimetro - lado2) * (semiperimetro - lado3));
}

// Retorna a distância entre dois pontos
double distancia_pontos(double x1, double y1, double z1,
                        double x2, double y2, double z2) {
    return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2) + pow(z1 - z2, 2));
}

// Leitura de um ponto
void leia_ponto(char *mensagem, double *x, double *y, double *z) {
    printf("%s", mensagem);
    char entrada[160];
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%lf%lf%lf", x, y, z);
}
```

```
Digite as coordenadas de P1: -1 7 3
Digite as coordenadas de P2: -1 0 0
Digite as coordenadas de P3: 2 3 5
Área do triângulo: 17.31.
```

A grande distância entre o algoritmo e a implementação é a complexidade introduzida pelo número de parâmetros que as funções possuem. Enquanto no algoritmo a instrução ÁreaTriângulo(P_1, P_2, P_3) é clara, a codificação `area_triangulo(x1, y1, z1, x2, y2, z2, x3, y3, z3)` é mais longa e propensa a erros[1].

A questão envolvida nesse momento é como representar ÁreaTriângulo(P_1, P_2, P_3) no código `area_triangulo(p1, p2, p3)`.

[1] Este autor reconhece que um certo número de recompilações foi necessário por erros na digitação dos parâmetros ao criar o exemplo.

Para esse fim, cada ponto pode ser estruturado como um registro e, em C, esse agrupamento é feito na forma de um `struct`.

23.2 Declaração e uso do struct

As variáveis compostas, chamadas registros, são criadas com `struct` em C. A declaração de registros na linguagem pode assumir diversos formatos. Neste texto será adotado, com poucas exceções, um formato padronizado para manter a clareza.

Um grupo de informações pode ser agrupado em uma única variável criando-se um registro. Como exemplo, um ponto no espaço, composto de coordenadas x, y e z pode ser estruturado como segue.

```
/*! @struct Ponto em R^3 */
struct ponto {
    double x, y, z;
};
```

Essa instrução define um novo tipo na linguagem, chamado `struct ponto`. Ela é apenas uma declaração e nenhuma variável criada ou espaço de armazenamento é reservado.

Variáveis podem ser criadas usando-se o formato geral de declarações: o tipo da variável é precedido pelo identificadores.

```
struct ponto ponto1, ponto2;  // duas variáveis (ponto1 e ponto2), ambas
                              // do tipo struct ponto
```

23.2.1 Uso e acesso a campos

O acesso aos campos é feito com o operador `.` (ponto). Assim, `ponto1.x` é o campo `x` da variável `ponto1`.

```
// Definição de ponto1 como o ponto (1.5, 0, 3.9)
ponto1.x = 1.5;
ponto1.y = 0;
ponto1.z = 3.9;
```

Uma vantagem interessante de variáveis `struct` é a capacidade de atribuição direta. Por exemplo, a instrução `ponto2 = ponto1` pode ser usada sem problemas. Na prática, o compilador não está ciente dos campos ou de seus valores, mas copia todos os bytes que formam um registro para o outro, resultando em uma cópia idêntica.

23.2.2 Declaração com iniciação

É possível, ao declarar uma varável do tipo registro, já incluir a iniciação de seus valores. O programa seguinte ilustra essa manipulação.

```
/*
 * Exemplo de iniciação de valores de um registro
 * Assegura: apresentação dos valores iniciados
 */
#include <stdio.h>

int main(void) {
    struct pessoa {
        char nome[100];
```

```
        char cpf[15];
        int ano_nascimento;
    };

    struct pessoa alguem = {"Fulano de tal", "123.456.789-00", 2008};

    printf("Nome: %s,\nCPF: %s,\nAno: %d.\n", alguem.nome, alguem.cpf,
           alguem.ano_nascimento);

    return 0;
}
```

```
Nome: Fulano de tal,
CPF: 123.456.789-00,
Ano: 2008.
```

Esse recurso, porém, apenas está disponível na declaração. Tentativas de atribuição posteriores não são aceitas. Nestes casos as atribuições necessariamente têm que ser feitas campo a campo.

```
/*
 * Exemplo de falha na atribuição de registros
 */
#include <stdio.h>

int main(void) {
    struct pessoa {
        char nome[100];
        char cpf[15];
        int ano_nascimento;
    };

    struct pessoa alguem;
    alguem = {"Muriel Gomes Faruak", "123.456.789-00", 2008};  // não funciona!

    printf("Nome: %s,\nCPF: %s,\nAno: %d.\n", alguem.nome, alguem.cpf,
           alguem.ano_nascimento);

    return 0;
}
```

```
main.c: In function ´main´:
main.c:14:14: error: expected expression before ´{´ token
   14 |     alguem = {"Muriel Gomes Faruak", "123.456.789-00", 2008};  //
não funciona!
      |              ^
```

23.3 Modularização com struct

Um ponto positivo do struct na linguagem C é a possibilidade da atribuição de um registro para outro, copiando todos os campos de uma única vez, desde que sejam do mesmo tipo. Essa característica é interessante, em particular, ao passar um registro completo como parâmetro.

Retomando o Algoritmo 23.1, a função para o cálculo da área da distância entre dois pontos pode se apresentar conforme segue, juntamente com a nova versão de DistânciaPontos.

```
/*! @struct ponto */
struct ponto {
    double x, y, z;
};

/*!
 * Retorna a área de um triângulo dados seus vértices
 * @param ponto1: primeiro vértice
 * @param ponto2: segundo vértice
```

```
 * @param ponto3: terceiro vértice
 * @return área do triângulo
 */
double area_triangulo(struct ponto ponto1,
                      struct ponto ponto2,
                      struct ponto ponto3);

/*!
 * Retorna a distância entre P1 e P2
 * @param ponto1: P1
 * @param ponto2: P2
 * @return a distância
 */
double distancia_pontos(struct ponto ponto1, struct ponto ponto2);
```

As funções possuem como parâmetros formais registros do tipo `struct ponto`. A declaração de tipo desse `struct`tem que ser global, uma vez que precisa ser conhecida nas funções subsequentes e, depois, dentro da função `main`.

Na sequência é apresentada a implementação da função `area_triangulo`.

```
// Retorna a área do triângulo dados os vértices
double area_triangulo(struct ponto ponto1,
                      struct ponto ponto2,
                      struct ponto ponto3) {
    double lado1 = distancia_pontos(ponto1, ponto2);
    double lado2 = distancia_pontos(ponto1, ponto3);
    double lado3 = distancia_pontos(ponto2, ponto3);
    double semiperimetro = (lado1 + lado2 + lado3) / 2;

    return sqrt(semiperimetro * (semiperimetro - lado1) *
            (semiperimetro - lado2) * (semiperimetro - lado3));
}
```

Todas as chamadas são por valor, uma vez que uma cópia do registro inteiro é feita na passagem dos parâmetros.

O código completo pode ser, então. apresentado.

```
/*
 * Determinação da área de um triângulo dados seus vértices em R^3
 * Requer: três pontos, cada um composto por suas coordenadas x, y e z
 * Assegura: a apresentação da área do triângulo definido pelos vértices
 */
#include <stdio.h>
#include <math.h>

/*! @struct ponto */
struct ponto {
    double x, y, z;
};

/*!
 * Retorna a área de um triângulo dados seus vértices
 * @param ponto1: primeiro vértice
 * @param ponto2: segundo vértice
 * @param ponto3: terceiro vértice
 * @return área do triângulo
 */
double area_triangulo(struct ponto ponto1,
                      struct ponto ponto2,
                      struct ponto ponto3);

/*!
 * Retorna a distância entre P1 e P2
 * @param ponto1: P1
 * @param ponto2: P2
 * @return a distância
 */
```

```c
double distancia_pontos(struct ponto ponto1, struct ponto ponto2);

/*!
 * Leitura de um ponto em R^3
 * @param mensagem: mensagem solicitando a digitação dos dados
 * @param x: referência a x
 * @param y: referência a y
 * @param z: referência a z
 */
void leia_ponto(char *mensagem, double *x, double *y, double *z);

/*
 * Main
 */
int main(void) {
    struct ponto vertice1;
    leia_ponto("Digite as coordenadas 1: ", &vertice1.x, &vertice1.y,
               &vertice1.z);

    struct ponto vertice2;
    leia_ponto("Digite as coordenadas 2: ", &vertice2.x, &vertice2.y,
               &vertice2.z);

    struct ponto vertice3;
    leia_ponto("Digite as coordenadas 1: ", &vertice3.x, &vertice3.y,
               &vertice3.z);

    double area = area_triangulo(vertice1, vertice2, vertice3);
    printf("Área do triângulo: %.2f.\n", area);
}

// Retorna a área do triângulo dados os vértices
double area_triangulo(struct ponto ponto1,
                      struct ponto ponto2,
                      struct ponto ponto3) {
    double lado1 = distancia_pontos(ponto1, ponto2);
    double lado2 = distancia_pontos(ponto1, ponto3);
    double lado3 = distancia_pontos(ponto2, ponto3);
    double semiperimetro = (lado1 + lado2 + lado3) / 2;

    return sqrt(semiperimetro * (semiperimetro - lado1) *
                (semiperimetro - lado2) * (semiperimetro - lado3));
}

// Retorna a distância entre dois pontos
double distancia_pontos(struct ponto ponto1, struct ponto ponto2) {
    return sqrt(pow(ponto1.x - ponto2.x, 2) +
                pow(ponto1.y - ponto2.y, 2) +
                pow(ponto1.z - ponto2.z, 2));
}

// Leitura de um ponto
void leia_ponto(char *mensagem, double *x, double *y, double *z) {
    printf("%s", mensagem);
    char entrada[160];
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%lf%lf%lf", x, y, z);
}
```

```
Digite as coordenadas 1: -1 7 3
Digite as coordenadas 2: -1 0 0
Digite as coordenadas 1: 2 3 5
Área do triângulo: 17.31.
```

23.3.1 Passagem de registro por referência

Da mesma forma que outras variáveis, é possível passar a referência de um registro para uma função. O endereço de um registro é o endereço de seu primeiro byte, independentemente de seus campos.

No caso do exemplo da área do triângulo, a leitura de cada ponto está passando cada campo de forma independente. Essa situação pode ser modificada passando um único parâmetro (o registro) para a função, todo ele por referência.

Segue a nova versão do procedimento `leia_ponto`.

```
/*!
 * Leitura de um ponto em R^3
 * @param mensagem: mensagem solicitando a digitação dos dados
 * @param ponto: referência a um ponto
 */
void leia_ponto(char *mensagem, struct ponto *ponto);
```

O primeiro parâmetro, `mensagem`, está inalterado e as três coordenadas foram substituídas pelo registro todo. Agora, o parâmetro formal aguarda o endereço de uma variável `struct ponto`, estruturando uma passagem por referência. A chamada para essa função pode ser, por exemplo, como apresentada logo na sequência.

```
struct ponto ponto;
leia_ponto("Digite as coordenadas: ", &ponto);
```

23.3.2 Ponteiros para `struct` e acesso a campos

O entendimento da passagem por referência quando os parâmetros são `struct` requer entender a natureza da notação específica usada em C para esses casos. Dessa forma, uma sequência de exemplos proporcionam o entendimento dos operadores utilizados.

O programa seguinte mostra o uso de um ponteiro para fazer acesso ao conteúdo de um registro.

```
/*
 * Registros e ponteiros para registros
 */
#include <stdio.h>

int main(void) {
    /*! @struct registro com campos diversos */
    struct registro {
        int i;
        double d;
        char c;
    };

    // Variável comum para registro
    struct registro r1 = {9, 3.7, 'x'};
    printf("r1 = %d / %g / '%c'.\n", r1.i, r1.d, r1.c);

    // Ponteiro para registro
    struct registro *pr = &r1;  // pr aponta para r1

    // Uso do ponteiro com outra variável comum
    struct registro r2 = *pr;  // copia r1 para r2
    printf("r2 = %d / %g / '%c'.\n", r2.i, r2.d, r2.c);

    return 0;
}
```

```
r1 = 9 / 3.7 / 'x'.
r2 = 9 / 3.7 / 'x'.
```

Nesse programa, `r1` é um registro com três campos, iniciados com alguns valores ilustrativos. Há, também, uma variável do tipo ponteiro, cujo tipo base é `struct registro`.

```
struct registro *pr;  // para guardar o endereço de um registro
```

Da mesma forma que outras variáveis e respectivos ponteiros, `pr` é usado para guardar um endereço (no caso, de `r1`) e a notação `*pr` permite o acesso aos dados apontado. Dessa forma, escrever `r2 = *pr` equivale, no código, a `r2 = r1`, já que `pr` aponta para `r1`. Em outras palavras, o tipo de `pr` é `struct registro*` (um ponteiro) e de `*pr` é `struct registro` (o registro real que está sendo apontado).

A questão que se apresenta é o acesso aos campos de um registro apontado. C disponibiliza uma notação particular para essa situação, usando o operador `->`, como se ilustra a seguir.

```c
/*
 * Registros e ponteiros para registros
 */
#include <stdio.h>

int main(void) {
    /*! @struct registro com campos diversos */
    struct registro {
        int i;
        double d;
        char c;
    };

    struct registro r1 = {9, 3.7, 'x'};
    printf("r1 = %d / %g / '%c'.\n", r1.i, r1.d, r1.c);

    struct registro *pr = &r1;
    printf("r2 = %d / %g / '%c'.\n", pr->i, pr->d, pr->c);

    return 0;
}
```

```
r1 = 9 / 3.7 / 'x'.
r2 = 9 / 3.7 / 'x'.
```

Quando uma variável `p` é um ponteiro para um `struct`, `*p` é o registro apontado inteiro e `p->c` indica o acesso ao campo `c` de `*p`.

23.3.3 Uso dos ponteiros na passagem por referência

Com o uso de ponteiros, a passagem por referência de um registro pode ser feita. O programa seguinte modifica a função de leitura de um ponto, trocando as três coordenadas separadas pelo registro como um todo. As demais funções ficaram inalteradas e foram suprimidas para simplificar a listagem.

```c
/*
 * Determinação da área de um triângulo dados seus vértices em R^3
 * Requer: três pontos, cada um composto por suas coordenadas x, y e z
 * Assegura: a apresentação da área do triângulo definido pelos vértices
 */
#include <stdio.h>
#include <math.h>

/*! @struct ponto */
struct ponto {
    double x, y, z;
};

// --- Alguns protótipos foram suprimidos

/*!
 * Leitura de um ponto em R^3
 * @param mensagem: mensagem solicitando a digitação dos dados
```

```
 * @param ponto: referência para o ponto
 */
void leia_ponto(char *mensagem, struct ponto *ponto);

/*
 * Main
 */
int main(void) {
    struct ponto vertice1;
    leia_ponto("Digite as coordenadas 1: ", &vertice1);

    struct ponto vertice2;
    leia_ponto("Digite as coordenadas 2: ", &vertice2);

    struct ponto vertice3;
    leia_ponto("Digite as coordenadas 1: ", &vertice3);

    double area = area_triangulo(vertice1, vertice2, vertice3);
    printf("Área do triângulo: %.2f.\n", area);
}

// --- Algumas implementações foram suprimidas

// Leitura de um ponto
void leia_ponto(char *mensagem, struct ponto *ponto) {
    printf("%s", mensagem);
    char entrada[160];
    fgets(entrada, sizeof entrada, stdin);
    sscanf(entrada, "%lf%lf%lf", &ponto->x, &ponto->y, &ponto->z);
}
```

```
Digite as coordenadas 1: -1 7 3
Digite as coordenadas 2: -1 0 0
Digite as coordenadas 1: 2 3 5
Área do triângulo: 17.31.
```

O procedimento `leia_ponto` mantém o primeiro parâmetro original, que é a mensagem apresentada e define mais um único parâmetro que é o `struct` passado como referência (com tipo `struct ponto*`). Dado que dentro desse procedimento o parâmetro `ponto` é um ponteiro, o acesso aos campos é escrito `ponto->x`, `ponto->y` e `ponto->z`. Para o `sscanf`, que também espera passagens por referência, são indicados os endereços de cada campo individualmente: `&ponto->x`, `&ponto->y` e `&ponto->z`.

Na função `main`, a chamada para `leia_ponto` requer que seja passado o endereço do registro que será modificado.

```
leia_ponto("Digite as coordenadas 1: ", &vertice1);
```

23.4 Mais sobre as declarações de `struct`

Declarações de registros na linguagem C apresentam uma certa variedade de formatos. Nesta seção há comentários sobre algumas variações, embora nem todas sejam consideradas.

23.4.1 Declaração sem nome de registro

Em algumas situações, é possível agrupar os campos em um registro sem que o `struct` tenha um nome próprio.

Segue um exemplo

```c
/*
 * Exemplo de struct sem nome
 * Assegura: apresentação dos valores iniciados
 */
#include <stdio.h>
#include <string.h>

int main(void) {
    struct {
        char nome[100];
        char cpf[15];
        int ano_nascimento;
    } alguem;

    strncpy(alguem.nome, "Muriel Gomes Faruak", sizeof alguem.nome - 1);
    strncpy(alguem.cpf, "123.456.789-00", sizeof alguem.cpf - 1);
    alguem.ano_nascimento = 2008;

    printf("Nome: %s,\nCPF: %s,\nAno: %d.\n", alguem.nome, alguem.cpf,
           alguem.ano_nascimento);

    return 0;
}
```

```
Nome: Muriel Gomes Faruak,
CPF: 123.456.789-00,
Ano: 2008.
```

Nesse exemplo, o registro é criado e a variável é declarada de uma única vez. Um nome para o registro, dado que a variável já está criada, é irrelevante e pode ser omitido.

23.4.2 Declarações "iguais"

Em C, ao se encontrar a definição do registro como um tipo, o compilador adiciona esse novo `struct` a uma tabela interna de novos tipos. Se duas definições são idênticas, porém criadas em momentos diferentes, elas não são compatíveis.

Segue um exemplo no qual duas variáveis são criadas sem especificação do nome do `struct` (Seção 23.4.1) para, em seguida, ser declarada uma terceira variável com `struct` idêntico no formato.

```c
/*
 * Exemplo de struct sem nome
 * Assegura: apresentação dos valores iniciados
 */
#include <stdio.h>
#include <string.h>

int main(void) {
    // registro1 e registro2 são compatíveis
    struct {
        int i;
        double d;
    } registro1, registro2;
    registro1.i = 10;
    registro1.d = 1.1;
    registro2 = registro1;  // cópia completa

    // registro3 possui a mesma organização, porém é incompatível
    struct {
        int i;
        double d;
    } registro3;
    registro3 = registro1;

    return 0;
}
```

```
main.c: In function ´main´:
main.c:24:17: error: incompatible types when assigning to type ´struct
<anonymous>´ from type ´struct <anonymous>´
   24 |     registro3 = registro1;
      |                 ^~~~~~~~~
main.c:23:7: warning: variable ´registro3´ set but not used
[-Wunused-but-set-variable]
   23 |     } registro3;
      |       ^~~~~~~~~
main.c:14:18: warning: variable ´registro2´ set but not used
[-Wunused-but-set-variable]
   14 |     } registro1, registro2;
      |                  ^~~~~~~~~
```

O uso de `struct` com nome permite a declaração de novas variáveis referenciando um único tipo e, assim, mantendo a compatibilidade entre as variáveis.

23.5 Exemplos

Mais alguns exemplos com registros são apresentados.

23.5.1 Apresentação de um registro em uma função

O programa seguinte mostra mais um exemplo de passagem de um registro como parâmetro por valor.

```c
/*
 * Apresentação de dados de um vetor 2D com rótulo
 * Assegura: apresentação dos campos do registro
 */
#include <stdio.h>

/*! @struct vetor 2D */
struct vetor {
    char rotulo;
    double x, y;
};

/*!
 * Escreve o conteúdo de um vetor
 * @param vetor: vetor a ser escrito
 */
void escreva_vetor(struct vetor vetor);

/*
 * Main
 */
int main(void) {
    struct vetor vetor1 = {'A', 0.0, 0.0};
    struct vetor vetor2 = {'B', -2.2, 1.7};

    escreva_vetor(vetor1);
    escreva_vetor(vetor2);

    return 0;
}

// Apresenta um vetor na tela
void escreva_vetor(struct vetor vetor) {
    printf("vetor %c: (%.1f, %.1f).\n", vetor.rotulo, vetor.x, vetor.y);
}
```

```
vetor A: (0.0, 0.0).
vetor B: (-2.2, 1.7).
```

É reforçado, aqui, que a declaração de `struct vetor` deve ser global, pois isso é necessário para que o registro definido possa ser usado tanto na função `escreva_vetor` quanto na função `main`.

Dessa forma, a função `escreva_vetor` tem como único parâmetro um `struct vetor`. Em `main`, duas variáveis (`vetor1` e `vetor2`) são declaradas e compartilham o mesmo tipo do parâmetro. Nas chamadas à função, cada um dos registros é passado por valor, com cópia do registro de `main` para o parâmetro `vetor` de `escreva_vetor`.

23.5.2 Normalização de vetores

Um vetor $\vec{v}$ é usualmente representando por um segmento orientado, indicando sua direção e intensidade. A norma de um vetor, também chamada módulo, é representada por $|\vec{v}|$ e corresponde, graficamente, ao comprimento do segmento.

Quando um vetor é normalizado, é mantida sua direção e sentido, alterando-se sua norma para uma unidade, ou seja $|\vec{v'}| = 1$. O programa seguinte ilustra como vetores são normalizados. Cada vetor é organizado em um `struct` e são usadas funções para as diversas manipulações.

```c
/*
 * Apresentação de dados de um vetor 2D com rótulo
 * Assegura: apresentação dos campos do registro
 */
#include <stdio.h>
#include <math.h>

/*! @struct vetor 2D */
struct vetor {
    char rotulo;
    double x, y;
};

/*!
 * Escreve o conteúdo de um vetor
 * @param vetor: vetor a ser escrito
 */
void escreva_vetor(struct vetor vetor);

/*!
 * Faz a normalização de um vetor para que tenha norma igual a 1
 * @param vetor: referência para o vetor
 */
void normalize_vetor(struct vetor *vetor);

/*!
 * Retorna a norma (módulo) de um vetor
 * @param vetor
 * @return norma do vetor
 */
double norma_vetor(struct vetor vetor);

/*
 * Main
 */
int main(void) {
    struct vetor vetor1 = {'A', 3.0, 4.0};
    escreva_vetor(vetor1);
    normalize_vetor(&vetor1);
    escreva_vetor(vetor1);

    struct vetor vetor2 = {'B', 300.0, 400.0};
    escreva_vetor(vetor2);
    normalize_vetor(&vetor2);
    escreva_vetor(vetor2);
```

```c
    struct vetor vetor3 = {'C', -18.7, 41.1};
    escreva_vetor(vetor3);
    normalize_vetor(&vetor3);
    escreva_vetor(vetor3);

    return 0;
}

// Apresenta um vetor na tela
void escreva_vetor(struct vetor vetor) {
    printf("vetor %c: (%.1f, %.1f); norma %.2f.\n", vetor.rotulo, vetor.x,
            vetor.y, norma_vetor(vetor));
}

// Normalização de um vetor
void normalize_vetor(struct vetor *vetor) {
    double norma = norma_vetor(*vetor);

    vetor->x /= norma;
    vetor->y /= norma;
}

// Calcula a norma de um vetor
double norma_vetor(struct vetor vetor) {
    return sqrt(vetor.x * vetor.x + vetor.y * vetor.y);
}
```

```
vetor A: (3.0, 4.0); norma 5.00.
vetor A: (0.6, 0.8); norma 1.00.
vetor B: (300.0, 400.0); norma 500.00.
vetor B: (0.6, 0.8); norma 1.00.
vetor C: (-18.7, 41.1); norma 45.15.
vetor C: (-0.4, 0.9); norma 1.00.
```

As funções `escreva_vetor` e `norma_vetor` recebem um `struct vetor` por valor, com cópia do conteúdo do argumento. A normalização, por sua vez, recebe seu parâmetro por referência e altera o conteúdo do registro apontado.

24 C: Dados em vetores

Os vetores são estruturas de dados que agrupam coleções de itens, todos com o mesmo tipo. Cada item de um vetor é individualizado por seu índice, que é um valor inteiro. Neste capítulo é abordado como vetores são criados em C e quais as peculiaridades que possuem.

24.1 Declaração de vetores em C

Ao se criar uma coleção de itens como um arranjo (outro nome comum para vetor), é preciso especificar o tipo que cada item deve ter e a quantidade de itens.

```
int v[100];  // criação de 100 valores do tipo int
```

Cada item individual pode ser especificado por seu índice, que é um valor inteiro sempre iniciado em zero. Dessa forma, se um vetor `v` possui n itens, seu primeiro item é `v[0]` e seu último, `v[`$n-1$`]`. Os comandos seguintes ilustram como preencher um vetor `v` com dados.

```
int v[100];
for(int i = 0; i < 100; i++)  // de 0 a 99
    v[i] = i;
```

Dica

A atenção aos índices é sempre importante, tanto para se ter acesso ao item desejado quanto não ultrapassar os limites do vetor, atribuindo valores a áreas que não pertencem à variável.
É sempre importante lembrar que C não faz verificações de acesso fora dos índices válidos e cabe, portanto, ao programador garantir que esse acesso não aconteça.

Quando um vetor é criado, um bloco de memória contínuo é criado, de forma que, por exemplo, os endereços do bytes de `v[i]` vêm na sequência de `v[i - 1]`.

Se não houver uma atribuição inicial de valores junto com uma declaração local, os valores contidos no vetor devem ser considerados como não iniciados, ou seja, lixo. Se houver uma declaração global, todos os valores conterão bytes nulos, o que implica em zero para inteiros e reais, `\0` para caracteres, `NULL` para ponteiros, mesmo que o tipo base do vetor seja um registro.

No programa C seguinte, dois vetores de `double` são criados, um global e outro local. Nenhum deles é explicitamente iniciado com valores.

```
/*
 * Programa mostrando a declaração de vetores
 * Assegura: apresentação da quantidade de valores nulos em cada vetor
 */
#include <stdio.h>

double vetor_global[1000];  // criado com todos itens iguais a zero

int main(void) {
    double vetor_local[1000];  // criado sem iniciação (i.e., contém lixo)
```

```
    int contador_de_zeros_global = 0;
    int contador_de_zeros_local = 0;
    for (int i = 0; i < 1000; i++) {
        if (vetor_global[i] == 0)
            contador_de_zeros_global++;
        if (vetor_local[i] == 0)
            contador_de_zeros_local++;
    }
    printf("No vetor global: %d de 1000 valores zero.\n",
           contador_de_zeros_global);
    printf("No vetor local: %d de 1000 valores zero.\n",
           contador_de_zeros_local);

    return 0;
}
```

```
main.c: In function ´main´:
main.c:17:24: warning: ´vetor_local´ may be used uninitialized
[-Wmaybe-uninitialized]
   17 |         if (vetor_local[i] == 0)
      |             ~~~~~~~~~~~^~~
main.c:10:12: note: ´vetor_local´ declared here
   10 |     double vetor_local[1000];  // criado sem iniciação (i.e.,
contém lixo)
      |            ^~~~~~~~~~~
```

```
No vetor global: 1000 de 1000 valores zero.
No vetor local: 534 de 1000 valores zero.
```

É interessante observar que o compilador ajuda o programador com uma mensagem sobre a possibilidade de uso de `vetor_local` com valores não iniciados (o que é, propositalmente, o caso), porém nada é apresentado sobre o vetor global, visto que esse foi, necessariamente, iniciado com zeros.

A saída produzida pelo programa, que o a quantidade de zeros em cada variável, atesta que todos os valores de `vetor_global` são zeros, mas o mesmo não é verdade para `vetor_local`.

Como sempre, o programador nunca deve usar uma variável sem que um valor tenha sido anteriormente atribuído a ela, mesmo que essa "variável" seja uma posição qualquer em um vetor.

24.1.1 Declarações com atribuição explícita

Assim como outras variáveis, é possível iniciar os valores de um vetor juntamente com sua declaração.

Para vetores com itens inteiros ou reais, a iniciação é indicada por uma atribuição com os valores indicados entre chaves. Segue um exemplo simples com a iniciação de um vetor de valores reais.

```
/*
 * Programa mostrando a declaração de vetor com iniciação
 * Assegura: apresentação dos valores do vetor
 */
#include <stdio.h>

int main(void) {
    double vetor[5] = {1.7, 8.2, -6.5, 0.0, 1.2};

    for(int i = 0; i < 5; i++)
        printf("%g ", vetor[i]);
    printf("\n");

    return 0;
}
```

```
1.7 8.2 -6.5 0 1.2
```

Também é possível atribuir apenas às posições iniciais do vetor, como se exemplifica.

```
/*
 * Programa mostrando a declaração de vetor com iniciação
 * Assegura: apresentação dos valores do vetor
 */
#include <stdio.h>

int main(void) {
    double vetor[10] = {-8.23, 0.0, -6.5};

    for(int i = 0; i < 10; i++)
        printf("%g ", vetor[i]);
    printf("\n");

    return 0;
}
```

```
-8.23 0 -6.5 0 0 0 0 0 0 0
```

No programa, apenas são indicados valores para as três primeiras posições de `vetor`, sendo que para as demais nada é explicitamente especificado. Neste caso, todas as posições não especificadas terão valores nulos (ou equivalente, dependendo do tipo base do vetor).

O programa seguinte exemplifica um vetor que tem apenas suas duas posições iniciais com valores explicitamente atribuídos. As demais, uma vez que houve a atribuição, têm seus valores zerados.

```
/*
 * Programa mostrando a declaração de vetores com iniciação parcial
 * Assegura: apresentação da quantidade de zeros no vetor
 */
#include <stdio.h>

int main(void) {
    double vetor[1000] = {1.1, 2.2};  // valores para posições 0 e 1

    int contador_de_zeros = 0;
    for (int i = 0; i < 1000; i++)
        if (vetor[i] == 0)
            contador_de_zeros++;
    printf("No vetor local: %d de 1000 valores zero.\n", contador_de_zeros);

    return 0;
}
```

```
No vetor local: 998 de 1000 valores zero.
```

A execução mostra que, exceto pelos pelas duas posições iniciais do vetor, todas as demais possuem valor nulo.

Essa atribuição é interessante para casos em que se precisa de um vetor com zeros inicialmente em todas suas posições. Assim, uma declaração como a seguinte é suficiente.

```
int v[500000] = {0};  // todo o vetor é iniciado com 500.000 zeros
```

Quando o tipo base do vetor é `char`, as mesmas regras gerais valem. Se um vetor local sem iniciação explícita é criado, seu conteúdo é considerado lixo.

```
char s[300];  // 300 caracteres simples
```

A iniciação do vetor pode ser realizada juntamente com a declaração da variável especificando-se os valores, posição a posição. Como nos casos de inteiros e reais, as primeiras posições recebem valores e as demais ficam com bytes nulos. No caso de cadeias de caracteres, o byte nulo é o caractere `\0`.

```
char s[10] = {'H', 'e', 'l', 'l', 'o'};
```

O programa seguinte comprova esse comportamento.

```
/*
 * Criação de vetor de caracteres com iniciação parcial
 * Assegura: apresentação do conteúdo do vetor, posição a posição
 */
#include <stdio.h>

int main(void) {
    char s[10] = {'H', 'e', 'l', 'l', 'o'};

    for (int i = 0; i < 10; i++)
        printf("s[%d] = '%c' \t(código %d)\n", i, s[i], s[i]);

    return 0;
}
```

```
s[0] = 'H'  (código 72)
s[1] = 'e'  (código 101)
s[2] = 'l'  (código 108)
s[3] = 'l'  (código 108)
s[4] = 'o'  (código 111)
s[5] = ''   (código 0)
s[6] = ''   (código 0)
s[7] = ''   (código 0)
s[8] = ''   (código 0)
s[9] = ''   (código 0)
```

Aqui é importante relembrar a compatibilidade dessa iniciação com a representação de cadeias de caraceteres. Por exemplo, a função printf, usando o formato `%s`, escreva na tela o texto *Hello*, pois interpreta o `\0` de `s[5]` com fim da cadeia.

Há também que se ter uma concordância dos programadores que digitar `'H', 'e', 'l', 'l', 'o'` é uma tarefa, no mínimo, aborrecida. A linguagem aceita a iniciação com os valores entre aspas, usando o conceito dos valores textuais. Dessa forma, a mesma declaração com a mesma iniciação pode ser escrita usando as aspas duplas, com o mesmo resultado.

```
char s[10] = "Hello";
```

A versão que especifica o conteúdo caractere a caractere é útil se o vetor não for usado como uma cadeia de caracteres usual, mas como um vetor de caracteres qualquer. O programa seguinte, variação do anterior, ilustra que o conteúdo do vetor é tratado como vários caracteres e não como uma *string*.

```
/*
 * Criação de vetor de caracteres com iniciação parcial
 * Assegura: apresentação do conteúdo do vetor, posição a posição
 */
#include <stdio.h>

int main(void) {
    char s[10] = {'a', 'e', 'K', '\0', 'G', '\t', 'o', '\a', '\0', 'A'};

    for (int i = 0; i < 10; i++)
        printf("s[%2d] = '%c' \t(código %d)\n", i, s[i], s[i]);

    return 0;
}
```

```
s[ 0] = 'a'   (código 97)
s[ 1] = 'e'   (código 101)
s[ 2] = 'K'   (código 75)
s[ 3] = ''  (código 0)
s[ 4] = 'G'   (código 71)
s[ 5] = ' '   (código 9)
s[ 6] = 'o'   (código 111)
s[ 7] = ''  (código 7)
s[ 8] = ''  (código 0)
s[ 9] = 'A'   (código 65)
```

24.1.2 Declarações com tamanho automático

A linguagem permite uma facilidade ao programador quando um vetor é criado com valores inicialmente atribuídos, que é omitir a quantidade de itens que o vetor possui. Esse tamanho é determinado pelos valores atribuídos. Segue um exemplo.

```c
/*
 * Programa mostrando a declaração de vetores com tamanho automático
 * Assegura: apresentação dos valores do vetor
 */
#include <stdio.h>

int main(void) {
    int vetor[] = {5, 4, 3, 2, 1};  // vetor com 5 itens

    for (int i = 0; i < 5; i++)
        printf("%d ", vetor[i]);
    printf("\n");

    return 0;
}
```

```
5 4 3 2 1
```

A variável `vetor` é criada com exatamente cinco itens, já que há especificação de cinco valores na iniciação. A declaração feita no programa é exatamente equivalente à declaração seguinte.

```c
int vetor[5] = {5, 4, 3, 2, 1};
```

A vantagem dessa omissão do tamanho é facilitar a escrita do código, sem que o programador tenha que contar quantos valores estão sendo usados na iniciação para colocá-lo dentro dos colchetes.

Embora esse recurso de escrita facilite a declaração, ele não se estende ao resto do código. Por exemplo, no programa exemplo o `for` usou a condição `i < 5`, o que significa que o programador tem que saber quantos valores há efetivamente no vetor. Para esses casos, um truque de programação pode ser usado e ele é exemplificado no programa seguinte.

```c
/*
 * Programa mostrando a declaração de vetores com tamanho automático
 * Assegura: apresentação do tamanho de vetor em bytes, do tamanho do
 *  tipo base em bytes e a quantidade de itens no vetor
 */
#include <stdio.h>

int main(void) {
    int vetor[] = {18, -2, 0, 3, 1, 7, 22, 13, 255, 1, 0, 0, 3};
    printf("O vetor de int possui %zu bytes.\n", sizeof vetor);
    printf("Um único int possui %zu bytes.\n", sizeof(int));
    printf("Portanto, o vetor possui %zu/%zu = %zu itens!\n",
           sizeof vetor, sizeof(int), sizeof vetor / sizeof(int));

    return 0;
}
```

```
O vetor de int possui 52 bytes.
Um único int possui 4 bytes.
Portanto, o vetor possui 52/4 = 13 itens!
```

Sabendo-se o tamanho total em bytes do vetor (`sizeof vetor`) e o tamanho em bytes de cada um de seus itens (`sizeof (int)`), é possível deduzir quantos itens há no vetor. Segue um novo programa exemplo que usa essa estratégia.

```c
/*
 * Apresentação das unidades monetárias da moeda brasileira
 * Assegura: apresentação de cada valor de cédula ou moeda (reais ou
 *  centavos)
 */
#include <stdio.h>

int main(void) {
    printf("Moeda brasileira\n"
           "----------------\n\n"
           "Notas:\n");
    double valores_notas[] = {200.00, 100.00, 50.00, 20.00, 10.00, 5.00, 2.00};
    for (int i = 0; i < (int)(sizeof valores_notas / sizeof(double)); i++)
        printf("R$ %6.2f\n", valores_notas[i]);

    printf("\nMoedas:\n");
    double valores_moedas[] = {1.00, 0.50, 0.25, 0.10, 0.05, 0.01};
    for (int i = 0; i < (int)(sizeof valores_moedas / sizeof(double)); i++)
        printf("R$ %.2f\n", valores_moedas[i]);

    return 0;
}
```

```
Moeda brasileira
----------------

Notas:
R$ 200.00
R$ 100.00
R$  50.00
R$  20.00
R$  10.00
R$   5.00
R$   2.00

Moedas:
R$ 1.00
R$ 0.50
R$ 0.25
R$ 0.10
R$ 0.05
R$ 0.01
```

24.1.3 Vetores com tamanho sob demanda

O número de itens de um vetor pode ser especificado de duas formas básicas: com a especificação explícita do tamanho nos colchetes ou fazendo a iniciação com o número desejado de itens. Uma possibilidade adicional, relativa ao primeiro formato, é o uso de uma expressão para indicar a quantidade.

```c
int n = 10;
double v1[n];  // vetor com 10 itens
double v2[2 * n];  // vetor com 20 itens
```

Na prática, a quantidade de itens usada para dimensionar o vetor pode depender de um cálculo ou uma informação em tempo de execução.

O programa seguinte mostra vetores criados com tamanhos aleatórios, usando a função `rand` de `stdlib.h`.

```
/*
 * Criação de vetores com diferentes quantidades de itens
 * Assegura: apresentação do número de itens de cada vetor
 */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    for (int exemplo = 1; exemplo <= 10; exemplo++) {
        // rand() % 20 + 1 resulta em um valor aleatório de 1 até 20
        double vetor[rand() % 20 + 1];
        printf("Exemplo %d: vetor criado com %zu itens.\n", exemplo,
               sizeof vetor/sizeof (double));
    }

    return 0;
}
```

```
Exemplo 1: vetor criado com 4 itens.
Exemplo 2: vetor criado com 7 itens.
Exemplo 3: vetor criado com 18 itens.
Exemplo 4: vetor criado com 16 itens.
Exemplo 5: vetor criado com 14 itens.
Exemplo 6: vetor criado com 16 itens.
Exemplo 7: vetor criado com 7 itens.
Exemplo 8: vetor criado com 13 itens.
Exemplo 9: vetor criado com 10 itens.
Exemplo 10: vetor criado com 2 itens.
```

A cada repetição do `for` é criado um vetor com um tamanho aleatório, podendo ter de 1 a 20 itens. A quantidade de itens é apresentada verificando o tamanho real do vetor.

O uso desse recurso é interessante, por exemplo, quando o usuário fornece a quantidade de itens inicialmente, seguida dos valores de cada dado. O vetor pode ser criado exatamente do tamanho para armazenar os dados esperados.

24.2 Vetores de `struct`

O tipo base de um vetore pode ser um registro (`struct`), de modo que uma coleção de registros pode ser facilmente organizada.

Como exemplo, pode-se considerar um registro contendo as três coordenadas de um ponto em $\mathbb{R}^3$.

```
struct ponto3d {
    double x, y, z;
};

struct ponto3d lista_pontos[200];  // 200 registros
```

Cada elemento do vetor `lista_pontos` é um registro. Assim, `lista_pontos[0]` é o primeiro registro do vetor, o que leva a `lista_pontos[0].x`, `lista_pontos[0].y` e `lista_pontos[0].z` serem a especificação de cada um dos seus campos.

24.3 Ponteiros para vetores

Em C, quando o identificador de um vetor é usado, o compilador associa a ele o endereço dessa variável na memória.

```
int i;
int vet[10];

// &i é o endereço de i
// vet é o endereço de vet
```

Referências

ABELSON, H.; SUSSMAN, G. J. *Structure and interpretation of computer programs*. The MIT Press, 1996. Citation Key: abelson1996structure.

JOHNSON, S. C.; KERNIGHAN, B. W. *The programming language B*. Bell Laboratories Murray Hill, New Jersey, 1973. https://minnie.tuhs.org/Mirrors/Dennis/btut.pdf.

RICHARDS, M. BCPL: A tool for compiler writing and system programming. In: 557–566. https://dl.acm.org/doi/pdf/10.1145/1476793.1476880.

RITCHIE, D. M. The development of the C language. *ACM Sigplan Notices*, v. 28, n. 3, 201–208, 1993. https://www.bell-labs.com/usr/dmr/www/chist.pdf.

RITCHIE, D. M. et al. The C programming language. *Bell Sys. Tech. J*, v. 57, n. 6, 1991–2019, 1978. https://www.academia.edu/download/67840358/1978.07_Bell_System_Technical_Journal.pdf#page=85.

WIKIPEDIA. C (programming language), 2023. https://en.wikipedia.org/w/index.php?title=C_(programming_language)&oldid=1182945224. Acesso em: 2 dez. 2023.

Índice Remissivo

algoritmo, 2
algoritmos
  computacionais, 3
arquivo
  físico, 126
  lógico, 126
atribuição, 28

`char`, 18
comando
  simples, 11

declaração, 27
`do while`, 118
documentação, 87

`feof`, 136
`for`, 111
`fprintf`, 123, **132**

`getline`, 40
`gets`, 40

identificador, 23

overflow, 51

pré-condição, 1
pós-condição, 1

`strtol`, 40

transbordo, 51

Unicode, 150
UTF-8, 150

variável, 22

`while`, 102