Programação de Computadores em C

Primeira edição

Carlos Camarão
Universidade Federal de Minas Gerais
Doutor em Ciência da Computação pela Universidade de Manchester, Inglaterra

Anolan Milanés
Universidade Federal de Minas Gerais
Doutora em Ciência da Computação pela PUC-Rio

Lucília Figueiredo
Universidade Federal de Ouro Preto
Doutora em Ciência da Computação pela UFMG

Sumário

Prefácio

Este livro se propõe a acompanhá-lo no início de um longo caminho, que é o do desenvolvimento do raciocínio necessário para construção de programas bem feitos. Para isto, o livro aborda conceitos básicos de programação de computadores, de forma que posteriormente assuntos mais avançados possam ser abordados.

Conteúdo e Organização do Livro

Este livro foi concebido para ser utilizado como texto didático em cursos introdutórios de programação, de nível universitário. O conteúdo do livro não pressupõe qualquer conhecimento ou experiência prévia do leitor em programação, ou na área de computação em geral, requerendo apenas conhecimentos básicos de matemática, usualmente abordados nos cursos de primeiro e segundo graus. O livro adota a linguagem de programação `C` (veja no anexo A uma discussão sobre essa escolha).

Como estudaremos daqui a pouco, a linguagem C é uma linguagem imperativa. Uma visão geral sobre os conceitos básicos da programação imperativa é apresentada inicialmente, com o objetivo de favorecer uma compreensão global sobre o significado e o propósito dos conceitos empregados nesse estilo de programação, facilitando assim um melhor entendimento da aplicação dos mesmos na construção de programas. Cada um desses conceitos é abordado mais detalhadamante em capítulos subsequentes, por meio de exemplos ilustrativos e exercícios.

Além desses conceitos básicos, o livro aborda também, de maneira introdutória, os seguintes tópicos adicionais: entrada e saída de dados em arquivos, e manipulação de estruturas de dados como arranjos e listas encadeadas.

Recursos Adicionais

Uma página na Internet associada a este livro pode ser encontrada no endereço:

`http://www.dcc.ufmg.br/~camarao/ipcc`

O texto desse livro e os códigos da maioria dos programas apresentados no livro como exemplos encontram-se disponíveis nesta página. Outros recursos disponíveis incluem sugestões de exercícios adicionais e projetos de programação, transparências para uso em cursos baseados neste livro e referências para outras páginas da Internet que contêm informações sobre a linguagem `C` ou sobre ferramentas para programação nessa linguagem.

Há diversas páginas na Web com informações sobre a linguagem `C` disponíveis na Web, assim como cursos sobre introdução a prgramação em `C`, dentre os quais citamos:

- `http://pt.wikibooks.org/wiki/Programar_em_C`: Páginas Wiki, criadas pela própria comunidade de usuários da Web, sobre programação em `C`. A versão em inglês pode ser encontrada em `http://en.wikipedia.org/wiki/C_(programming_language)`.

- `http://www.ead.cpdee.ufmg.br/cursos/C`: Material usado no curso de introdução a programação em `C` ministrado no Departamento de Engenharia Elétrica da UFMG.

- `http://cm.bell-labs.com/cm/cs/cbook/`: *The C Programming Language*, B. Kernighan & Dennis M. Ritchie, Prentice Hall, 1988.

- http://publications.gbdirect.co.uk/c_book/: Página com a versão gratuita da segunda edição do livro *The C Book*, de Mike Banahan, Declan Brady e Mark Doran, publicado pela Addison Wesley em 1991.
- http://www.cyberdiem.com/vin/learn.html: *Learn C/C++ today*, de V. Carpenter. Uma coleção de referências e tutoriais sobre as linguagens C e C++ disponíveis na Internet.
- http://c-faq.com/index.html: Perguntas frequentes sobre a linguagem `C`, e suas respostas (em inglês).
- http://cm.bell-labs.com/cm/cs/who/dmr/chist.html: "The Development of the C Language", Dennis M. Ritchie (Janeiro de 1993).
- http://www.livinginternet.com/i/iw_unix_c.htm: "History of the C Programming Language", Bill Stewart (Janeiro de 2000).
- http://www.cs.ucr.edu/ nxiao/cs10/errors.htm: "10 Common Programming Mistakes in C".

Livros adicionais sobre a linguagem `C` incluem:

- *A linguagem de programação padrão ANSI C*. B. Kernighan & D.C. Ritchie. Editora Campus, 1990.
- *C — completo e total*. H. Schildt. Editora McGraw-Hill, 1990.a
- *C: A Reference Manual*, Samuel P. Harbison & Guy L. Steele, 5ª edição, Prentice Hall, 2002.
- *C Programming: A Modern Approach*, K.N. King, Norton, 2008.

E divirta-se assistindo:

http://www.youtube.com/watch?v=XHosLhPEN3k

Capítulo 1

Computadores e Programas

1.1 Computadores e Algoritmos

Computadores são empregados atualmente nas mais diversas atividades, como edição e composição de textos, sons e imagens, mecanização e automatização de diversas tarefas, simulação dos mais variados fenômenos e transferência de grandes volumes de informação entre locais possivelmente muito distantes. Seu uso em atividades científicas permitiu transpor inúmeras fronteiras, assim como criar novas e desafiantes áreas de investigação.

Como se pode explicar tão grande impacto? O que diferencia um computador de uma ferramenta qualquer?

De fato, o conjunto das operações básicas executadas por um computador é tão simples quanto o de uma pequena calculadora. A grande distinção de um computador, responsável por sua enorme versatilidade, está no fato dele ser programável. Computadores podem realizar tarefas complexas pela combinação de suas operações básicas simples. Mas, como isso é possível? Como podemos descrever para o computador a tarefa que queremos que ele execute?

Para que um computador possa executar qualquer tarefa, é preciso que lhe seja fornecida uma descrição, em linguagem apropriada, de como essa tarefa deve ser realizada. Tal descrição é chamada de *programa* e a linguagem usada para essa descrição, de *linguagem de programação*. A idéia ou processo que esse programa representa é chamada de *algoritmo*.

1.2 Algoritmo e Programa

Algoritmo e programa não são noções peculiares à computação. Um *algoritmo* consiste simplesmente em uma descrição finita de como realizar uma tarefa ou resolver um problema. Essa descrição deve ser composta de operações executáveis. Cozinhar, montar móveis ou briquedos, realizar cálculos matemáticos ou tocar um instrumento musical, são exemplos de tarefas que podem ser executadas usando um algoritmo. Receitas de cozinha, instruções de montagem, regras para realização de cálculos e partituras musicais são exemplos de programas (representações de algoritmos) para realizar essas tarefas.

A distinção entre algoritmo e programa é similar à distinção existente, por exemplo, entre número e numeral. Essa distinção muitas vezes não se faz perceber, por se tornar dispensável. Por exemplo, não escrevemos "número representado por 7" ou "número denotado por 7", mas simplesmente "número 7". É útil saber, no entanto, que podem existir diversas denotações para um mesmo algoritmo, assim como um número pode ser escrito de diferentes maneiras, tais como:

`7   VII   sete   seven` |||||||

Embora seja capaz de executar apenas um pequeno número de operações básicas bastante simples, um computador pode ser usado, em princípio, para resolver qualquer problema cuja solução possa ser obtida por meio de um algoritmo. O segredo é que um computador provê também um conjunto de instruções para a combinação dessas operações que, embora também reduzido, é suficiente para expressar qualquer algoritmo. Essa tese, de que o conjunto de operações básicas e instruções de um computador é suficiente para expressar qualquer algoritmo, constitui uma

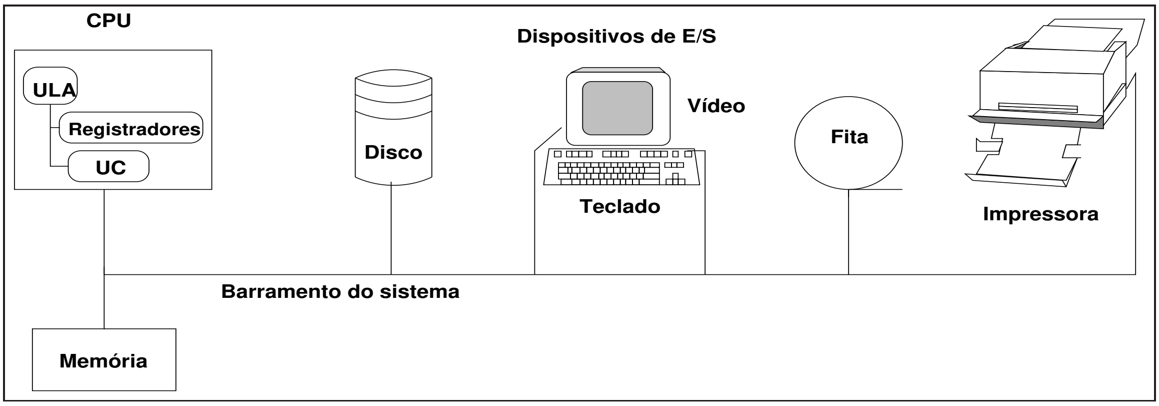

Figura 1.1: Organização básica de um computador

tese resultante de trabalhos com modelos computacionais distintos usados nas primeiras pesquisas teóricas em computação, que se mostraram equivalentes, em termos de o que se poderia computar com eles. Essas pesquisas lançaram as bases para a ciência da computação e para a construção dos primeiros computadores.[1]

Como construir algoritmos para a solução de problemas e como expressar tais algoritmos de modo adequado usando uma linguagem de programação constituem os temas centrais deste livro. Entretanto, faremos agora um pequeno preâmbulo para estudar brevemente as bases da organização e o funcionamento de um sistema de computação. Esse conhecimento nos permitirá entender como o computador executa os nossos programas.

1.3 Funcionamento e Organização de Computadores

A atual tecnologia de construção de computadores é baseada em dispositivos eletrônicos que são capazes de distinguir, com precisão, entre dois estados diferentes de um sinal elétrico, caracterizados comumente pelos símbolos `0` e `1`. Devido a essa característica, dados e operações são representados, em um computador, em uma linguagem que tem apenas esses dois símbolos, isto é, uma linguagem *binária*. Cada um desses símbolos é comumente chamado de *bit*.[2]

Apesar do extraordinário avanço da atual tecnologia de construção de computadores, todo computador moderno mantém uma organização básica semelhante. Essa organização é mostrada na Figura 1.1 e seus componentes principais são descritos brevemente a seguir.

O *processador*, também chamado de *unidade central de processamento* (em inglês, CPU — *Central Processing Unit*), é o componente do computador que executa as instruções de um programa, expresso em uma linguagem que ele pode entender. Durante a execução de um programa, o processador "lê" a instrução corrente, executa a operação especificada nessa instrução e determina qual é a próxima instrução do programa que deve ser executada. Isso se repete até a execução de uma instrução que indica o término do programa.

Muitas dessas instruções precisam de um lugar de onde tirar os operandos e onde armazenar temporariamente os resultados. Operandos e resultados são chamados em computação de *valores*, ou *dados*: são sequencias de bits que podem representar um número, ou um caractere, ou outro valor qualquer. O lugar que armazena um operando de uma instrução em um computador é chamado de *registrador*. Podemos ver um registrador como uma caixinha, cada uma com um nome distinto, que armazena apenas um valor de cada vez.

Outro componente do computador é a *memória principal*, em geral chamada simplesmente de *memória*, ou *RAM* (do inglês *Random Access Memory*). A memória é usada para armazenar os programas a serem executados pelo computador e os dados manipulados por esses programas. Essa característica de utilizar um único dispositivo de memória para armazenar tanto programas quanto dados, peculiar a todos os computadores modernos, é distintiva da chamada *arquitetura de von*

[1] Veja as Notas Bibliográficas incluídas no final deste capítulo.
[2] Do inglês *binary digit*.

Neumann, assim denominada em homenagem ao pesquisador alemão que originalmente publicou artigo sobre essa arquitetura, em 1946.

A memória do computador consiste em uma seqüência finita de unidades de armazenamento de dados, cada qual identificada pelo seu *endereço*, isto é, por um número inteiro não-negativo que corresponde à sua posição nessa seqüência. Cada unidade de armazenamento de dados da memória é comumente chamada de uma *palavra*. Uma palavra de memória usualmente é composta de um pequeno número de *bytes* (em geral, 4 ou 8). Cada byte armazena uma sequência de 8 bits.

O outro grupo de componentes do computador é constituído pelos seus *dispositivos de entrada e saída*, também chamados de *dispositivos periféricos* (ou apenas de *periféricos*). Os periféricos são usados para a comunicação de dados entre o computador e o mundo externo. O teclado e o *mouse* são exemplos de dispositivos de entrada. A tela, ou monitor, e a impressora são exemplos de dispositivos de saída. Alguns dispositivos, como discos e *pendrives*, constituem dispositivos tanto de entrada quanto de saída de dados.

A linguagem constituída pelas instruções que podem ser diretamente executadas por um computador, representadas na forma de sequências de bits, é chamada de *linguagem de máquina*.

1.3.1 Linguagem de máquina

A linguagem de máquina de um computador consiste das instruções que seu processador é capaz de executar. Elas são instruções para realizar a execução de operações básicas, tais como somar dois números ou comparar se dois números são iguais, transferir dados entre a memória e um registrador, ou entre a memória e um dispositivo de entrada e saída, desviar a execução de um programa para uma instrução armazenada em um dado endereço. Um processador é capaz de processar sequências de bits, e portanto um programa escrito em linguagem de máquina deve ser uma sequência de bits.

A característica de linguagens de máquina que faz com que um processador seja capaz de executar instruções de programas escritos nessas linguagens é que existe um código único para cada instrução que determina o tamanho dos operandos e do resultado da operação. Ao ler e decodificar o código de cada instrução, o processador pode iniciar a leitura de cada operando e comandar a execução da operação necessária, e armazenar o resultado no lugar designado pelo código.

Vejamos um exemplo. Vamos considerar um fragmento de uma linguagem de máquina que contém, dentre o conjunto de instruções dessa linguagem, as seguintes instruções:

- Copiar valor para registrador; em computação, é também dito "carregar" (do inglês, *load*) valor em registrador. O valor é um operando armazenado diretamente na instrução, assim como o número do registrador.
- Copiar valor em memória para registrador. O endereço da memória no qual o valor está contido é um operando armazenado na instrução, assim como o número do registrador.
- Somar valores em dois registradores e armazenar resultado em registrador. Os números dos registradores são armazenados na instrução. O registrador que recebe o resultado é o primeiro dos dois registradores especificados como operando.
- Desviar execução se o resultado obtido pela execução da instrução anterior for maior ou igual a zero. O endereço da instrução para a qual a execução será desviada é armazenado na instrução.

Vamos escrever um programa para somar dois números, sendo que um deles é igual a `5` e o outro está guardado, previamente, na memória do computador no endereço `BBEh` (`3006` em decimal). Suponhamos que, dentre o conjunto de registradores da nossa máquina, há dois registradores `R1` e `R2`, que correspondem aos códigos `00` e `01` respectivamente. Nosso programa irá, um pouco mais detalhadamente:

- `CARREGAR` `5` no registrador `R1` (ou seja: copiar o valor `5` para `R1`);
- `CARREGAR` o valor armazenado no endereço `3006` da memória para o registrador `R2`;
- Somar os valores contidos nos registradores `R1` e `R2` e armazenar o resultado em `R1`;

- Desviar a execução para a instrução armazenada em certo endereço da memória se o resultado obtido pela execução da instrução anterior for maior ou igual a zero.

O código de cada uma dessas instruções é mostrado, na segunda coluna, a seguir.

Operação	**Código**
CARREGAR valor inteiro em registrador	0000
CARREGAR valor em memória em registrador	0001
SOMAR valor em registrador com valor em registrador e guardar resultado no primeiro registrador	0010
DESVIAR execução se resultado de instrução anterior for maior ou igual a zero	0011

O trecho de programa é mostrado, na terceira coluna, a seguir.

Localização	Operação	Sequência de bits
0x00e3	`CARREGAR` #5 em `R1`	0000 000000000101 000000000000
0x00e4	`CARREGAR` em `R2` valor armazenado no endereço `BBEh` (3006)	0001 101110111110 000000000001
`0x00e5`	`SOMAR` `R1` com `R2` e guardar resultado em `R1`	0010 000000000000 000000000001
`0x00e6`	`DESVIAR` para endereço `0x00e5` se resultado da instrução anterior for maior ou igual a zero	0011 000011100011 000000000000

Essa sequência de bits poderia ser um pequeno trecho de um programa escrito em linguagem de máquina. Como já mencionamos, tais programas consistem em instruções muito básicas, tais como somar dois números, ou comparar se dois números são iguais, ou transferir dados entre a memória e um registrador, ou entre a memória e um dispositivo de entrada e saída, e controlar o fluxo de execução das instruções de um programa.

Na época em que foram construídos os primeiros computadores, os programadores usavam instruções como essas, cada qual formada por aproximadamente uma dezena de bits, para descrever seus algoritmos. Programar nessa linguagem era uma tarefa trabalhosa e extremamente sujeita a erros, difíceis de detectar e corrigir, que geravam programas difíceis de serem estendidos e modificados. Por esses motivos, os programadores logo passaram a usar nomes para as operações e dados, como mostrado na seção seguinte.

1.3.2 Linguagem de montagem

As instruções mostradas em linguagem de máquina na seção anterior como a seguir. Supomos que `x` é o nome de um lugar ou posição da área de dados, correspondente ao endereço `BBEh`, e que `L` é um lugar ou posição da área de instruções do programa, correspondente ao endereço `0x00e5`.

```
MOV R1, #5
MOV R2, x
ADD R1, R2
JGE L
```

Um programa escrito nessa forma era então manualmente traduzido para linguagem de máquina, e depois carregado na memória do computador para ser executado.

O passo seguinte foi transferir para o próprio computador essa tarefa de *"montar"* o programa em linguagem de máquina, desenvolvendo um programa para realizar essa tarefa. Esse programa é chamado de *montador*[3], e a notação simbólica dos programas que ele traduz é chamada de *linguagem de montagem*.[4]

[3]Em inglês, *assembler*.
[4]Em inglês, *assembly language*.

1.3.3 Linguagem de alto nível, compilação e interpretação

Desenvolver programas em linguagem de montagem continuava sendo, entretanto, uma tarefa difícil e sujeita a grande quantidade de erros, que geravam programas difíceis de serem estendidos e modificados. A razão é que as instruções dessa linguagem, exatamente as mesmas da linguagem de máquina, têm pouca relação com as abstrações usualmente empregadas pelo programador na construção de algoritmos para a solução de problemas.

Para facilitar a tarefa de programação, e torná-la mais produtiva, foram então desenvolvidas novas linguagens de programação. Essas novas linguagens foram chamadas *linguagens de alto nível*, por oferecer um conjunto muito mais rico de operações e construções sintáticas adequadas para expressar, de maneira mais natural, algoritmos usados na solução de problemas. Linguagens de máquina e linguagens de montagem são chamadas, em contraposição, *linguagens de baixo nível*.

Para que um programa escrito em uma linguagem de alto nível possa ser executado pelo computador, ele precisa ser primeiro traduzido para um programa equivalente em linguagem de máquina. Esse processo de tradução é chamado de *compilação*; o programa que faz essa tradução é chamado de *compilador*. Um compilador é, portanto, simplesmente um programa tradutor, de programas escritos em uma determinada linguagem, chamada de *linguagem fonte*, para programas em outra linguagem, chamada de *linguagem objeto*. Os programas fornecidos como entrada e obtidos como saída de um compilador são também comumente chamados, respectivamente, de *programa fonte* (ou *código fonte*) e *programa objeto* (ou *código objeto*).

Um compilador analisa o texto de um programa fonte para determinar se ele está sintaticamente correto, isto é, em conformidade com as regras da gramática da linguagem e, em caso afirmativo, gera um código objeto equivalente. Caso o programa fonte contenha algum erro, o compilador então emite mensagens que auxiliam o programador na identificação e correção dos erros existentes.

Outro processo para execução de um programa em linguagem de alto nível, em vez da compilação desse programa seguida pela execução do código objeto correspondente, é a *interpretação* do programa fonte diretamente. Um *interpretador* é, como o nome indica, um programa que interpreta diretamente as frases do programa fonte, isto é, simula a execução dos comandos desse programa sobre um conjunto de dados, também fornecidos como entrada para o interpretador. A interpretação de programas escritos em uma determinada linguagem define uma "máquina virtual", na qual é realizada a execução de instruções dessa linguagem.

A interpretação de um programa em linguagem de alto nível pode ser centenas de vezes mais lenta do que a execução do código objeto gerado para esse programa pelo compilador. A razão disso é que o processo de interpretação envolve simultaneamente a análise e simulação da execução de cada instrução do programa, ao passo que essa análise é feita previamente, durante a compilação, no segundo caso. Apesar de ser menos eficiente, o uso de interpretadores muitas vezes é útil, principalmente devido ao fato de que, em geral, é mais fácil desenvolver um interpretador do que um compilador para uma determinada linguagem.

Esse aspecto foi explorado pelos projetistas de linguagens tais como Java, no desenvolvimento de sistemas (ou ambientes) para programação e execução de programas nessa linguagem: esses ambientes são baseados em uma combinação dos processos de compilação e interpretação. Um ambiente de programação Java é constituído de um compilador Java, que gera um código de mais baixo nível, chamado de *bytecodes*, que é então interpretado. Um interpretador de *bytecodes* interpreta instruções da chamada "Máquina Virtual Java" (Em inglês JVM – Java Virtual Machine) Esse esquema usado no ambiente de programação Java não apenas contribuiu para facilitar a implementação da linguagem em grande número de computadores diferentes, mas constitui uma característica essencial no desenvolvimento de aplicações voltadas para a Internet, pois possibilita que um programa compilado em um determinado computador possa ser transferido através da rede e executado em qualquer outro computador que disponha de um interpretador de *bytecodes*. Outras linguagens interpretadas muito conhecidas são *Phyton* e *Lua*. Lua é uma linguagem de programação projetada na PUC-Rio e desenvolvida principalmente no Brasil, formando atualmente parte do Ginga, o padrão brasileiro de televisão digital.

Ambientes de Programação

Além de compiladores e interpretadores, um ambiente de programação de uma determinada linguagem de alto nível oferece, em geral, um conjunto de bibliotecas de componentes ou módulos de programas, usados comumente no desenvolvimento de programas para diversas aplicações. Além disso, ambientes de programação incluem outras ferramentas para uso no desenvolvimento de programas, como editores de texto e depuradores de programas.

Um *editor* é um programa usado para criar um arquivo de dados e modificar ou armazenar dados nesse arquivo. O tipo mais simples de editor é um *editor de texto*, que permite "editar" (i.e. criar ou modificar) qualquer documento textual (um "texto" significando uma seqüência de caracteres, separados linha por linha). Alguns editores usam caracteres especiais, chamados de caracteres de controle, para facilitar a visualização do texto editado, por exemplo colocando em destaque (em negrito ou com uma cor diferente) palavras-chave da linguagem.

Um *depurador* é um programa que oferece funções específicas para acompanhamento da execução de um programa, com o objetivo de auxiliar o programador na detecção e identificação da origem de erros que possam existir em um programa.

Em um ambiente de programação convencional, editores, compiladores, interpretadores e depuradores são programas independentes. Em um *ambiente integrado* de programação, ao contrário, as tarefas de edição, compilação, interpretação e depuração são oferecidas como opções disponíveis em um mesmo programa.

A execução de programas em um computador é iniciada e controlada por um programa denominado *sistema operacional*. O sistema operacional controla a operação em conjunto dos diversos componentes do computador — processador, memória e dispositivos de entrada e saída — assim como a execução simultânea de diversos programas pelo computador. A execução do núcleo do sistema operacional é iniciada no momento em que o computador é ligado, quando esse núcleo é transferido do disco para a memória do computador, permanecendo residente na memória enquanto o computador estiver ligado. O núcleo do sistema operacional provê uma interface adequada entre a máquina e os demais programas do sistema operacional que, por sua vez, oferecem uma interface adequada entre os diversos componentes do computador e os usuários e seus programas, em um ambiente de programação.

1.4 Exercícios Resolvidos

1. Mencionamos que os números são representados no computador usando a notação arábica, no sistema de numeração de base 2, ou sistema de numeração binário. Este exercício aborda a representação de números usando essa notação e a conversão entre as representações de números nos sistemas de numeração binário e decimal.

A *notação hindu-arábica*, que usamos para escrever números em nosso sistema de numeração decimal, teria sido originada na Índia, no terceiro século a.C., sendo mais tarde levada para Bagdá, no oitavo século d.C. É interessante observar que o símbolo que representa o número zero só apareceu em um estágio posterior do desenvolvimento dessa notação, no século nove d.C. O nome *notação arábica*, mais comumente usado, se deve ao fato de que essa notação foi divulgada pela primeira vez por um matemático árabe, chamado al-Khuarizmi — daí o nome algarismo, dado aos símbolos que usamos atualmente para a representação de números no nosso sistema de numeração.

A característica fundamental da notação hindu-arábica, que torna mais fácil representar números grandes e realizar operações sobre números, é o fato de ela ser uma *notação posicional*. No nosso sistema de numeração, de base 10, a posição de cada algarismo determina as potências de 10 pelas quais devem ser multiplicados os números denotados por esses algarismos, para obter o número representado.

Por exemplo:

$$496 = 4 \times 10^2 + 9 \times 10^1 + 6 \times 10^0$$

Essa notação pode ser usada, de modo geral, para representação de números em um sistema de numeração de base b qualquer, onde b é um número inteiro positivo, com base no fato de que qualquer número inteiro não-negativo p pode ser univocamente representado na forma

$$p = \sum_{i=0}^{n} d_i \times b^i$$

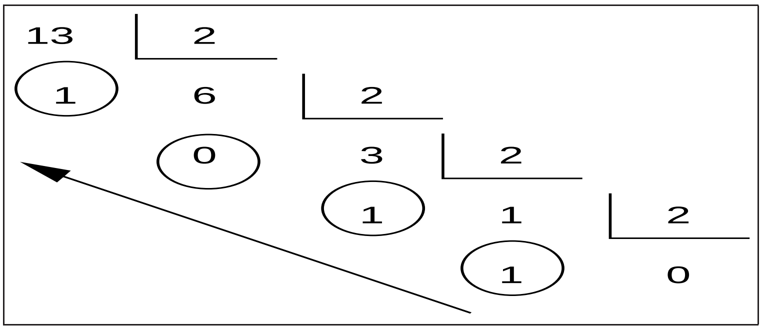

Figura 1.2: Conversão de representação de número, de decimal para binária

onde cada d_i, para $i = 0, \ldots, n$, é um símbolo que representa um número de 0 a $b - 1$.

No sistema de numeração binário (de base 2), o numeral 1011, por exemplo, representa o número decimal 11, conforme se mostra a seguir (um subscrito é usado para indicar a base do sistema de numeração em cada caso):

$$1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11_{10}$$

Exercício: Converta o número 110101_2 para a sua representação no sistema decimal.

Para converter um número p, escrito na base 10, para a sua representação na base 2, basta notar que, se $p = \sum_{i=0}^{n} d_i \times 2^i$, onde $d_i = 0$ ou $d_i = 1$, para $i = 0, \ldots, n$, e $d_n \neq 0$, temos que $2^{n+1} < p \leq 2^n$. Portanto, efetuando n divisões sucessivas de p por 2, obtemos:

$$\begin{array}{rcl}
p & = & 2q_0 + d_0 \\
 & = & 2(2q_1 + d_1) + d_0 = 2^2 q_1 + 2d_1 + d_0 \\
\vdots & & \\
 & = & 2(\ldots 2((2q_n + d_n) + d_{n-1}) \ldots + d_1) + d_0 \\
 & = & 2^{n+1} q_n + 2^n d_n + 2^{n-1} d_{n-1} + \ldots + 2d_1 + d_0 \\
 & = & 2^n d_n + 2^{n-1} d_{n-1} + \ldots + 2d_1 + d_0
\end{array}$$

uma vez que teremos $q_n = 0$.

O processo de conversão de um número da sua representação decimal para a sua representação binária, pode ser feito, portanto, como mostra a Figura 1.2, onde se ilustra essa conversão para o número decimal 13.

Exercício: converta o número 295_{10} para a sua representação na base binária.

2. Uma das operações básicas que um computador é capaz de realizar é a operação de somar dois números inteiros. Como essa operação é executada em um computador?

Os componentes básicos dos circuitos eletrônicos de um computador moderno são chamados de portas lógicas. Uma *porta lógica* é simplesmente um circuito eletrônico que produz um sinal de saída, representado como `1` ou `0` e interpretado como "verdadeiro" (`V`) ou "falso" (`F`), respectivamente, que é o resultado de uma operação lógica sobre os seus sinais de entrada.

Essas operações lógicas — "não", "ê", "ou", "ou exclusivo", representadas pelos símbolos (ou conectivos lógicos) $\neg$, $\wedge$, $\vee$, $\oplus$, respectivamente — são definidas na Tabela 1.1.

O conjunto constituído dos valores "verdadeiro" e "falso" é chamado de conjunto *Booleano*, em homenagem ao matemático George Boole (1815-1864), um dos pioneiros na formalização da lógica matemática. Analogamente, os valores desse conjunto são chamados de *valores booleanos* e as operações lógicas definidas sobre esse conjunto são chamadas de *operações booleanas*, ou operações da *Lógica Booleana* (ou Lógica Proposicional).

A operação lógica "ê" tem resultado verdadeiro se ambos os operandos são verdadeiros, e falso em caso contrário. A operação lógica "ou" tem resultado falso se ambos os operando

Tabela 1.1: As operações lógicas "não", "ê", "ou" e "ou exclusivo"

Operação "não"	**Resultado**
¬V	F
¬F	V

Operação	**Resultado**		
	("ê") `op = ∧`	("ou") `op = ∨`	("ou exclusivo") `op = ⊕`
`V op V`	V	V	F
`V op F`	F	V	V
`F op V`	F	V	V
`F op F`	F	F	F

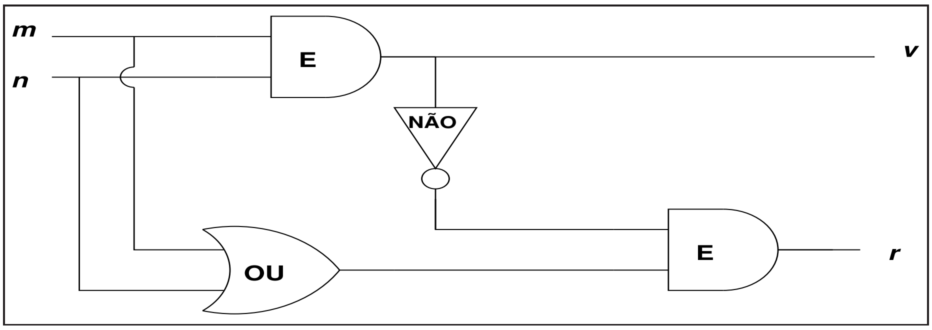

Figura 1.3: Circuito do meio-somador

são falsos, e verdadeiro em caso contrário. A operação lógica "ou exclusivo" tem resultado verdadeiro se um dos operandos, mas não ambos, é verdadeiro, e falso caso contrário.

Para entender como portas lógicas podem ser usadas para implementar a soma de números inteiros positivos em um computador, considere primeiramente a soma de dois números n e m, representados na base binária, cada qual com apenas 1 bit, ilustrada a seguir:

$$\begin{array}{ccccc} 1 & + & 1 & = & 10 \\ 1 & + & 0 & = & 01 \\ 0 & + & 1 & = & 01 \\ 0 & + & 0 & = & 00 \end{array}$$

ou

n	**m**	**"vai um"**	**r**
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Ao comparar a tabela acima com as operações lógicas definidas na Tabela 1.1, é fácil perceber que a operação lógica "ou exclusivo" fornece o bit r do numeral que representa o resultado da soma $n + m$: o bit r é igual a `1` se n ou m for igual a `1`, mas não ambos, e é igual a `0`, caso contrário. O bit "vai um" desse numeral é obtido pela operação lógica "ê": o bit "vai um" é igual a `1` quando n e m são iguais a `1`, e é igual a `0` em caso contrário.

O resultado da soma $n + m$ pode ser, portanto, representado pelo par $(n \wedge m, n \oplus m)$, em que o primeiro componente é o bit "vai um" e o segundo é o bit r do resultado. Note que $m \oplus n = (m \vee n) \wedge \neg(m \wedge n)$.

Com base nessas observações, fica fácil construir um circuito para somar dois números binários n e m, cada qual representado com apenas 1 bit. Esse circuito, chamado de *meio-somador*, é apresentado na Figura 1.3. Símbolos usuais são empregados, nessa figura, para representar as portas lógicas que implementam as operações "ê", "ou" e "não".

O meio-somador pode ser usado para construir um circuito que implementa a soma de três números binários n, m e p, cada qual representado com apenas 1 bit, usando o fato de que a operação de adição é associativa: $n + m + p = (n + m) + p$. Sendo $n + m = (v_1, r_1)$ e $r_1 + p = (v_2, r)$, temos que $n + m + p = (v_1 \vee v_2, r)$, uma vez que v_1 e v_2 não podem ser

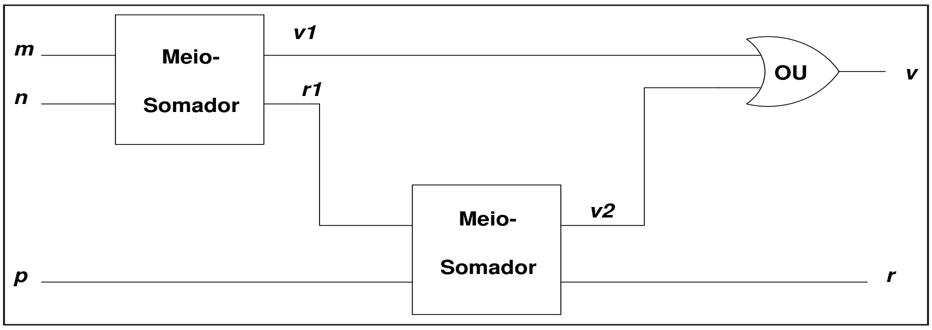

Figura 1.4: Circuito do somador completo

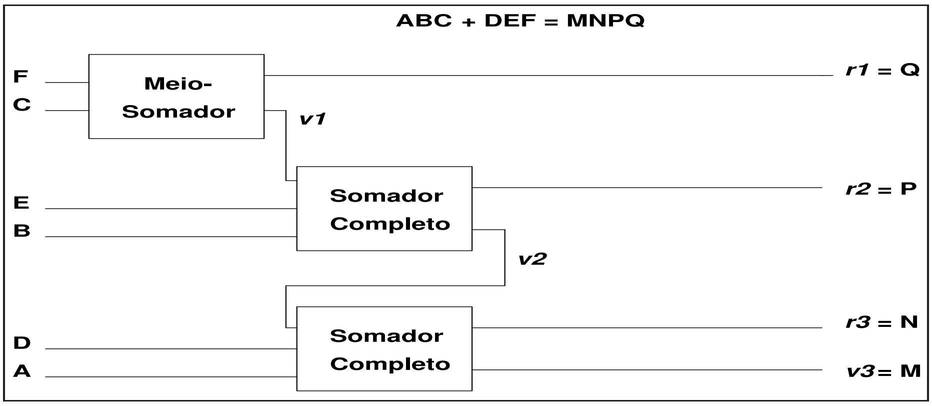

Figura 1.5: Circuito do somador paralelo

ambos iguais a `1`. O circuito lógico que implementa a soma $n + m + p$, chamado de *somador completo*, pode ser construído como mostra a Figura 1.4.

Podemos agora facilmente construir o chamado *somador paralelo*, para somar números inteiros, representados no sistema de numeração binário, com qualquer número fixo de bits.

A Figura 1.5 ilustra um circuito somador paralelo para somar números binários n e m, cada qual representado com 3 bits, `ABC` e `DEF`, respectivamente.

3. Sabemos que um computador é capaz de operar com números inteiros, positivos ou negativos. Como números inteiros negativos são representados no computador?

Para maior facilidade de armazenamento e de operação, todo número é representado, em um computador, por uma seqüência de bits de tamanho fixo. No caso de números inteiros, esse tamanho é igual ao número de bits que podem ser armazenados na palavra do computador. Em grande parte dos computadores modernos, esse tamanho é de 32 bits ou, em computadores ainda mais modernos, de 64 bits.

Para representar tanto números inteiros não-negativos quanto negativos, um determinado bit dessa seqüência poderia ser usado para indicar o sinal do número — essa abordagem é chamada de *sinal-magnitude*.

A abordagem de *sinal-magnitude* não é muito adequada, pois existem nesse caso duas possíveis representações para o zero e as operações de somar números não é tão simples quanto no caso da representação em *complemento de dois*, usada em todos os computadores modernos.

A característica fundamental dessa representação é a de que a operação de somar `1` ao maior inteiro positivo fornece o menor inteiro negativo. Desse modo existe apenas uma representação para o zero e pode-se realizar operações aritméticas de modo bastante simples.

Ilustramos, na Tabela 1.2, a representação de números na notação de complemento de 2 em um computador com uma palavra de apenas 4 bits. Note que existem 2^n combinações distintas em uma palavra de n bits, e portanto é possível representar 2^n números inteiros, que na representação de complemento de dois compreendem os inteiros na faixa de $-2^{(n-1)}$ a $2^{(n-1)} - 1$.

O *complemento de 2* de um número inteiro positivo p, $0 < p \leq 2^{n-1}$, com relação a n bits, denotado por $c_2^n(p)$, é definido como sendo a representação na base binária, com n bits, do número positivo $2^n - p$. Na notação de complemento de 2, um número inteiro $p \geq 0$ é representado na base binária, com n bits, da maneira usual, e um número inteiro $p < 0$ é representado pelo complemento de 2 do valor absoluto de p, $c_2^n(|p|)$.

Tabela 1.2: Representação de inteiros em 4 bits

0	0000		-8	1000
1	0001		-1	1111
2	0010		-2	1110
3	0011		-3	1101
4	0100		-4	1100
5	0101		-5	1011
6	0110		-6	1010
7	0111		-7	1001

Dado um número inteiro positivo p, uma maneira eficiente para calcular $c_2^n(p)$, a partir da representação de p na base binária, pode ser obtida pela observação de que $c_2^n(p) = 2^n - p = (2^n - 1) - p + 1$. Como a representação de $2^n - 1$ na base binária consiste de uma seqüência de n `1`s, é fácil ver que, para obter o resultado da subtração $(2^n - 1) - p$, ou seja, $c_2^n(p) - 1$, também chamado de *complemento de 1* de p, basta tomar a representação de p na base binária e trocar, nessa representação, os `1`s por `0`s e vice-versa. Para obter $c_2^n(p)$, precisamos então apenas somar `1` ao resultado obtido.

Por exemplo:

$$\text{representação de -1 em 4 bits} = c_2^4(1) = c_2^4(\texttt{0001}) = \texttt{1110} + 1 = \texttt{1111}$$
$$\text{representação de -7 em 4 bits} = c_2^4(7) = c_2^4(\texttt{0111}) = \texttt{1000} + 1 = \texttt{1001}$$

Exercício: Como seriam representados os números 17 e -17, em um computador com palavra de tamanho igual a 8 bits?

Para obter a representação usual de um número inteiro p na base binária, dada a sua representação na notação de complemento de 2, basta observar que $2^n - (2^n - p) = p$. Portanto, dado $c_n^2(p)$, a representação de p na base binária pode ser obtida calculando o complemento de 2 de $c_2^n(p)$, ou seja, $c_2^n(c_2^n(p))$. Por exemplo: de $c_2^4(p) = \texttt{1111}_2$ obtemos $p = c_2^4(\texttt{1111}_2) = \texttt{0001}_2$.

Exercício: Determine a representação na base decimal do número p tal que $c_2^4(p) = \texttt{1010}_2$.

Exercício: Determine a representação na base decimal dos números representados por `01100001` e `10011111`, em um computador com palavra de tamanho igual a 8 bits.

Você provavelmente terá observado que o bit mais à esquerda da representação de um número inteiro na notação de complemento de 2 é igual a `1`, se o número for negativo, e igual a `0`, caso contrário. Esse bit pode ser, portanto, interpretado como o sinal do número.

Usando essa representação, a adição de dois números inteiros n e m pode ser feita da maneira usual, sendo descartado o bit "vai um" obtido mais à esquerda. Por exemplo:

$$\begin{array}{rr} & 0110_2 \\ + & 1001_2 \\ \hline = & 1111_2 \end{array} \qquad\qquad \begin{array}{rr} & 1110_2 \\ + & 1101_2 \\ \hline = & 1011_2 \end{array}$$

Ou seja, $6 + (-7) = -1$ e $(-2) + (-3) = (-5)$.

A demonstração de que esse procedimento fornece o resultado correto para a operação de adição foge um pouco do escopo deste livro.

1.5 Exercícios

1. Determine a representação, no sistema de numeração binário, de cada um dos seguintes números, escritos na base decimal:

(a) 19 (b) 458

2. Determine a representação, no sistema de numeração decimal, de cada um dos seguintes números, escritos na base binária:

(a) 1110_2 (b) 110110_2

3. Realize as operações abaixo, sobre números representados no sistema de numeração binário:

(a) $1011 + 101$ (b) $10100 - 1101$

4. Determine a representação na notação de complemento de 2, com 8 bits, de cada um dos seguintes números:

(a) -23 (b) -108

5. Determine a representação na base decimal de cada um dos seguintes números, representados na notação de complemento de 2, com 8 bits:

(a) 11010011 (b) 11110000

6. Indique como seria feito o cálculo das seguintes operações, em um computador que utiliza notação de complemento de 2 para representação de números e tem palavra com tamanho de 8 bits:

(a) $57 + (-118)$ (b) $(-15) + (-46)$

1.6 Notas Bibliográficas

Os primeiros estudos em ciência da computação, realizados por volta de 1935, estabeleceram os fundamentos teóricos da área, lançando as bases para a construção dos primeiros computadores. Como resultado desses estudos, foi estabelecida uma caracterização formal para a noção intuitiva de algoritmo, e concluiu-se também existem problemas chamados *indecidíveis*, cuja solução não pode ser obtida por meio de nenhum programa de computador. Ao leitor interessado em saber mais sobre esse assunto, recomendamosa leitura de [24]. Uma discussão interessante sobre a influência dos recentes resultados da teoria da computação no desenvolvimento tecnológico e científico alcançados no século 20 é apresentada em [13] e [22].

Para um bom entendimento sobre os fundamentos teóricos da computação, é necessário algum conhecimento de matemática discreta, que abrange temas como lógica matemática, teoria de conjuntos, relações e funções, indução e recursão. Dois livros excelentes, que abordam esses temas de maneira introdutória, são [6] e [29].

O funcionamento e organização de computadores, descritos brevemente neste capítulo, é discutido detalhadamente em diversos livros específicos sobre o assunto, dentre os quais recomendamos [30, 2, 7].

Capítulo 2

Paradigmas de Programação

O número de linguagens de programação existentes atualmente chega a ser da ordem de alguns milhares. Esse número impressionante reflete o esforço no sentido de projetar linguagens que facilitem sempre mais a atividade de programação, tornando-a cada vez mais produtiva. Outro objetivo importante no projeto dessas linguagens é o de favorecer a construção de programas mais eficientes, isto é, que originem programas executáveis rapidamente e que usam relativamente pouca quantidade de memória de um computador, e seguros, isto é, menos sujeitos a erros que possam ocasionar um comportamento da execução do programa diferente daquele que é esperado.

Apesar dessa grande diversidade de linguagens de programação, a maioria delas apresenta, essencialmente, o mesmo conjunto de comandos básicos,[1] embora esses possam apresentar formas diferentes em diferentes linguagens. Esse conjunto é constituído de:

- um comando básico — denominado *comando de atribuição* — usado para armazenar um valor em uma determinada posição de memória;
- comandos para leitura de dados, de dispositivos de entrada, e para escrita de dados, em dispositivos de saída;
- três formas distintas de combinação de comandos:
  - *composição seqüencial* — execução de um comando após outro,
  - *seleção* (ou *escolha condicional*) — escolha de um comando para ser executado, de acordo com o resultado da avaliação de uma condição,
  - *repetição* — execução de um comando repetidas vezes, até que uma condição seja satisfeita.

Em uma linguagem com essas características, um programa consiste em uma seqüência de comandos que descreve, em essência, como devem ser modificados os valores armazenados na memória do computador, de maneira que uma determinada tarefa seja realizada. Esse paradigma de programação é denominado *paradigma imperativo* e linguagens baseadas nesse paradigma são chamadas de *linguagens imperativas*.

Existem, em contraposição, outros paradigmas de programação, chamados *declarativos*, nos quais um programa se assemelha mais a uma descrição de *o que* constitui uma solução de um determinado problema, em lugar de *como* proceder para obter essa solução.

A linguagem `C` é uma linguagem que provê suporte ao paradigma imperativo de programação. Os principais conceitos da programação imperativa são apresentados neste capítulo, como uma visão geral e introdutória. Cada um desses conceitos é novamente abordado em capítulos subseqüentes, nos quais introduzimos, passo a passo, por meio de vários exemplos, a aplicação desses conceitos na construção de programas para a solução de diversos problemas.

O objetivo deste capítulo é prover uma visão mais global sobre conceitos básicos de linguagens de programação, o que acreditamos irá contribuir para uma melhor compreensão da função e do significado de cada um deles individualmente e da estrutura de uma linguagem de programação

[1] O termo "instrução" é em geral usado para linguagens de mais baixo nível, enquanto o termo "comando", em princípio equivalente, é mais usado para linguagens de alto nível.

como um todo, além de facilitar o aprendizado de como aplicar esses conceitos no desenvolvimento de programas.

2.1 Variável e Atribuição

Em um programa em linguagem imperativa, uma *variável* representa um lugar que contém um certo valor. Esse lugar é uma determinada área da memória do computador.

Esse conceito de variável difere daquele a que estamos acostumados em matemática. Em uma expressão matemática, toda ocorrência de uma determinada variável denota um mesmo valor. Em um programa em linguagem imperativa, ao contrário, ocorrências distintas de uma mesma variável podem representar valores diferentes, uma vez que tais linguagens são baseadas em comandos que têm o efeito de modificar o valor armazenado em uma variável, durante a execução do programa. Além disso, uma determinada ocorrência de uma variável no texto de um programa pode também representar valores diferentes, em momentos distintos da execução desse programa, uma vez que um comando pode ser executado repetidas vezes.

Em `C`, assim como na maioria das linguagens de programação imperativas, toda variável usada em um programa deve ser *declarada* antes de ser usada. Uma declaração de variável especifica o *nome* e o *tipo* da variável, e tem o efeito de criar uma nova variável com o nome especificado. O *tipo* denota o conjunto de valores que podem ser armazenados na variável.

Um nome de uma variável (ou função) deve ser uma sequência de letras ou dígitos ou o caractere sublinha (`'_'`), e deve começar com uma letra ou com o caractere sublinha.

Um nome de uma variável ou função em `C` pode ser qualquer sequência de letras ou dígitos ou o caractere sublinha `'_'`), e deve começar com uma letra ou com o caractere sublinha. No entanto, existem nomes reservados, que não podem ser usados como nomes de variáveis ou funções pelo programador `C`. Por exemplo, `return` é uma palavra reservada em `C` — que inicia um comando usado para especificar o resultado de uma função e fazer com que a execução da função seja interrompida, retornando o resultado especificado — e portanto não pode ser usada como nome de variável ou função.

Nota sobre escolha de nomes de variáveis:
O nome de uma variável ou função pode ser escolhido livremente pelo programador, mas é importante que sejam escolhidos nomes apropriados — ou seja, mnemônicos, que lembrem o propósito ou significado da entidade (variável ou função) representada.
Por exemplo, procure usar nomes como *soma*, *media* e *pi* para armazenar, respectivamente, a soma e a média de determinados valores, e uma aproximação do número π, em vez de simplesmente, digamos, s, m, p (outras letras que não s, m e p são ainda menos mnemônicas, por não ter relação com *soma*, *media* e *pi*). No entanto, é útil procurar usar nomes pequenos, a fim de tornar o programa mais conciso. Por isso, muitas vezes é comum usar nomes como, por exemplo, i e j como contadores de comandos de repetição, e n como um número natural qualquer, arbitrário — de modo semelhante ao uso de letras gregas em matemática. Evite usar nomes como, por exemplo, *aux*, que são pouco significativos e poderiam ser substituídos por nomes mais concisos, caso não haja um nome que represente de forma concisa e mnemônica o propósito de uso de uma variável ou função.

Uma declaração de variável pode, opcionalmente, especificar também o valor a ser armazenado na variável, quando ela é criada — isto é, quando uma área de memória é alocada para essa variável.

Em `C`, a declaração:

```
char x; int y = 10; int z;
```

especifica que x é uma variável de tipo `char`, ou seja, que pode armazenar um caractere, e que y e z são variáveis de tipo `int` (variáveis inteiras, ou de tipo inteiro). A declaração da variável y especifica que o valor `10` deve ser armazenado nessa variável, quando ela é criada. Não são especificados valores iniciais para as variáveis x e z; em `C`, isso significa que um valor inicial indefinido (determinado de acordo com a configuração da memória no instante da execução do comando de declaração) é armazenado em cada uma dessas variáveis. A inexistência de inicialização implícita em `C`, e muitas outras características da linguagem, têm como principal motivação procurar proporcionar maior *eficiência* na execução de programas (ou seja, procuram fazer com que programas escritos em `C` levem menos tempo para serem executados).

Uma expressão de uma linguagem de programação é formada a partir de variáveis e constantes, usando funções ou operadores. Por exemplo, f`(`x`+`y`)` é uma expressão formada pela aplicação de uma função, de nome f, à expressão x`+`y, essa última formada pela aplicação do operador `+` às expressões x e y (nesse caso, variáveis). Toda linguagem de programação oferece um conjunto de funções e operadores predefinidos, que podem ser usados em expressões. Operadores sobre valores de tipos básicos predefinidos em `C` são apresentados no capítulo a seguir.

O valor de uma expressão (ou valor "retornado" pela expressão, como é comum dizer, em computação) é aquele obtido pela avaliação dessa expressão durante a execução do programa. Por exemplo, y`+1` é uma expressão cujo valor é obtido somando `1` ao valor contido na variável y.

Na linguagem `C`, cada expressão tem um tipo, conhecido *estaticamente* — de acordo com a estrutura da expressão e os tipos dos seus componentes. Estaticamente significa *durante a compilação* (ou seja, antes da execução) ou, como é comum dizer em computação, "em tempo de compilação" (uma forma de dizer que tem influência da língua inglesa). Isso permite que um compilador `C` possa detectar "erros de tipo", que são erros devidos ao uso de expressões em contextos em que o tipo não é apropriado. Por exemplo, supondo que `+` é um operador binário (deve ser chamado com dois argumentos para fornecer um resultado), seria detectado um erro na expressão x`+`, usada por exemplo `int` y `=` x`+;` — uma vez que não foram usados dois argumentos (um antes e outro depois do operador `+`) nessa expressão. Um dos objetivos do uso de tipos em linguagens de programação é permitir que erros sejam detectados, sendo uma mensagam de erro emitida pelo compilador, para que o programador possa corrigi-los (evitando assim que esses erros possam ocorrer durante a execução de programas).

A linguagem `C` provê os tipos básicos `int`, `float`, `double`, `char` e `void`. O tipo inteiro representa valores inteiros (sem parte fracionária).

Números de ponto flutuante (`float` ou `double`) contêm uma parte fracionária. Eles são representados em um computador por um valor inteiro correspondente à mantissa e um valor inteiro correspondente ao expoente do valor de ponto flutuante. A diferença entre `float` e `double` é de precisão: um valor de tipo `double` usa um espaço (número de bits) pelo menos igual, mas em geral maior do que um valor de tipo `float`.

Um valor de tipo `char` é usado para armanezar um caractere. Em `C` um valor de tipo `char` é um inteiro, sem sinal (com um tamanho menor ou igual ao de um inteiro).

Não existe valor de tipo `void`; esse tipo é usado basicamente para indicar que uma função não retorna nenhum resultado, ou não tem nenhum parâmetro.

Um *comando de atribuição* armazena um valor em uma variável. Em `C`, um comando de atribuição tem a forma:

v `=` e`;`

A execução desse comando tem o efeito de atribuir o valor resultante da avaliação da expressão e à variável v. Após essa atribuição, não se tem mais acesso, através de v, ao valor que estava armazenado anteriormente nessa variável — o comando de atribuição modifica o valor da variável.[2]

Note o uso do símbolo `=` no comando de atribuição da linguagem `C`, diferente do seu uso mais comum, como símbolo de igualdade. Em `C`, o operador usado para teste de igualdade é `==`. Usar `=` em vez de `==` é um erro cometido com freqüência por programadores iniciantes; é bom ficar atento para não cometer tal erro.

Note também a distinção existente entre o significado de uma variável v como *variável alvo* de um comando de atribuição, isto é, em uma ocorrência do lado esquerdo de um comando de atribuição, e o significado de um uso dessa variável em uma expressão: o uso de v como variável alvo de uma atribuição representa um "lugar" (o endereço da área de memória alocada para v, durante a execução do programa), e não o valor armazenado nessa área, como no caso em que a variável ocorre em uma expressão.

[2] É comum usar, indistintamente, os termos "valor armazenado em uma variável", "valor contido em uma variável" ou, simplesmente, "valor de uma variável".

Por exemplo, supondo que o valor contido em uma variável inteira x seja `10`, após a execução do comando de atribuição "`x = x + 1;`", o valor contido em x passa a ser `11`: o uso de x no lado direito desse comando retorna o valor `10`, ao passo que o uso de x do lado esquerdo desse comando representa uma posição de memória, onde o valor `11` é armazenado.

Um programa em linguagem como `C` pode incluir definições de novas funções, que podem então ser usadas em expressões desse programa. Em linguagens imperativas, a possibilidade de uso de comandos em definições de funções torna possível que a avaliação de uma expressão não apenas retorne um resultado, mas também modifique valores de variáveis. Quando uma expressão tem tal efeito, diz-se que tem um *efeito colateral*.

Em `C`, o próprio comando de atribuição é uma expressão (com efeito colateral). Por exemplo, supondo que a e b são duas variáveis inteiras, podemos escrever:

```
a = b = b + 1;
```

Na execução desse comando, o valor da expressão `b + 1` é calculado, e então atribuído a b e, em seguida, atribuído a a. Se b contém, por exemplo, o valor `3`, antes da execução desse comando, então a expressão `b = b + 1` não só retorna o valor `b + 1`, igual a `4`, como modifica o valor contido em b (que passa a ser `4`).

O comando de atribuição não é o único comando que pode modificar o valor de uma variável. Isso ocorre também no caso de comandos de entrada de dados, também chamados de *comandos de leitura*, que transferem valores de dispositivos externos para variáveis. Um comando de leitura funciona basicamente como um comando de atribuição no qual o valor a ser armazenado na variável é obtido a partir de um dispositivo de entrada de dados. Comandos de entrada e saída de dados são abordados no Capítulo 3.

2.2 Composição Seqüencial

A composição seqüencial é a forma mais simples de combinação de comandos. A composição seqüencial de dois comandos c_1 e c_2 é escrita na forma:

$$c_1\texttt{;}\ c_2\texttt{;}$$

e consiste na execução de c_1 e, em seguida, do comando c_2.

Os comandos c_1 e c_2 podem, por sua vez, também ser formados por meio de composição seqüencial.

A composição seqüencial de comandos é naturalmente associativa (não sendo permitido o uso de parênteses). Por exemplo, a seqüência de comandos:

```
int a; int b; int c; a = 10; b = 20; c = a + b;
```

tem o efeito de:

1. Declarar uma variável, de nome a, como tendo tipo `int`. Declarar uma variável significa alocar uma área de memória e associar um nome a essa área, que pode conter valores do tipo declarado.

2. Analogamente, declarar variáveis b e c, de tipo `int`.

3. Em seguida, atribuir o valor `10` à variável a.

4. Em seguida, atribuir o valor `20` à variável b.

5. Em seguida, atribuir o valor `30`, resultante da avaliação de `a + b`, à variável c.

2.3 Seleção

Outra forma de combinação de comandos, a seleção, possibilita selecionar um comando para execução, conforme o valor de um determinado teste seja verdadeiro ou falso. Em C, a seleção é feita com o chamado "comando `if`", que tem a seguinte forma:

`if ( `b` ) `c_1`; else `c_2`;`

Nesse comando, b tem que ser uma expressão de tipo `int`. A linguagem C não tem um tipo para representar diretamente os valores verdadeiro ou falso, e usa para isso o tipo `int`. A convenção usada é que, em um contexto como o da expressão b, em que um valor verdadeiro ou falso é esperado, `0` representa falso e qualquer valor diferente de zero representa verdadeiro.

Na execução do comando `if`, se o valor retornado pela avaliação de b for qualquer valor direrente de `0`, o comando c_1 é executado; se o valor retornado for `0`, o comando c_2 é executado.

A parte "`else` c_2;" (chamada de "cláusula `else`") é opcional. Se não for especificada, simplesmente nenhum comando é executado no caso em que a avaliação da expressão b retorna falso.

Em C, para se usar uma seqüência com mais de um comando no lugar de c_1, ou no lugar de c_2, devem ser usadas chaves para indicar o início e o fim da seqüência de comandos. Por exemplo:

```
if (a > 10) { a = a + 10; b = b + 1; }
else { b = 0; if (c > 1) a = a + 5; }
```

Uma seqüência de comandos entre chaves é chamada de um *bloco*. Note que, em C, se um bloco for usado no lugar de um comando — como no comando `if` do exemplo anterior —, o caractere "`;`" não deve ser usado após o mesmo: o caractere "`;`" é usado como um *terminador* de comandos, devendo ocorrer após cada comando do programa, que não seja um bloco.

Existe também outra forma de comando de seleção, que seleciona um comando para ser executado, de acordo com o valor de uma expressão, dentre uma série de possibilidades, e não apenas duas, como no caso do comando `if`. Esse comando, tratado no Exercício Resolvido 6 do Capítulo 4, tem o mesmo efeito que uma seqüência de comandos `if`, com testes sucessivos sobre o valor da condição especificada nesse comando.

2.4 Repetição

Em um comando de repetição, um determinado comando, chamado de *corpo* do comando de repetição, é executado repetidas vezes, até que uma *condição de terminação* do comando de repetição se torne verdadeira, o que provoca o término da execução desse comando.

Cada avaliação da condição de terminação, seguida da execução do corpo desse comando, é denominada uma *iteração*, sendo o comando também chamado *comando iterativo*.

Para que a condição de terminação de um comando de repetição possa tornar-se verdadeira, depois de um certo número de iterações, é preciso que essa condição inclua alguma variável que tenha o seu valor modificado pela execução do corpo desse comando (mais especificamente exista um comando de atribuição no corpo do comando de repetição que modifique o valor de uma variável usada na condição de terminação).

O comando `while` é um comando de repetição, que tem, em C, a seguinte forma, onde b é a condição de terminação, e c, o corpo do comando:

`while ( `b` ) `c`;`

A execução desse comando consiste nos seguintes passos: antes de ser executado o corpo c, a condição b é avaliada; se o resultado for verdadeiro (isto é, diferente de `0`), o comando c é executado, e esse processo se repete; senão (i.e., se o resultado da avaliação de b for falso), então a execução do comando `while` termina.[3]

Como exemplo de uso do comando `while`, considere o seguinte trecho de programa, que atribui à variável *soma* a soma dos valores inteiros de `1` a n:

[3]Devido a esse "laço" envolvendo b e depois c repetidamente, um comando de repetição é também chamado de *loop* (fala-se "lup"), que significa laço, em inglês.

```
soma = 0;
i = 1;
while ( i <= n ) {
   soma = soma + i;
   i = i + 1;
}
```

Essa seqüência de comandos determina que, primeiramente, o valor `0` é armazenado em *soma*, depois o valor `1` é armazenado em i, e em seguida o comando `while` é executado.

Na execução do comando `while`, primeiramente é testado se o valor de i é menor ou igual a n. Se o resultado desse teste for falso (igual a `0`), a execução do comando termina. Caso contrário, o corpo do comando `while` é executado, seguindo-se novo teste etc. A execução do corpo do comando `while` adiciona i ao valor da variável *soma*, e adiciona `1` ao valor armazenado na variável i, nessa ordem. Ao final da n-ésima iteração, o valor da variável i será, portanto, n`+1`. A avaliação da condição de terminação, no início da iteração seguinte, retorna então o valor falso (`0`), e a execução do comando `while` termina.

A linguagem `C` possui dois outros comandos de repetição, além do comando `while`: os comandos `do-while` e `for`. Esses comandos têm comportamento semelhante ao do comando `while`, e são abordados no Capítulo 4.

2.5 Funções e Procedimentos

Linguagens de programação de alto nível oferecem construções para definição de novas *funções*, que podem então ser usadas em expressões desse programa, aplicadas a argumentos apropriados. Dizemos que uma função constitui uma abstração sobre uma expressão, uma vez que representa uma expressão, possivelmente parametrizada sobre valores que ocorrem nessa expressão.

O uso de uma função em uma expressão é também chamado, em computação, de uma "chamada" a essa função.

De maneira análoga, um programa pode também incluir definições de *procedimentos*, que constituem abstrações sobre comandos — ou seja, um procedimento consiste em uma seqüência de comandos, possivelmente parametrizada sobre valores usados nesses comandos, podendo ser chamado em qualquer ponto do programa em que um comando pode ser usado.

A possibilidade de definição e uso de funções e procedimentos constitui um recurso fundamental para decomposição de programas, evitando duplicação de código e contribuindo para construção de programas mais concisos e legíveis, assim como mais fáceis de corrigir e modificar.

O termo *função* costuma também ser usado no lugar de *procedimento*, uma vez que em geral uma função pode usar efeitos colaterais (comandos de atribuição e outros comandos que alteram o valor de variáveis) em expressões.

Definições de funções são o tema dos nossos primeiros exemplos, no Capítulo 3.

2.5.1 Blocos, Escopo e Tempo de Vida de Variáveis

A execução de programas em linguagens de programação, como `C` por exemplo, é baseada de modo geral na alocação e liberação de variáveis e de memória para essas variáveis em uma estrutura de *blocos*. A cada função ou procedimento corresponde um bloco, que é o texto de programa correspondente ao corpo da função ou procedimento.

Um bloco pode no entanto, em linguagens como `C`, ser simplesmente um comando constituído por comandos que incluem pelo menos um comando de declaração de variável. Por exemplo, o seguinte programa define um bloco internamente à função *main*, onde é definida uma variável de mesmo nome que uma variável declarada na função *main*:

```
int main() {
  int x = 1;
  { int x=2;
    x = x+1;
  }
  int y = x+3;
}
```

Um bloco determina o *escopo* e o *tempo de vida* de variáveis nele declaradas.

O escopo de uma variável v, criada em um comando de declaração que ocorre em um bloco b, é o trecho (conjunto de pontos) do programa em que a variável pode ser usada, para denotar essa variável v. Em geral, e em particular em `C`, o escopo de v é o trecho do bloco b que é textualmente seguinte à sua declaração, excluindo escopos internos ao bloco de variáveis declaradas com o mesmo nome. Essa regra costuma ser chamada: *definir antes de usar*.

Por exemplo, o escopo da variável x declarada em *main* é o trecho da função *main* que segue a declaração de x no bloco de *main* (i.e. `int x = 1;`), excluindo o escopo de x no bloco onde x é redeclarado (i.e. excluindo o escopo de x correspondente à declaração `int x=2;`). Note que o escopo de x declarado em *main* está restrito a essa função, não inclui outras funções que poderiam estar definidas antes ou depois de *main*.

O tempo de vida de uma variável é o intervalo da execução do programa entre sua criação e o término da existência da variável. Em uma linguagem baseada em uma estrutura de blocos, como `C`, o tempo de vida de uma variável, criada em um comando de declaradação, que ocorre em um bloco b, é o intervalo entre o início e o término da execução do bloco b (i.e. o intervalo entre o início da execução do primeiro e o último comandos do bloco).

As noções de tempo de vida e escopo de variáveis serão mais abordadas no Capítulo seguinte, ao tratarmos de chamadas de funções, em particular de chamadas de funçoes recursivas.

2.6 Outros Paradigmas de Programação

Além dos paradigmas de programação imperativo e orientado por objetos, existem, como mencionamos na introdução deste capítulo, outros paradigmas de programação, mais *declarativos*: o paradigma funcional e o paradigma lógico. Embora esses paradigmas sejam ainda relativamente pouco utilizados, o interesse por eles tem crescido de maneira significativa.

No *paradigma funcional*, um programa consiste, essencialmente, em uma coleção de definições de funções, cada qual na forma de uma série de equações. Por exemplo, a função que determina o fatorial de um número inteiro não-negativo n, poderia ser definida pelas equações:

```
fatorial 0 = 1
fatorial n = n * fatorial(n-1)
```

A execução de um programa em linguagem funcional consiste na avaliação de uma determinada expressão desse programa, que usa as funções nele definidas. O fato de que essas definições de funções podem também ser vistas como regras de computação estabelece o caráter operacional dessas linguagens.

Uma característica importante de linguagens funcionais é o fato de que possibilitam definir *funções de ordem superior*, isto é, funções que podem ter funções como parâmetros, ou retornar uma função como resultado. Essa característica facilita grandemente a decomposição de programas em componentes (funcionais) e a combinação desses componentes na construção de novos programas.

O maior interesse pela programação funcional apareceu a partir do desenvolvimento da linguagem ML e, mais recentemente, da linguagem Haskell (veja Notas Bibliográficas).

No *paradigma lógico*, um programa tem a forma de uma série de asserções (ou regras), que definem relações entre variáveis. A denominação dada a esse paradigma advém do fato de que a linguagem usada para especificar essas asserções é um subconjunto da *Lógica de Primeira Ordem* (também chamada de *Lógica de Predicados*). Esse subconjunto da lógica de primeira ordem usado em linguagens de programação em lógica usa asserções simples (formada por termos, ou expressões, que têm valor verdadeiro ou falso), da forma:

$$P \text{ se } P_1, P_2, \ldots, P_n$$

A interpretação *declarativa* dessa asserção é, informalmente, a de que P é verdadeiro se e somente se todos os termos $P_1, \ldots, P_n$ ($n \geq 0$) forem verdadeiros. Além dessa interpretação declarativa, existe uma interpretação operacional: para executar (ou resolver) P, execute P_1, depois P_2, etc., até P_n, fornecendo o resultado verdadeiro se e somente se o resultado de cada uma das avaliações de $P_1, \ldots, P_n$ fornecer resultado verdadeiro.

A linguagem Prolog é a linguagem de programação mais conhecida e representativa do paradigma de programação em lógica. Existe também, atualmente, um interesse expressivo em pesquisas com novas linguagens que exploram o paradigma de programação em lógica com restrições, e com linguagens que combinam a programação em lógica, a programação funcional e programação orientada por objetos (veja Notas Bibliográficas).

2.7 Exercícios

1. Qual é o efeito das declarações de variáveis durante a execução do trecho de programa abaixo?

```
int x;
int y = 10;
```

2. Quais são os valores armazenados nas variáveis x e y, ao final da execução do seguinte trecho de programa?

```
int x; int y = 10;
x = y * 3;
while (x > y) {
  x = x - 5; y = y + 1;
}
```

3. Qual é o valor da variável s ao final da execução do seguinte trecho de programa, nos dois casos seguintes:

(a) as variáveis a e b têm, inicialmente, valores `5` e `10`, respectivamente;

(b) as variáveis a e b têm, inicialmente, valores `8` e `2`, respectivamente.

```
s = 0;
if (a > b) s = (a+b)/2;
while (a <= b) {
  s = s + a;
  a = a + 1;
  b = b - 2;
}
```

2.8 Notas Bibliográficas

Existem vários livros introdutórios sobre programação de computadores, a maioria deles em língua inglesa, abordando aspectos diversos da computação. Grande número desses livros adota a linguagem de programação PASCAL [12], que foi originalmente projetada, na década de 1970, especialmente para o ensino de programação. Dentre esses livros, citamos [8, 16].

A década de 1970 marcou o período da chamada *programação estruturada* [8], que demonstrou os méritos de programas estruturados, em contraposição à programação baseada em linguagens de mais baixo nível, mais semelhantes a linguagens de montagem ou linguagens de máquina. A linguagem Pascal, assim como outras linguagens também baseadas no paradigma de programação imperativo, como `C` [4], são ainda certamente as mais usadas no ensino introdutório de programação de computadores. Livros de introdução à programação de computadores baseados no uso de Pascal ou de linguagens similares, escritos em língua portuguesa, incluem, por exemplo, [3].

A partir do início da década de 1980, o desenvolvimento de software, em geral, e as linguagens de programação, em particular, passaram a explorar a idéia de decomposição de um sistema em partes e o conceito de *módulo*, originando o estilo de *programação modular*. As novas linguagens de programação desenvolvidas, tais como Modula-2 [17], Ada [11] e, mais tarde, Modula-3 [23], passaram a oferecer recursos para que partes de um programa pudessem ser desenvolvidas e compiladas separadamente, e combinadas de forma segura.

A programação orientada por objetos, que teve sua origem bastante cedo, com a linguagem Simula [19], só começou a despertar maior interesse a partir da segunda metade da década de 1980, após a definição da linguagem e do ambiente de programação Smalltalk [1]. A linguagem Smalltalk teve grande influência sobre as demais linguagens orientadas por objeto subseqüentes, tais como C++ [25], Eiffel [15] e Java [10].

O grande trunfo da programação orientada por objetos, em relação à programação modular, é o fato de explorar mais o conceito de *tipo*, no sentido de que uma parte de um programa, desenvolvida separadamente, constitui também um tipo, que pode ser usado como tipo de variáveis e expressões de um programa.

A influência da linguagem Java se deve, em grande parte, ao enorme crescimento do interesse por aplicações voltadas para a Internet, aliado às características do sistema de tipos da linguagem, que favorecem a construção de programas mais seguros, assim como ao grande número de classes e ferramentas existentes para suporte ao desenvolvimento de programas nessa linguagem.

O recente aumento do interesse por linguagens funcionais é devido, em grande parte, aos sistemas de tipos dessas linguagens. Os sistemas de tipos de linguagens funcionais modernas, como ML [20] e Haskell [27, 5], possibilitam a definição e uso de tipos e funções polimórficas, assim como a inferência automática de tipos.

Uma função polimórfica é uma função que opera sobre valores de tipos diferentes — todos eles instâncias de um tipo polimórfico mais geral. No caso de polimorfismo paramétrico, essa função apresenta um comportamento uniforme para valores de qualquer desses tipos, isto é, comportamento independente do tipo específico do valor ao qual a função é aplicada. De maneira semelhante, tipos polimórficos são tipos parametrizados por variáveis de tipo, de maneira que essas variáveis (e os tipos polimórficos correspondentes) podem ser "instanciadas", fornecendo tipos específicos. Em pouco tempo de contacto com a programação funcional, o programador é capaz de perceber que os tipos das estruturas de dados e operações usadas repetidamente na tarefa de programação são, em sua grande maioria, polimórficos. Por exemplo, a função que calcula o tamanho (ou comprimento) de uma lista é polimórfica, pois seu comportamento independe do tipo dos elementos da lista.

Essas operações, usadas com freqüência em programas, são também, comumente, funções de ordem superior, isto é, funções que recebem funções como parâmetros ou retornam funções como resultado. Como um exemplo bastante simples, a operação de realizar uma determinada operação sobre os elementos de uma estrutura de dados pode ser implementada como uma função polimórfica de ordem superior, que recebe como argumento a função a ser aplicada a cada um dos elementos dessa estrutura.

A inferência de tipos, introduzida pioneiramente na linguagem ML, que também introduziu o sistema de tipos polimórficos, possibilita combinar duas características convenientes: a segurança e maior eficiência proporcionadas por sistemas com verificação de erros de tipo em tempo de compilação e a flexibilidade de não requerer que tipos de variáveis e expressões de um programa sejam especificados explicitamente pelo programador (essa facilidade era anteriormente encontrada apenas em linguagens que realizam verificação de tipos em tempo de execução, as quais são, por isso, menos seguras e menos eficientes).

Estes e vários outros temas interessantes, como o uso de estratégias de avaliação de expressões até então pouco exploradas e o modelamento de mudanças de estado em programação funcional, são ainda objeto de pesquisas na área de projeto de linguagens de programação. O leitor interessado nesses temas certamente encontrará material motivante nos livros sobre programação em linguagem funcional mencionados acima.

Ao leitor interessado em aprender mais sobre o paradigma de programação em lógica e a linguagem Prolog recomendamos a leitura de [14, 28].

Capítulo 3

Primeiros Problemas

Neste capítulo, introduzimos um primeiro conjunto de problemas, bastante simples, e exploramos um raciocínio também bastante simples de construção de algoritmos. A solução de cada um desses problemas é expressa na forma de uma definição de função, em `C`.

Esses primeiros problemas têm o propósito de ilustrar os mecanismos de definição e uso de funções em programas e o uso de comandos de seleção. Além disso, visam também introduzir operações sobre valores inteiros e alguns aspectos sintáticos da linguagem.

O nosso primeiro conjunto de problemas é especificado a seguir:

1. dado um número inteiro, retornar o seu quadrado;
2. dados dois números inteiros, retornar a soma dos seus quadrados;
3. dados três números inteiros, determinar se são todos iguais ou não;
4. dados três números inteiros, a, b e c, determinar se eles podem representar os lados de um triângulo ou não (isso é, se existe um triângulo com lados de comprimentos iguais a a, b e c).
5. dados dois números inteiros, retornar o máximo entre eles;
6. dados três números inteiros, retornar o máximo entre eles.

3.1 Funções sobre Inteiros e Seleção

A Figura 3.1 apresenta definições de funções para solução de cada um dos problemas relacionados acima. Cada uma das funções opera apenas sobre valores inteiros e tem definição bastante simples, baseada apenas no uso de outras funções, predefinidas na linguagem, ou definidas no próprio programa.

O programa começa com um comentário. Comentários são adicionados a um programa para tornar mais fácil a sua leitura. Eles não têm nenhum efeito sobre o comportamento do programa, quando esse é executado. Nos exemplos apresentados neste livro, comentários são muitas vezes omitidos, uma vez que a função e o significado dos programas são explicados ao longo do texto.

Existem dois tipos de comentários em `C`. O primeiro começa com os caracteres `/*` e termina com `*/` — qualquer sequência de caracteres entre `/*` e `*/` faz parte do comentário. O comentário no início do nosso programa é um exemplo desse tipo de comentário. O segundo tipo de comentário começa com os caracteres `//` em uma dada linha e termina no final dessa linha. Um exemplo é o comentário usado na definição da função *eTriang*.

A definição de cada função obedece à seguinte estrutura padrão de declarações de funções:

1. Inicialmente é especificado o tipo do valor fornecido como resultado, em uma chamada (aplicação) da função.

   O tipo do valor retornado por cada função declarada acima é `int` (os nomes das funções são *quadrado*, *somaDosQuadrados*, *tresIguais*, *max* e *max3*).

```
/*****************************************************
*                                                   *
*               Primeiros exemplos                  *
*                                                   *
*      Definições de funções                        *
*---------------------------------------------------*/

int quadrado (int x) { return x*x; }

int somaDosQuadrados (int x, int y) {
  return (quadrado(x) + quadrado(y));
}

int tresIguais (int a, int b, int c) {
  return ((a==b) && (b==c));
}

int eTriang (int a, int b, int c) {
  // a, b e c positivos e
  // cada um é menor do que a soma dos outros dois
  return (a>0) && (b>0) && (c>0) &&
         (a<b+c) && (b<a+c) && (c<a+b);
}

int max (int a, int b) {
  if (a >= b) return a; else return b;
}

int max3 (int a, int b, int c) {
  return (max(max(a,b),c));
}
```

Figura 3.1: Definições de funções: primeiros exemplos em `C`

2. Em seguida vem o nome da função, e depois, entre parênteses, a lista dos seus parâmetros (que pode ser vazia).

   A especificação de um parâmetro consiste em um tipo, seguido do nome do parâmetro. A especificação de cada parâmetro é separada da seguinte por uma vírgula.

   Por exemplo, a declaração de *somaDosQuadrados* especifica que essa função tem dois parâmetros: o primeiro tem tipo `int` e nome x, o segundo também tem tipo `int` e nome y.

3. Finalmente, é definido o corpo do método, que consiste em um bloco, ou seja, uma sequência de comandos, delimitada por "{" (abre-chaves) e "}" (fecha-chaves).

A execução do corpo do método *quadrado* — { `return` x `*` x } — retorna o quadrado do valor (x) passado como argumento em uma chamada a esse método.

O valor retornado pela execução de uma chamada a um método é determinado pela expressão que ocorre como argumento do comando `return`, que deve ocorrer no corpo desse método. O efeito da execução de um comando da forma:

`return` e;

é o de avaliar a expressão e, obtendo um determinado valor, e finalizar a execução do método, retornando esse valor.

Tabela 3.1: Operadores de comparação

Operador	Significado	Exemplo	Resultado
`==`	Igual a	`1 == 1`	verdadeiro
`!=`	Diferente de	`1 != 1`	falso
`<`	Menor que	`1 < 1`	falso
`>`	Maior que	`1 > 1`	falso
`<=`	Menor ou igual a	`1 <= 1`	verdadeiro
`>=`	Maior ou igual a	`1 >= 1`	verdadeiro

Uma função definida em um programa pode ser usada do mesmo modo que funções predefinidas na linguagem. Por exemplo, a função *quadrado* é usada na definição da função *somaDosQuadrados*, assim como o operador predefinido `+`, que representa a operação (ou função) de adição de inteiros. A avaliação da expressão *quadrado*`(3) +` *quadrado*`(4)` consiste em avaliar a expressão *quadrado*`(3)`, o que significa executar o comando `return 3 * 3`, que retorna `9` como resultado da chamada *quadrado*`(3)`; em seguida, avaliar a expressão *quadrado*`(4)`, de maneira análoga, retornando `16`; e finalmente avaliar a expressão `9 + 16`, o que fornece o resultado `25`.

O número e o tipo dos argumentos em uma chamada de método devem corresponder ao número e tipo especificados na sua definição. Por exemplo, em uma chamada a *tresIguais*, devem existir três expressões e_1, e_2 e e_3, como a seguir:

$$\textit{tresIguais}\ (e_1,e_2,e_3)$$

Essa chamada pode ser usada em qualquer contexto que requer um valor de tipo `int` ou, em outras palavras, em qualquer lugar onde uma expressão de tipo `int` pode ocorrer.

O corpo do método *tresIguais* usa o operador `&&`, que representa a operação booleana "ê". Três valores a, b e c são iguais, se a é igual a b *e* b é igual a c, o que é expresso pela expressão `(`a`==`b`) && (`b`==`c`)`.

Como vimos anteriormente, o símbolo `==` é usado para comparar a igualdade de dois valores. A avaliação de uma expressão da forma "e_1 `==` e_2" tem resultado verdadeiro (em `C`, qualquer valor inteiro diferente de zero) se os resultados da avaliação de e_1 e e_2 são iguais, e falso (em `C`, o inteiro zero), caso contrário.

A operação de desigualdade é representada por `!=`. Outras operações de comparação (também chamadas operações relacionais) são apresentadas na Tabela 3.1.

O operador `&&` é também usado na expressão que determina o valor retornado pela função *eTriang*. O valor dessa expressão será falso (zero) ou verdadeiro (diferente de zero), conforme os valores dos parâmetros da função — a, b e c — constituam ou não lados de um triângulo. Isso é expresso pela condição: os valores são positivos — `(`a`>0) && (`b`>0) && (`c`>0)` — e cada um deles é menor do que a soma dos outros dois — `(`a`<`b`+`c`) && (`b`<`a`+`c`) && (`c`<`a`+`b`)`.

O operador `&&`, assim como outros operadores lógicos que podem ser usados em `C`, são descritos na Seção 3.8.

O método *max* retorna o máximo entre dois valores, passados como argumentos em uma chamada a essa função, usando um comando "`if`". O valor retornado por uma chamada a *max*`(`a`,`b`)` é o valor contido em a, se o resultado da avaliação da expressão a `>=` b for verdadeiro, caso contrário o valor contido em b.

Finalmente, no corpo da definição de *max3*, o valor a ser retornado pelo método é determinado pela expressão *max*`(`*max*`(`a`,`b`),`c`)`, que simplesmente usa duas chamadas ao método *max*, definido anteriormente. O resultado da avaliação dessa expressão é o máximo entre o valor contido em c e o valor retornado pela chamada *max*`(`a`,`b`)`, que é, por sua vez, o máximo entre os valores dados pelos argumentos a e b.

Uma maneira alternativa de definir as funções *max* e *max3* é pelo uso de uma *expressão condicional*, em vez de um comando condicional. Uma expressão condicional tem a forma:

$$e\ \texttt{?}\ e_1\ \texttt{:}\ e_2$$

onde e é uma expressão booleana e e_1 e e_2 são expressões de um mesmo tipo. O resultado da avaliação de e ? e_1 : e_2 é igual ao de e_1 se a avaliação de e for igual a `true`, e igual ao de e_2, em caso contrário. As funções *max* e *max3* poderiam ser então definidas como a seguir:

```
int max (int a, int b) {
  return (a >= b ?  a : b);
}

int max3 (int a, int b, int c) {
  return (a >= b ?  max(a,c) : max(b,c));
}
```

3.2 Entrada e Saída: Parte 1

Sob o ponto de vista de um usuário, o comportamento de um programa é determinado, essencialmente, pela *entrada* e pela *saída* de dados desse programa.

Nesta seção, apresentamos uma introdução aos mecanismos de entrada e saída (E/S) de dados disponíveis na linguagem `C`, abordando inicialmente um tema relacionado, que é o uso de cadeias de caracteres (chamados comumente de *string*s em computação).

O suporte a caracteres em `C` é apresentado mais detalhadamente na seção 3.4 (isto é, a seção descreve como caracteres são tratados na linguagem `C`, apresentando o tipo `char` e a relação desse tipo com tipos inteiros em `C`), e o suporte a sequências (ou cadeias) de caracteres é apresentado na seção 5.5.

Em `C`, assim como em grande parte das linguagens de programação, os mecanismos de E/S não fazem parte da linguagem propriamente dita, mas de uma biblioteca padrão, que deve ser implementada por todos os ambientes para desenvolvimento de programas na linguagem. Essa biblioteca é chamada de *stdio*.

A biblioteca *stdio* provê operações para entrada de dados no *dispositivo de entrada padrão* (geralmente o teclado do computador), e de saída de dados no *dispositivo de saída padrão* (geralmente a tela do computador).

Qualquer programa que use a biblioteca *stdio* para realizar alguma operação de entrada ou saída de dados deve incluir a linha

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

antes do primeiro uso de uma função definida na biblioteca.

Vamos usar principalmente duas funções definidas na biblioteca *stdio*, respectivamente para entrada e para saída de dados: *scanf* e *printf*.

Um uso da função *printf* tem o seguinte formato:

```
int printf( str, v1, ..., vn )
```

onde *str* é um literal de tipo string (cadeia de caracteres), escrita entre aspas duplas, e $v_1, \ldots, v_n$ são argumentos (valores a serem impressos).

A sequência de caracteres impressa em uma chamada a *printf* é controlada pelo parâmetro *str*, que pode conter especificações de controle da operação de saída de dados. Essas especificações de controle contêm o caractere `%` seguido de outro caractere indicador do tipo de conversão a ser realizada.

Por exemplo:

```
int x = 10;
printf("Resultado = %d", x)
```

faz com que o valor da variável inteira x (`10`) seja impresso em notação decimal, precidido da sequência de caracteres `"Resultado = "`, ou seja, faz com que seja impresso:

```
Resultado = 10
```

Existem ainda os caracteres `x,o,e,f,s,c`, usados para leitura e impressão de, respectivamente, inteiros em notação hexadecimal e octal, valores de tipo `double`, com (`e` e sem (`f`) parte referente ao expoente, cadeias de caracteres (strings) e caracteres.

Um número pode ser usado, antes do caractere indicador do tipo de conversão, para especificar um tamanho fixo de caracteres a ser impresso (brancos são usados para completar este número mínimo se necessário), assim como um ponto e um número, no caso de impressão de valores de ponto flutuante, sendo que o número indica neste caso o tamanho do número de dígitos da parte fracionária.

Um uso da função *scanf* tem o seguinte formato:

`int` *scanf* `(` *str*`,` v_1 `, ...,` v_n `)`

onde *str* é um literal de tipo string (cadeia de caracteres), escrito entre aspas duplas, e v_1`,...,`v_n são argumentos (valores a serem impressos).

A sequência de caracteres impressa em uma chamada a *printf* é igual a *str* mas pode conter o que são chamadas *especificações de formato*. Especificações de formato contêm o caractere `%` seguido de outro caractere indicador de um tipo de conversão que deve ser relizada. Especificações de formato podem também ocorrer em chamadas a *scanf*, para conversão do valor lido, como mostramos a seguir.

Por exemplo:

```
int x = 10;
printf("Resultado = %d", x)
```

faz com que o valor da variável inteira x (`10`) seja impresso em notação decimal, precedido da sequência de caracteres `"Resultado = "`, ou seja, faz com que seja impresso:

```
Resultado = 10
```

Neste exemplo poderíamos ter impresso diretamente a sequência de caracteres `"Resultado = 10`), mas o intuito é o de ilustrar o uso de `%d`, que será usado de modo mais interessante depois que introduzirmos a função *scanf* de entrada de dados, a seguir.

Para ler um valor inteiro e armazená-lo em uma variável, digamos a, a função *scanf* pode ser usada, como a seguir:

```
scanf("%d",&a);
```

Basicamente, esse comando deve ser entendido como: leia um valor inteiro do dispositivo de entrada entrada padrão e armazene esse valor na variável a.

O dispositivo de entrada padrão é normalmente o teclado, de modo que a operação de leitura interrompe a execução do programa para esperar que um valor inteiro seja digitado no teclado, seguido da tecla de terminação de linha (`Enter`).

O uso de `"%d"` em uma especifação de formato indica, como explicado acima, que deve ser feita uma conversão do valor digitado, em notação decimal, para um valor inteiro correspondente (representado como uma sequência de bits).

O caractere `&` siginifica que o segundo argumento é o endereço da variável a (e não o valor armazenado nessa variável). Ou seja, `&`a deve ser lido como "endereço de a", ou "referência para a. Esse assunto é abordado mais detalhadamente na seção 6.

A execução do comando *scanf* acima consiste em uma espera, até que um valor inteiro seja digitado no teclado (se o dispositivo de entrada padrão for o teclado, como ocorre se não houver redirecionamento do dispositivo de entrada padrão, como explicado na seção 3.2.1) seguido do caractere de terminação de linha (`Enter`), e no armazenamento do valor inteiro digitado na variável a.

Para ler dois valores inteiros e armazená-los em duas variáveis inteiras a e b, a função *scanf* pode ser usada como a seguir:

```
scanf("%d %d",&a, &b);
```

Podemos fazer agora nosso primeiro programa, que lê dois inteiros e imprime o maior dentre eles, como a seguir:

```
#include <stdio.h>

int max (int a, int b) {
  return (a>=b ? a : b);
}

int main() {
  int v1, v2;
  printf("Digite dois inteiros ");
  scanf("%d %d",&v1, &v2);
  printf("Maior dentre os valores digitados = ", max(v1,v2));
}
```

A primeira linha deste nosso primeiro programa contém uma *diretiva de preprocessamento*, que indica que o arquivo *stdio.h* deve ser lido e as definições contidas neste arquivo devem ser consideradas para compilação do restante do programa.

Além de "ler valores", isto é, além de modificar valores armazenados em variáveis, uma chamada à função *scanf* também retorna um valor. Esse valor é igual ao número de variáveis lidas, e pode ser usado para detectar fim dos dados de entrada a serem lidos (isto é, se não existe mais nenhum valor na entrada de dados a ser lido). No Capítulo 4 mostraremos exemplos de leitura de vários inteiros até que ocorra uma condição ou até que não haja mais valores a serem lidos.

Toda diretiva de preprocessamento começa com o caractere `#`, e deve ser inserida na primeira coluna de uma linha. A linguagem `C` permite usar espaços em branco e dispor comandos e declarações como o programador desejar (no entanto, programadores usam tipicamente convenções que têm o propósito de homegeneizar a disposição de trechos de programas de modo a facilitar a leitura). Uma diretiva de preprocessamento no entanto é uma exceção a essa regra de dispor livremente espaços, devendo começar sempre na primeira coluna de uma linha. Após o caractere `#` vem o nome do arquivo a ser lido, o qual deve ter uma extensão `.h`, entre os caracteres `<` e `>`, no caso de um arquivo de uma biblioteca padrão.

Para arquivos com extensão `.h` definido pelo programador, o nome do arquivo deve ser inserido entre aspas duplas (como por exemplo em `"interface.h"`, sendo `interface.h` o nome do arquivo).

A diretiva `#include <stdio.h>` é a diretiva mais comumente usada em programas `C`.

3.2.1 Entrada e Saída em Arquivos via Redirecionamento

É muitas vezes necessário ou mais adequado que os dados lidos por um programa estejam armazenados em arquivos, em vez de serem digitados repetidamente por um usuário em um teclado, e sejam armazenados em arquivos após a execução de um programa, permanecendo assim disponíveis após a execução desse programa.

Uma maneira simples de fazer entrada e saída em arquivos é através de *redirecionamento*, da entrada padrão no caso de leitura, ou da saída padrão no caso de impressão, para um arquivo. Em outras palavras, o redirecionamento da entrada padrão especifica que os dados devem ser lidos de um arquivo, em vez de a partir do teclado, e o redirecionamento da saída padrão especifica que os dados devem ser impressos em um arquivo, em vez de serem mostrados na tela do computador. A desvantagem desse esquema é que a entrada e saída de dados devem ser redirecionados para arquivos determinados antes da execução do programa, uma única vez.

As operações de entrada e de saída de dados *scanf* e *printf* funcionam normalmente, mas acontecem em arquivos, para os quais a entrada ou saída foi redirecionada no momento da chamada ao sistema operacional para iniciação do programa.

Para especificar o redirecionamento da entrada para um arquivo selecionado, o programa é iniciado com uma chamada que inclui, além do nome do programa a ser iniciado, o caractere `<` seguido do nome do arquivo de entrada que deve substituir o dispositivo de entrada padrão:

```
programa < arquivoDeEntrada
```

As operações de entrada de dados, que seriam feitas a partir do dispositivo de entrada padrão (usualmente o teclado), são realizadas então a partir do arquivo de nome `arquivoDeEntrada`.

O nome desse arquivo pode ser uma especificação completa de arquivo.

De modo similar, podemos redirecionar a saída padrão:

```
programa > arquivoDeSaida
```

Ao chamarmos `programa` dessa forma, a saída de dados vai ser feito em um arquivo que vai ser criado com o nome especificado, no caso `arquivoDeSaida`, em vez de a saída aparecer na tela do computador.

Podemos é claro fazer o redirecionamento tanto da entrada quanto da saída:

```
programa < arquivoDeEntrada > arquivoDeSaida
```

Um exemplo de iniciação de `programa.exe` a partir da interface de comandos do `DOS` com especificação completa de arquivos (em um computador com um sistema operacional Windows), para entrada de dados a partir do arquivo `c:\temp\dados.txt` e saída em um arquivo `c:\temp\saida.txt` deve ser feito como seguir:

```
programa.exe < c:\temp\dados.txt > c:\temp\saida.txt
```

3.2.2 Especificações de Formato

A letra que segue o caractere `%` em uma especificação de formato especifica qual conversão deve ser realizada. Várias letras podem ser usadas. As mais comuns são indicadas abaixo. Para cada uma é indicado o tipo do valor convertido:

Especificação de controle	Tipo
`%d` (ou `%i`)	`int`
`%c`	`char`
`%f`	`float`
`%lf`	`double`
`%s`	`char*`

`%e` pode também ser usado, para converter um valor em notação científica. Números na notação científica são escritos na forma $a \times 10^b$, onde a parte a é chamada de *mantissa* e b de *expoente*.

`%x` e `%o` podem ser usados para usar, respetivamente, notação hexadecimal e octal de cadeias de caracteres (dígitos), para conversão de um valor inteiro em uma cadeia de caracteres.

As letras `lf` em `%lf` são iniciais de `long float`.

`%d` e `%i` são equivalentes para saída de dados, mas são distintos no caso de entrada, com *scanf*. `%i` considera a cadeia de caracteres de entrada como hexadecimal quando ela é precedida de `"0x"`, e como octal quando precedida de `"0"`. Por exemplo, a cadeia de caracteres `"031"` é lida como `31` usando `%d`, mas como `25` usando `%i` ($25 = 3 \times 8 + 1$).

Em uma operação de saída, podem ser especificadas vários parâmetros de controle. A sintaxe de uma especificação de formato é bastante elaborada, e permite especificar:

”%`n.pt`”

onde `t` é uma letra que indica o tipo da conversão (que pode ser `d`, `i`, `c`, `s` etc., como vimos acima), e `n`, `p` especificam um tamanho, como explicado a seguir:

- Um valor inteiro `n` especifica um tamanho (número de caracteres) mínimo do valor a ser impresso: se o valor a ser impresso tiver um número de caracteres menor do que `n`, espaços são inseridos antes para completar `n` caracteres.

  Por exemplo, o programa:

```
#include <stdio.h>
int main() {
  printf("%3d\n",12345);
  printf("%3d\n",12);
}
```

imprime:

```
12345
 12
```

- Se `n` for o caractere `*`, isso indica que o tamanho mínimo é indicado como parâmetro de *scanf* antes do valor a ser impresso.

  Por exemplo, *printf* `("%*d", 5, 10)` imprime `"10"` com tamanho mínimo 5.

- Um valor inteiro `p` especifica um tamanho mínimo para o número de dígitos da parte fracionária de um número de ponto flutuante, no caso de valor numérico, e especifica o número máximo de caracteres impressos, no caso de cadeias de caracteres (fazendo com que uma cadeia de caracteres com um tamanho maior do que o tamanho máximo especificado seja truncada).

  Por exemplo, o programa:

```
#include <stdio.h>
int main() {
  printf("%3d\n",12345);
  printf("%3d\n",12);
  printf("%10.3f\n",12.34);
  printf("%10.3f\n",1234.5678);
  printf("%.3s", "abcde");
}
```

imprime:

```
12345
 12
    12.340
  1234.568
abc
```

- Se `p` for o caractere `*`, isso indica que o tamanho mínimo da parte fracionária de um valor de ponto flutuante, ou o tamanho máximo de uma cadeia de caracteres, é especificado como como parâmetro de *scanf* antes do valor a ser impresso.

  Por exemplo, *printf* `("%*s", 3, "abcde")` imprime `"abc"`.

Nota sobre uso do formato `%c` *com scanf em programas* `C` *executando sob o sistema operacional Windows, em entrada de dados interativa*:

O formato `%c` não deve ser usado com *scanf* em programas `C` executando sob o sistema operacional Windows; em vez disso, deve-se usar *getChar*. Para entender porque, considere as execuções dos dois programas a seguir, que vamos chamar de `ecoar1.c` e `ecoar2.c`:

ecoar1.c
```
#include <stdio.h>
int main () {
  char c;
  int fim = scanf("%c",&c);
  while (fim != EOF) {
    printf("%c",c);
    fim = scanf("%c",&c);
  }
  return 0;
}
```

ecoar2.c
```
#include <stdio.h>
int main() {
  int c = getchar();
  while (c != EOF) {
    printf("%c",c);
    c = getchar();
  }
  return 0;
}
```

Estranha e infelizmente, as execuções de `ecoar1.c` e `ecoar2.c` não são equivalentes em entrada *interativa*, no sistema operacional Windows: nesse caso, o valor retornado por *scanf*`("%c",&c)` só é igual a `-1` quando se pressiona `Control-Z` seguido de `Enter` duas vezes no início de uma linha. A razão para tal comportamento, bastante indesejável, é misteriosa, mas provavelmente trata-se de algum erro na imnplementação da função *scanf*.

3.3 Números

A área de memória reservada para armazenar um dado valor em um computador tem um tamanho fixo, por questões de custo e eficiência. Um valor de tipo `int` em `C` é usualmente armazenado, nos computadores atuais, em uma porção de memória com tamanho de 32 ou 64 bits, assim como um *número de ponto flutuante* — em `C`, um valor de tipo `float` ou `double`.

Também por questões de eficiência (isto é, para minimizar tempo ou espaço consumidos), existem qualificadores que podem ser usados para aumentar ou restringir os conjuntos de valores numéricos inteiros e de números de ponto flutuante que podem ser armazenados ou representados por um tipo numérico em `C`. Os qualificadores podem ser: `short` e `long`.

O tipo `char`, usado para representar caracteres, é também considerado como tipo inteiro, nesse caso sem sinal. O qualificador `unsigned` também pode ser usado para tipos inteiros, indicando que valores do tipo incluem apenas inteiros positivos ou zero.

A definição da linguagem `C` não especifica qual o tamanho do espaço alocado para cada variável de um tipo numérico específico (`char`, `short`, `int`, `long`, `float` ou `double`), ou seja, a linguagem não especifica qual o número de bits alocado. No caso de uso de um qualificador (`short` ou `long`), o nome `int` pode ser omitido.

Cada implementação da linguagem pode usar um tamanho que julgar apropriado. As únicas condições impostas são as seguintes. Elas usam a função `sizeof`, predefinida em `C`, que retorna o número de bytes de um nome de tipo, ou de uma expressão de um tipo qualquer:

$$\begin{array}{l}
\texttt{sizeof}(\texttt{short}) \leq \texttt{sizeof}(\texttt{int}) \leq \texttt{sizeof}(\texttt{long}) \\
\texttt{sizeof}(\texttt{short}) \geq 16 \text{ bits} \\
\texttt{sizeof}(\texttt{int}) \geq 16 \text{ bits} \\
\texttt{sizeof}(\texttt{long}) \geq 32 \text{ bits} \\
\texttt{sizeof}(\texttt{long long int}) \geq 64 \text{ bits} \\
\texttt{sizeof}(\texttt{float}) \leq \texttt{sizeof}(\texttt{double}) \leq \texttt{sizeof}(\texttt{long double})
\end{array}$$

Implementações usam comumente 32 bits para variáveis de tipo `int`, 64 bits para `long int` e 16 bits para variáveis de tipo `short int`.

Um numeral inteiro é do tipo `long` se ele tiver como sufixo a letra `L`, ou `l` (mas a letra `L` deve ser preferida, pois a letra `l` se parece muito com o algarismo `1`). Do contrário, o numeral é do tipo `int`. Não existem numerais do tipo `short`. Entretanto, um numeral do tipo `int` pode ser armazenado em uma variável do tipo `short`, ocorrendo, nesse caso, uma conversão implícita de `int` para `short`. Por exemplo, no comando de atribuição:

```
short s = 10;
```

o valor `10`, de tipo `int`, é convertido para o tipo `short` e armazenado na variável s, de tipo `short`. Essa conversão obtém apenas os n bits mais à direita (menos significativos) do valor a ser convertido, onde n é o número de bits usados para a representação de valores do tipo para o qual é feita a conversão, sendo descartados os bits mais à esquerda (mais significativos) restantes.

Números inteiros podem também ser representados nos sistemas de numeração hexadecimal e octal. No primeiro caso, o numeral deve ser precedido dos caracteres `0x` ou `0X`, sendo representado com algarismos hexadecimais: os números de 0 a 15 são representados pelos algarismos `0` a `9` e pelas letras de `a` até `f`, ou `A` até `F`, respectivamente. Um numeral octal é iniciado com o dígito `0` (seguido de um ou mais algarismos, de `0` a `7`).

Números de ponto flutuante são números representados com uma parte inteira (mantissa) e outra parte fracionária, como, por exemplo:

```
2.0        3.1415        1.5e-3        7.16e1
```

Um ponto decimal é usado para separar a parte inteira (mantissa) da parte fracionária. Um expoente na base `10` pode (ou não) ser especificado, sendo indicado pela letra `e`, ou `E`, seguida de um inteiro, opcionalmente precedido de um sinal (`+` ou `-`). Os dois últimos exemplos, que contêm também um expoente de 10, representam, respectivamente, 1.5×10^{-3} e 7.16×10^{1}.

Um sufixo, `f` ou `F`, como em `1.43f`, indica um valor do tipo `float`, e a ausência do sufixo, ou a especificação de um sufixo `d` ou `D`, indica um valor do tipo `double`.

Exemplos de numerais de tipo `float`:

```
2e2f        4.f        .5f        0f        2.71828e+4f
```

Exempos de numerais de tipo `double`:

```
2e2        4.        .5        0.0        1e-9d
```

3.3.1 Consequências de uma representação finita

Como números são representados em um computador com um número fixo de bits, a faixa de valores representáveis de cada tipo numérico é limitada. O uso de uma operação que retorna um valor positivo maior do que o maior inteiro representável em uma variável de tipo `int` constitui, em geral, um erro. Esse tipo de erro é chamado, em computação, de *overflow*, ou seja, espaço insuficiente para armazenamento.

Em `C`, um erro de *overflow* não é detectado, e o programa continua a sua execução: como o valor que causou a ocorrência de *overflow* não pode ser representado no espaço reservado para que ele seja armazenado, o programa usa então um outro valor (incorreto). Quando ocorre *overflow*, por exemplo, na adição de dois números inteiros positivos, o resultado é um número negativo. O inverso também é verdadeiro: quando ocorre *overflow* na adição de dois números inteiros negativos, o resultado é um número positivo.

3.4 Caracteres

Valores do tipo `char`, ou caracteres, são usados para representar símbolos (caracteres visíveis), tais como letras, algarismos etc., e caracteres de controle, usados para indicar fim de arquivo, mudança de linha, tabulação etc.

Cada caractere é representado, em um computador, por um determinado valor (binário). A associação entre esses valores e os caracteres correspondentes constitui o que se chama de *código*.

O código usado para representação de caracteres em `C` é o chamado código ASCII (*American Standard Code for Information Interchange*). O código ASCII é baseado no uso de 8 bits para cada caractere.

Caracteres visíveis são escritos em `C` entre aspas simples. Por exemplo: `'a'`, `'3'`, `'*'`, `' '`, `'%'` etc. É preciso notar que, por exemplo, o caractere '3' é diferente do numeral inteiro `3`. O primeiro representa um símbolo, enquanto o segundo representa um número inteiro.

Caracteres de controle e os caracteres `'` e \ são escritos usando uma sequência especial de caracteres. Por exemplo:

`'\n'`	indica *terminação de linha*
`'\t'`	indica *tabulação*
`'\''`	indica o caractere '
`'\\'`	indica o caractere \

Um valor do tipo `char` pode ser usado em `C` como um valor inteiro. Nesse caso, ocorre uma conversão de tipo implícita, tal como no caso de conversões entre dois valores inteiros de tipos diferentes (como, por exemplo, `short` e `int`). O valor inteiro de um determinado caractere é igual ao seu "código ASCII" (isto é, ao valor associado a esse caractere no código ASCII).

Valores de tipo `char` incluem apenas valores não-negativos. O tipo `char` é, portanto, diferente do tipo `short`, que inclui valores positivos e negativos. Conversões entre esses tipos, e conversões de tipo em geral, são abordadas na Seção 3.10.

As seguintes funções ilustram o uso de caracteres em `C`:

```
int minusc (char x) {
  return (x >= 'a') && (x <= 'z');
}

int maiusc (char x) {
  return (x >= 'A') && (x <= 'Z');
}

int digito (char x) {
  return (x >= '0') && (x <= '9');
}
```

Essas funções determinam, respectivamante, se o caractere passado como argumento é uma letra minúscula, uma letra maiúscula, ou um algarismo decimal (ou dígito).

A função a seguir ilustra o uso de caracteres como inteiros, usada na transformação de letras minúsculas em maiúsculas, usando o fato de que letras minúsculas e maiúsculas consecutivas têm (assim como dígitos) valores consecutivos no código ASCII:

```
char minusc_maiusc (char x) {
  int d = 'A' - 'a';
  if (minusc(x)) return (x+d);
  else return x;
}
```

Note que o comportamento dessa função não depende dos valores usados para representação de nenhuma letra, mas apenas do fato de que letras consecutivas têm valores consecutivos no código ASCII usado para sua representação. Por exemplo, a avaliação de:

minusc_maiusc `('b')`

tem como resultado o valor dado por `'b' + ('A' - 'a')`, que é igual a `'A' + 1`, ou seja, `'B'`.

Como mencionado na seção 3.4, caracteres podem ser expressos também por meio do valor da sua representação no código ASCII. Por exemplo, o caractere chamado *nulo*, que é representado com o valor `0`, pode ser denotado por:

`'\x0000'`

Nessa notação, o valor associado ao caractere no código ASCII ("código ASCII do caractere") é escrito na base hexadecimal (usando os algarismos hexadecimais `0` a `F`, que correspondem aos valores de 0 a 15, representáveis com 4 bits), precedido pela letra `x` minúscula.

```
const int NaoETriang = 0;
const int Equilatero   = 1;
const int Isosceles    = 2;
const int Escaleno     = 3;

int tipoTriang (int a, int b, int c) {
  if (eTriang(a,b,c))
     if (a == b && b == c) return Equilatero;
     else if (a == b || b == c || a == c) return Isosceles;
          else return Escaleno;
  else return NaoETriang;
}
```

Figura 3.2: Constantes em C

3.5 Constantes

Valores inteiros podem ser usados em um programa para representar valores de conjuntos diversos, segundo uma convenção escolhida pelo programador em cada situação. Por exemplo, para distinguir entre diferentes tipos de triângulos (equilátero, isósceles ou escaleno), podemos usar a convenção de que um triângulo equilátero corresponde ao inteiro `1`, isósceles ao `2`, escaleno ao `3`, e qualquer outro valor inteiro indica "não é um triângulo".

Considere o problema de definir uma função *tipoTriang* que, dados três números inteiros positivos, determina se eles podem formar um triângulo (com lados de comprimento igual a cada um desses valores) e, em caso positivo, determina se o triângulo formado é um triângulo equilátero, isósceles ou escaleno. Essa função pode ser implementada em `C` como na Figura 3.5.

Para evitar o uso de valores inteiros (`0`, `1`, `2` e `3`, no caso) para indicar, respectivamente, que os lados não formam um triângulo, ou que formam um triângulo equilátero, isósceles ou escaleno, usamos nomes correspondentes — *NaoETriangulo*, *Equilatero*, *Isosceles* e *Escaleno*. Isso torna o programa mais legível e mais fácil de ser modificado, caso necessário.

O atributo `const`, em uma declaração de variável, indica que o valor dessa variável não pode ser modificado no programa, permanecendo sempre igual ao valor inicial especificado na declaração dessa variável, que tem então que ser especificado. Essa "variável" é então, de fato, uma constante.

3.6 Enumerações

Em vez de atribuir explicitamente nomes a valores numéricos componentes de um mesmo tipo de dados, por meio de declarações com o atributo `const`, como foi feito na seção anterior, podemos declarar um tipo enumeração. Por exemplo, no caso das constantes que representam tipos de triângulos que podem ser formados, de acordo com o número de lados iguais, podemos definir:

```
enum TipoTriang { NaoETriang, Equilatero, Isosceles, Escaleno }
```

Por exemplo, o programa da seção anterior (Figura 3.5) pode então ser reescrito como mostrado na Figura 3.6. Em `C`, a palavra reservada `enum` deve ser usada antes do nome do tipo enumeração.

O uso de tipos-enumerações, como no programa da Figura 3.6, apresenta as seguintes vantagens, em relação ao uso de nomes de constantes (como no programa da Figura 3.5):

- O tipo-enumeração provê documentação precisa e confiável sobre o conjunto de valores possíveis do tipo.

  Por exemplo, no programa da Figura 3.6, o tipo do resultado da função *tipoTriang* é *TipoTriang*, em vez de `int`. Isso documenta o fato de que o valor retornado pela função é um

```
enum TipoTriang { NaoETriang, Equilatero, Isosceles, Escaleno };

enum TipoTriang tipoTriang (int a, int b, int c) {
  return (eTriang(a,b,c))              ?  NaoETriang :
          (a == b && b == c)           ?  Equilatero :
          (a == b || b == c || a == c) ?  Isosceles  :
          Escaleno);
}
```

Figura 3.3: Tipo-Enumeração

dos tipos de triângulo especificados na declaração do tipo *TipoTriang* (i.e. um dos valores *NaoETriang*, *Equilatero*, *Isosceles*, *Escaleno*).

- O valor de um tipo-enumeração é restrito aos valores definidos na declaração do tipo; assim, um erro é reportado pelo compilador se um valor diferente for explicitamente usado como um valor desse tipo.

  Por exemplo, no programa da Figura 3.6, é garantido que um literal usado para denotar o valor retornado pela função não é um inteiro qualquer, mas sim um dos valores definidos no tipo-enumeração.

3.7 Ordem de Avaliação de Expressões

Uma expressão em `C` é sempre avaliada da esquerda para a direita, respeitando-se contudo a precedência predefinida para operadores e a precedência especificada pelo uso de parênteses.

O uso de parênteses em uma expressão é, em alguns casos, opcional, apenas contribuindo para tornar o programa mais legível. Por exemplo, a expressão `3+(5*4)` é equivalente a `3+5*4`, uma vez que o operador de multiplicação (`*`) tem maior precedência do que o de adição (`+`) (o que resulta na avaliação da multiplicação antes da adição). Outro exemplo é `3<5*4`, que é equivalente a `3<(5*4)`.

Em outros casos, o uso de parênteses é necessário, tal como, por exemplo, na expressão `3*(5+4)` — a ausência dos parênteses, nesse caso, mudaria o resultado da expressão (pois `3*5+4` fornece o mesmo resultado que `(3*5)+4`).

Os operadores aritméticos em `C` são mostrados na Tabela 3.2. A precedência desses operadores, assim como de outros operadores predefinidos usados mais comumente, é apresentada na Tabela 3.3. O estabelecimento de uma determinada precedência entre operadores tem como propósito reduzir o número de parênteses usados em expressões.

Note que a divisão de um valor inteiro (de tipo `int`, com ou sem qualificadores) por outro valor inteiro, chamada em computação de *divisão inteira*, retorna em `C` o quociente da divisão (a parte fracionária, se existir, é descartada). Veja o exemplo da Tabela 3.2: `5/2` é igual a `2`. Para obter um resultado com parte fracionária, é necessário que pelo menos um dos argumentos seja um valor de ponto flutuante.

O programa a seguir ilustra que não é uma boa prática de programação escrever programas que dependam de forma crítica da ordem de avaliação de expressões, pois isso torna o programa mais difícil de ser entendido e pode originar resultados inesperados, quando esse programa é modificado.

A execução do programa abaixo imprime `4` em vez de `20`. Note como o resultado depende criticamente da ordem de avaliação das expressões, devido ao uso de um comando de atribuição como uma expressão. Nesse caso, o comando de atribuição (*i*=`2`) é usado como uma expressão (que retorna, nesse caso, o valor `2`), tendo como "efeito colateral" modificar o valor armazenado na variável *i*.

Tabela 3.2: Operadores aritméticos em `C`

Operador	**Significado**	**Exemplo**	**Resultado**
`+`	Adição	`2 + 1`	`3`
		`2 + 1.0`	`3.0`
		`2.0 + 1.0`	`3.0`
`-`	Subtração	`2 - 1`	`1`
		`2.0 - 1`	`1.0`
		`2.0 - 1.0`	`1.0`
`-`	Negação	`-1`	`-1`
		`-1.0`	`-1.0`
`*`	Multiplicação	`2 * 3`	`6`
		`2.0 * 3`	`6.0`
		`2.0 * 3.0`	`6.0`
`/`	Divisão	`5 / 2`	`2`
		`5 / 2.0`	`2.5`
		`5.0 / 2.0`	`2.5`
`%`	Resto	`5 % 2`	`1`
		`5.0 % 2.0`	`1.0`

Tabela 3.3: Precedência de operadores

<table>
<tr><th colspan="2">Precedência maior</th></tr>
<tr><th>Operadores</th><th>Exemplos</th></tr>
<tr><td>*, /, %</td><td>i * j / k ≡ (i * j) / k</td></tr>
<tr><td>+, -</td><td>i + j * k ≡ i + (j*k)</td></tr>
<tr><td>>, <, >=, <=</td><td>i < j + k ≡ i < (j+k)</td></tr>
<tr><td>==, !=</td><td>b == i < j ≡ b == (i < j)</td></tr>
<tr><td>&</td><td>b & b1 == b2 ≡ b & (b1 == b2)</td></tr>
<tr><td>^</td><td>i ^ j & k ≡ i ^ (j & k)</td></tr>
<tr><td>|</td><td>i < j | b1 & b2 ≡ (i < j) | (b1 & b2)</td></tr>
<tr><td>&&</td><td>i + j != k && b1 ≡ ((i + j) != k) && b1</td></tr>
<tr><td>||</td><td>i1 < i2 || b && j < k ≡ (i1<i2) || (b && (j<k))</td></tr>
<tr><td>? :</td><td>i < j || b ? b1 : b2 ≡ ((i<j) || b) ? b1 : b2</td></tr>
<tr><th colspan="2">Precedência menor</th></tr>
</table>

Tabela 3.4: Operadores lógicos em `C`

Operador	Significado
`!`	Negação
`&&`	Conjunção (não-estrita) ("ê")
`||`	Disjunção (não-estrita) ("ou")

```
int main () {
  int i = 10;
  int j = (i=2) * i;
  printf("%d",j);
}
```

3.8 Operações lógicas

Existem operadores lógicos (também chamados de operadores booleanos) — que operam com valores falso ou verdadeiro, representados em `C` como inteiros zero e diferente de zero — predefinidos em `C`, que são bastante úteis. Eles estão relacionados na Tabela 3.4 e são descritos a seguir.

A operação de negação (ou complemento) é representada por `!`; ela realiza a negação do argumento (de modo que um valor falso (zero) fique verdadeiro (diferente de zero), e o inverso também é verdadeiro: a negação de um valor verdadeiro em `C` (diferente de zero) é falso (zero em `C`).

A operação de conjunção lógica é representada por `&&` (lê-se "ê"): e_1 `&&` e_2 é verdadeiro (diferente de zero) se somente se a avaliação de cada uma das expressões, e_1 e e_2 tem como resultado o valor verdadeiro.

O operador `&&` é um operador não-estrito, no segundo argumento; isso significa que a avaliação de e_1 `&&` e_2 pode retornar um resultado verdadeiro mesmo que a avaliação de e_2 não retorne nenhum valor válido; por exemplo:

```
0 && (0/0 == 0)
```

retorna falso (`0`).

Note que o operador de conjunção bit-a-bit, `&`, é um operador estrito (nos dois argumentos), ou seja, a avaliação de e_1 `&` e_2 retorna um resultado se e somente se a avaliação de e_1 e de e_2 retornam. Por exemplo,

```
0 & (0/0 == 0)
```

provoca a ocorrência de um erro (pois `0/0` provoca a ocorrência de um erro).

A operação de conjunção não-estrita é definida como a seguir: o valor de e_1 `&&` e_2 é igual ao de e_1 se esse for falso (`0`); caso contrário, é igual ao valor de e_2. Dessa forma, e_2 só é avaliado se a avaliação de e_1 fornecer valor verdadeiro (diferente de zero).

Ao contrário, a avaliação de e_1 `&` e_2 sempre envolve a avaliação tanto de e_1 quanto de e_2.

Analogamente, a operação de disjunção lógica é representada por `||` (lê-se "ou"): e_1 `||` e_2 é igual a falso somente se a avaliação de cada uma das expressões, e_1 e e_2, tem como resultado o valor falso.

Observações análogas às feitas anteriormente, para os operadores de conjunção bit-a-bit `&` e conjunção lógica não-estrita `&&`, são válidas para os operadores de disjunção bit-a-bit `|` e disjunção lógica não-estrita `||`.

Nesse caso, temos que e_1 `||` e_2 é igual a verdadeiro (diferente de zero) se e_1 for igual a verdadeiro, e igual a e_2 em caso contrário.

Existe ainda predefinido em `C` o operador ou-exclusivo bit-a-bit `^`.

3.9 Programas e Bibliotecas

Um programa en `C` consiste em uma sequência de uma ou mais definições de funções, sendo que uma dessas funções, de nome *main*, é a função que inicia a execução do programa. A definição de uma função de nome *main* deve sempre estar presente, para que uma sequência de definições de funções forme um programa `C`.

A *assinatura* — ou interface — de uma função é uma definição de uma função que omite o corpo (sequência de comandos que é executada quando a função é chamada) da função, mas especifica o nome, o tipo de cada argumento e do resultado da função.

Em uma assinatura, os nomes dos argumentos são opcionais (mas os tipos são necessários).

A assinatura da função *main* de um programa `C` deve ser:

`int` *main*`(void)`

ou

`int` *main*`(int` *argc*`, char*` *argv*`[])`

A primeira assinatura especifica que a função *main* não tem parâmetros e o tipo do resultado é `int`. Esse valor é usado para especificar, para o sistema operacional, que a função foi executada normalmente, sem causar nenhum erro (nesse caso, o valor zero é retornado), ou para especificar que a execução da função provocou a ocorrência de algum erro (nesse caso, um valor diferente de zero é retornado). Como o sistema operacional pode usar uma convenção diferente para indicar a presença ou ausência de erro, é boa prática usar as constantes *EXIT_SUCCESS* e *EXIT_FAILURE*, definidos na biblioteca *stdlib*, para indicar respectivamente sucesso e falha da execução.

A linguagem `C` permite o uso de funções de bibliotecas, que requerem o uso de *diretivas de preprocessamento*. Como mencionado na seção 3.2, Uma diretiva de preprocessamento começa com o caractere `#`, e deve ser inserida na primeira coluna de uma linha. Uma diretiva de preprocessamento deve começar sempre na primeira coluna de uma linha. Após o caractere `#` vem o nome do arquivo a ser lido, o qual deve ter uma extensão `.h`, entre os caracteres `<` e `>`, no caso de um arquivo de uma biblioteca padrão. Para arquivos com extensão `.h` definido pelo programador, o nome do arquivo deve ser inserido entre aspas duplas (como por exemplo em `"interface.h"`, sendo `interface.h` o nome do arquivo).

A diretiva `#include <`*stdio.h*`>` é a diretiva mais comumente usada em programas `C`.

Existem muitas funções definidas em bibliotecas da linguagem `C`, e a questão, bastante comum para iniciantes no aprendizado de programação em uma determinada linguagem, de quando existe ou não uma função definida em uma biblioteca, só pode ser respondida em geral por experiência ou pesquisa em textos sobre a linguagem e as bibliotecas específicas. As bibliotecas mais comuns usadas em programas `C` são as seguintes (são incluídas a assinatura e uma descrição sucinta de algumas funções de cada biblioteca):

- *stdio*: contém funções para entrada e saída em dispositivos padrão, como *printf* e *scanf* (descritas na seção 3.2).
- *string*: contém funções para manipulação de *strings* (seção 5.5).
- *ctype.h*: contém funções sobre valores de tipo `char`, como:
  - `int` *isdigit*`(char)`: retorna verdadeiro (diferente de zero) se o argumento é um dígito (de `'0'` a `'9'`),
  - `int` *isalpha*`(char)`: retorna verdadeiro se o argumento é uma letra,
  - `int` *isalnum*`(char)`: retorna verdadeiro se o argumento é uma letra ou um dígito,
  - `int` *islower*`(char)`: retorna verdadeiro se o argumento é uma letra minúscula,
  - `int` *isupper*`(char)`: retorna verdadeiro se o argumento é uma letra maiúscula
- *math*: contém funções matemáticas, como:
  - `double` *cos*`(double)`: retorna o cosseno do argumento,
  - `double` *sen*`(double)`: retorna o seno do argumento,

  - `double` *exp*`(double)`: retorna e elevado ao argumento, onde e é a base do logarítmo natural (constante de Euler),
  - `double` *fabs*`(double)`: retorna o valor absoluto do argumento,
  - `double` *sqrt*`(double)`: retorna a raiz quadrada do argumento,
  - `double` *pow*`(double` a`, double` b`)`: retorna a^b (a elevado a b),
  - `double` *pi*: uma representação aproximada do número irracional π.

- *stdlib*: contém funções diversas, para diversos fins, como:
  - `int` *abs*`(int)`: retorna o valor absoluto do argumento,
  - `long` *labs*`(long)`: retorna o valor absoluto do argumento,
  - `void` *srand*`(unsigned int)`: especifica a semente do gerador de números pseudo-aleatórios *rand*,
  - `int` *rand*`()`: retorna um número pseudo-aleatório entre 0 e *RAND_MAX*,
  - `int` *RAND_MAX*: valor maior ou igual a 32767, usado pelo gerador de números pseudo-aleatórios *rand*,
  - *atoi*, *atol*, *atof* convertem, respectivamente, *string* para valor de tipo `int`, `long int` e `double` (veja seção 5.5),
  - *malloc* aloca memória dinamicamente (seção 6).
  - `int` *EXIT_SUCCESS*: valor usado para indicar que a execução de um programa ocorreu com sucesso,
  - `int` *EXIT_FAILURE*: valor usado para indicar que ocorreu alguma falha na execução de um programa.

Na seção 3.9 mostraremos como um programa em `C` pode ser dividido em várias unidades de compilação, cada uma armazenada em um arquivo, e como nomes definidos em uma unidade de compilação são usados em outra unidade.

3.10 Conversão de Tipo

Conversões de tipo podem ocorrer, em `C`, nos seguintes casos:

- implicitamente: em atribuições e passagem de argumentos a funções, dependendo dos tipos envolvidos (isto é, dependendo de se, respectivamente, o tipo da expressão e em um comando de atribuição v `=` e é diferente do tipo da variável v, ou se o tipo do argumento é diferente do tipo do parâmetro correspondente em uma chamada de função);
- explicitamente: em conversões de tipo (em inglês, *type casts*).

Em geral há converesão automática (implícita) quando um valor é de um tipo numérico t_0 que tem um conjunto de valores contido em outro tipo numérico t. Neste caso, o valor de tipo t_0 é convertido automaticamente, sem alteração, para um valor de tipo t. Isso ocorre, por exemplo, quando t_0 é `char` ou `short` ou `byte` e t é igual a `int`.

Uma conversão de tipo explícita tem a seguinte sintaxe:

$$(t)\ e$$

onde t é um tipo e e uma expressão. A conversão de tipo indica que o valor resultante da avaliação da expressão e deve ser convertido para o tipo t. Em geral, a conversão de tipo pode resultar em alteração de um valor pelo truncamento de bits mais significativos (apenas os bits menos significativos são mantidos), no caso de conversão para um tipo t cujos valores são representados por menos bits do que valores do tipo da expressão e. No caso de uma conversão para um tipo t cujos valores são representados por mais bits do que valores do tipo da expressão e, há uma extensão do bit mais à esquerda do valor de e para completar os bits da representação desse valor como um valor do tipo t.

3.11 Exercícios Resolvidos

1. Escreva um programa que leia três valores inteiros e imprima uma mensagem que indique se eles podem ou não ser lados de um triângulo e, em caso positivo, se o triângulo formado é equilátero, isósceles ou escaleno.

Solução:

```
enum TipoTriang { NaoETriang, Equilatero, Isosceles, Escaleno };
enum TipoTriang tipoTriang (int a, int b, int c) {
  return (eTriang(a,b,c))              ?  NaoETriang :
          (a == b && b == c)           ?  Equilatero :
          (a == b || b == c || a == c) ?  Isosceles :
          Escaleno);
}

char* show(enum TipoTriang t) {
  return (t==Equilatero ?  "equilatero":
          t==Isosceles  ?  "isosceles":
          "escaleno");
}

int main() {
  int a, b, c;
  printf("Digite 3 valores inteiros:  ");
  scanf("%d %d %d", &a, &b, &c);
  enum Tipotriang t = tipoTriang(a, b, c);
  printf("Os valores digitados ");
  if (t == NaoETriang) printf("nao podem formar um triangulo\n");
  else printf("formam um triangulo %s\n", show(t));
}
```

O valor retornado pela função *show* é um *string*. Em `C`, um *string* é um ponteiro para um valor de tipo `char` (ou, equivalentemente em `C`, um arranjo de caracteres). Isso será explicado mais detalhadamente na seção 5.5.

2. Simplifique a definição da função *tipoTriang* de modo a utilizar um menor número de operações (de igualdade e de disjunção), supondo que, em toda chamada a essa função, os argumentos correspondentes aos parâmetros a, b e c são passados em ordem não-decrescente.

3. Como vimos na Seção 3.8, a avaliação da expressão `0 && 0/0 == 0` tem resultado diferente da avaliação da expressão `0 & 0/0 == 0`. Construa duas expressões análogas, usando `||` e `|`, para as quais a avaliação fornece resultados diferentes.

Solução:

```
1 ||  0/0 == 0
1 |   0/0 == 0
```

A avaliação da primeira expressão fornece resultado verdadeiro, e a avaliação da segunda provoca a ocorrência de um erro.

4. Defina os operadores lógicos `&&` e `||` usando uma expressão condicional.

Solução: Para quaisquer expressões booleanas a e b, a operação de conjunção a `&&` b tem o mesmo comportamento que:

```
a ? b : 0
```

Note que o valor da expressão é falso quando o valor de a é falso, independentemente do valor de b (ou mesmo de a avaliação de b terminar ou não, ou ocasionar ou não um erro).

Antes de ver a resposta abaixo, procure pensar e escrever uma expressão condicional análoga à usada acima, mas que tenha o mesmo comportamento que *a* `||` *b*.

A expressão desejada (que tem o mesmo significado que *a* `||` *b*) é:

```
a ? 1 : b
```

3.12 Exercícios

1. Sabendo que a ordem de avaliação de expressões em `C` é da esquerda para a direita, respeitando contudo a precedência de operadores e o uso de parênteses, indique qual é o resultado da avaliação das seguintes expressões (consulte a Tabela 3.3, se necessário):

(a) `2 + 4 - 3`

(b) `4 - 3 * 5`

(c) `(4 - 1) * 4 - 2`

(d) `2 >= 1 && 2 != 1`

2. A função *eQuadrado*, definida a seguir, recebe como argumentos quatro valores inteiros e retorna `true` se esses valores podem formar os lados de um quadrado, e `false` em caso contrário.

```
int eQuadrado (int a,int b,int c,int d) {
  // todos os valores são positivos, e iguais entre si
  return (a>0) && (b>0) && (c>0) && (d>0) &&
         (a==b && b==c && c==d);
}
```

Escreva um programa que leia quatro valores inteiros e imprima uma mensagem indicando se eles podem ou não os lados de um retângulo, usando função que, dados quatro números inteiros, retorna verdadeiro se eles podem representar lados de um retângulo, e falso em caso contrário.

3. A seguinte definição de função determina se um dado valor inteiro positivo representa um ano bissexto ou não. No calendário gregoriano, usado atualmente, um ano é bissexto se for divisível por 4 e não for divisível por 100, ou se for divisível por 400.

```
int bissexto (int ano) {
  return ((ano % 4 == 0 && ano % 100 != 0)
          || ano % 400 == 0 );
}
```

Reescreva a definição de *bissexto* de maneira a usar uma expressão condicional em vez de usar, como acima, os operadores lógicos `&&` e `||`.

Escreva um programa que leia um valor inteiro e responda se ele é ou não um ano bissexto, usando a função definida acima com a expressão condicional.

4. Defina uma função *somaD3* que, dado um número inteiro representado com até três algarismos, fornece como resultado a soma dos números representados por esses algarismos. Exemplo: *somaD3*`(123)` deve fornecer resultado `6`.

   Escreva um programa que leia um valor inteiro, e imprima o resultado da soma dos algarismos do número lido, usando a função *somaD3*. Note que você pode supor que o inteiro lido contém no máximo 3 algarismos (o seu programa deve funcionar corretamente apenas nesses casos).

5. Defina uma função *inverteD3* que, dado um número representado com até três algarismos, fornece como resultado o número cuja representação é obtida invertendo a ordem desses algarismos. Por exemplo: o resultado de *inverteD3*`(123)` deve ser `321`.

   Escreva um programa que leia um valor inteiro, e imprima o resultado de inverter os algarismos desse valor, usando a função *inverteD3*. Note que você pode supor que o inteiro lido contém no máximo 3 algarismos (o seu programa deve funcionar corretamente apenas nesses casos).

6. Considere a seguinte definição, que associa a *pi* o valor `3.1415`:

   ```
   final float pi = 3.1415f;
   ```

   Use essa definição do valor de *pi* para definir uma função que retorna o comprimento aproximado de uma circunferência, dado o raio.

   Escreva um programa que leia um número de ponto flutuante, e imprima o comprimento de uma circunferência que tem raio igual ao valor lido, usando a função definida acima. Por simplicidade, você pode supor que o valor lido é um número de ponto flutuante algarismos (o seu programa deve funcionar corretamente apenas nesse caso).

7. Defina uma função que, dados cinco números inteiros, retorna verdadeiro (inteiro diferente de zero) se o conjunto formado pelos 2 últimos números é um subconjunto daquele formado pelos 3 primeiros, e falso em caso contrário.

   Escreva um programa que leia 5 valores inteiros, e imprima o resultado de determinar se o conjunto formado pelos 2 últimos é um subconjunto daquele formado pelos três primeiros, usando a função definida acima. Por simplicidade, você pode supor que os valores lidos são todos inteiros (o seu programa deve funcionar corretamente apenas nesse caso).

8. Defina uma função que, dado um valor inteiro não-negativo que representa a nota de um aluno em uma disciplina, retorna o caractere que representa o conceito obtido por esse aluno nessa disciplina, de acordo com a tabela:

Nota	**Conceito**
0 a 59	R
60 a 74	C
75 a 89	B
90 a 100	A

   Escreva um programa que leia um valor inteiro, e imprima a nota correspondente, usando a função definida acima. Por simplicidade, você pode supor que o valor lido é um valor inteiro (o seu programa deve funcionar corretamente apenas nesse caso).

9. Defina uma função que, dados dois caracteres, cada um deles um algarismo, retorna o maior número inteiro que pode ser escrito com esses dois algarismos. Você não precisa considerar o caso em que os caracteres dados não são algarismos.

   Escreva um programa que leia 2 caracteres, cada um deles um algarismo, e imprima o maior número inteiro que pode ser escrito com esses dois algarismos. Por simplicidade, você pode supor que os valores lidos são de fato um caractere que é um algarismo (o seu programa deve funcionar corretamente apenas nesse caso).

10. Escreva uma função que, dados um número inteiro e um caractere — representando respectivamente a altura e o sexo de uma pessoa, sendo o sexo masculino representado por `'M'` ou `'m'` e o sexo feminino representado por `'F'` ou `'f'` —, retorna o peso supostamente ideal para essa pessoa, de acordo com a tabela:

homens	mulheres
$(72,7 \times altura) - 58$	$(62,1 \times altura) - 44,7$

Escreva um programa que leia um número inteiro e um caractere, que você pode supor que sejam respectivamente o peso de uma pessoa e um dos caracteres dentre `'M'`, `'m'`, `'F'`, `'f'`, e imprima o peso ideal para uma pessoa, usando a função definida acima. Você pode supor que os dados de entrada estão corretos (o seu programa deve funcionar corretamente apenas nesse caso).

Capítulo 4

Recursão e Iteração

Os conceitos de recursão e iteração constituem, juntamente com as noções de composição seqüencial e seleção, as ferramentas fundamentais para construção de algoritmos e programas, a partir de um conjunto apropriado de operações ou comandos básicos. Esses dois conceitos são introduzidos neste capítulo, por meio de uma série de exemplos ilustrativos, de construção de programas para cálculo de operações aritméticas simples, tais como:

- cálculo da multiplicação de dois números inteiros, expressa em termos da operação de adição;
- cálculo do resultado da operação de exponenciação, com um expoente inteiro, expressa em termos da operação de multiplicação;
- cálculo do fatorial de um número inteiro;
- cálculo de somas de séries numéricas.

A solução de qualquer problema que envolva a realização de uma ou mais operações repetidas vezes pode ser expressa, no paradigma de programação imperativo, por meio de um comando de repetição (também chamado de comando iterativo, ou comando de iteração), ou usando funções com definições recursivas.

Definições recursivas de funções são baseadas na mesma idéia subjacente a um princípio de prova fundamental em matemática — o *princípio da indução*. A idéia é a de que a solução de um problema pode ser expressa da seguinte forma: primeiramente, definimos a solução desse problema para casos básicos; em seguida, definimos como resolver o problema para os demais casos, em termos da solução para casos mais simples que o original.

4.1 Multiplicação e Exponenciação

Considere por exemplo o problema de definir a multiplicação de dois números inteiros m e n, sendo $n > 0$, em termos da operação de adição. O caso mais simples dessa operação é aquele no qual $n = 0$: nesse caso, o resultado da operação é igual a 0, independentemente do valor de m. De acordo com a idéia exposta acima, precisamos agora pensar em como podemos expressar a solução desse problema no caso em que $n > 0$, supondo que sabemos determinar sua solução para casos mais simples, que se aproximam do caso básico. Neste caso, podemos expressar o resultado de $m \times n$ em termos do resultado da operação, mais simples $m \times (n-1)$; ou seja, podemos definir $m \times n$ como sendo igual a $m + (m \times (n-1))$, para $n > 0$.

Ou seja, a operação de multiplicação pode então ser definida indutivamente pelas seguintes equações:

$$\begin{aligned} m \times 0 &= 0 \\ m \times n &= m + (m \times (n-1)) \quad \text{se } n > 0 \end{aligned}$$

Uma maneira alternativa de pensar na solução desse problema seria pensar na operação de multiplicação $m * n$ como a repetição, n vezes, da operação de adicionar m, isto é:

$$m \times n = \underbrace{m + \ldots + m}_{n \text{ vezes}}$$

```
/****************************************************
int mult (int m, int n) {
  int r=0, i;
  for (i=1; i<=n; i++) r += m;
  return r;
}

int multr (int m, int n) {
  if (n==0) return 0;
  else return (m + multr(m, n-1));
}

int exp (int m, int n) {
  int r=1, i;
  for (i=1; i<=n; i++) r*=m;
  return r;
}

int expr (int m, int n) {
  if (n==0) return 1;
  else return (m * expr(m, n-1));
}
****************************************************/
```

Figura 4.1: Primeiros exemplos de definições recursivas e baseadas em comandos de repetição

Raciocínio análogo pode ser usado para expressar a operação de exponenciação, com expoente inteiro não-negativo, em termos da operação de multiplicação.

A Figura 4.1 apresenta duas definições alternativas, na linguagem `C`, para computar o resultado da multiplicação e da exponenciação de dois números, dados como argumentos dessas operações. As funções *mult* e *exp* são definidas usando-se um comando de repetição. As funções *multr* e *expr* são definidas usando-se chamadas à própria função que está sendo definida, razão pela qual essas definições são chamadas de *recursivas*.

Usamos a letra "*r*" adicionada como sufixo ao nome de uma função para indicar que se trata de uma versão recursiva da definição de uma função.

Como vimos na Seção 2.4, a execução de um comando de repetição consiste na execução do corpo desse comando repetidas vezes, até que a condição associada a esse comando se torne falsa (caso isso nunca ocorra, a repetição prossegue indefinidamente). Para entender melhor o significado desse comando e, portanto, o comportamento das funções *mult* e *exp*, consideramos seguir a execução de uma chamada à função *mult* — por exemplo, *mult*`(3,2)`.

Note que essas definições são usadas apenas com fins didáticos, para explicar inicialmente definições recursivas e comandos de repetição. Essas definições não têm utilidade prática pois existe, no caso da multiplicação, o operador predefinido (`*`) e, no caso da exponenciação, a função *math.pow* (cf. seção 3.9).

A execução de uma chamada de função é iniciada, como vimos na Seção 2.5, com a avaliação dos argumentos a serem passados à função: no exemplo acima, essa avaliação fornece os valores representados pelos *literais* `3` e `2`, do tipo `int`. Esses valores são atribuídos aos parâmetros m e n, respectivamente, e o corpo da função *mult* é então executado.

Depois da execução do corpo da função *mult*, os parâmetros, assim como, possivelmente, variáveis declaradas internamente no corpo da função, deixam de existir, isto é, são desalocados. Ou seja, o *tempo de vida* de parâmetros e variáveis declaradas localmente é determinado pelo bloco correspondente à função.

A execução do corpo da função *mult* é iniciada com a criação da variável r, à qual é atribuído inicialmente o valor `0`. Em seguida, o comando `for` é executado.

Tabela 4.1: Passos na execução de *mult*`(3,2)`

Comando/ Expressão	**Resultado** (expressão)	**Estado** (após execução/avaliação)
$\boxed{mult\texttt{(3,2)}}$	...	$m \mapsto 3$, $n \mapsto 2$
`int` r `= 0`		$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 0$
i `= 1`		$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 0$, $i \mapsto 1$
i `<=` n	verdadeiro	$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 0$, $i \mapsto 1$
r `+=` m	3	$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 3$, $i \mapsto 1$
i `++`	2	$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 3$, $i \mapsto 2$
i `<=` n	verdadeiro	$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 3$, $i \mapsto 2$
r `+=` m	6	$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 6$, $i \mapsto 2$
i `++`	3	$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 6$, $i \mapsto 3$
i `<=` n	falso	$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 6$, $i \mapsto 3$
`for ...`		$m \mapsto 3$, $n \mapsto 2$, $r \mapsto 6$, $i \mapsto 3$
`return` r		
mult`(3,2)`	6	

O comando `for` é um comando de repetição (assim como o comando `while`, introduzido na Seção 2.4). A execução do comando `for` usado no corpo da função *mult* consiste simplesmente em adicionar (o valor contido em) m ao valor de r, a cada iteração, e armazenar o resultado dessa adição em it r. O número de iterações executadas é igual a n. Por último, o valor armazenado em r após a execução do comando `for` é retornado, por meio da execução do comando "`return` r`;`". Consideramos, a seguir, o comando `for` mais detalhadamente.

O cabeçalho de um comando `for` — no exemplo, a parte entre parênteses `(int` i`=1;` i`<=n;` i`++)` — é constituído de três partes (ou cláusulas), descritas a seguir:

- a primeira parte, de *"inicialização"*, atribui valores iniciais a variáveis, antes do início das iterações. Essa parte de inicialização só é executada uma vez, antes do início das iterações.
- a segunda parte, chamada de *teste de terminação*, especifica uma expressão booleana que é avaliada antes do início de cada iteração (inclusive antes da execução do corpo do comando `for` pela primeira vez). Se o valor obtido pela avaliação dessa expressão for verdadeiro (em `C`, diferente de zero), então o corpo do comando `for` é executado, seguido pela avaliação da terceira cláusula do cabeçalho desse comando, e depois uma nova iteração é iniciada; caso contrário, ou seja, se o valor for falso (em `C`, zero), a execução do comando `for` termina.
- a terceira parte, de *atualização*, contém expressões que têm o objetivo de modificar o valor de variáveis, depois da execução do corpo do comando `for`, a cada iteração. No nosso exemplo, "i`++`", que representa o mesmo que "i `=` i`+1`", incrementa o valor de i a cada iteração.

Note a ordem na execução de cada iteração em um comando `for`: teste de terminação, corpo, atualização.

É instrutivo acompanhar a modificação do valor armazenado em cada variável durante a execução da chamada *mult*`(3,2)`. Em outras palavras, acompanhar, em cada passo da execução, o *estado* da computação: o estado da computação é uma função que associa um valor a cada variável.

A Tabela 4.1 ilustra os passos da execução da chamada *mult*`(3,2)`, registrando, a cada passo, o estado da computação. O resultado fornecido pela avaliação de expressões é também mostrado nessa tabela. Note que o resultado fornecido pela cláusula de atualização de um comando `for` é sempre descartado. Essa expressão é avaliada apenas com o propósito de atualizar o valor de uma ou mais variáveis (ou seja, é uma expressão com efeito colateral).

A variável i é usada no comando `for` como um *contador de iterações*.

Considere agora a função *multr* (Figura 4.1). A definição de *multr* espelha diretamente a definição indutiva da operação de multiplicação, em termos da adição, apresentada anteriormente. Ou seja, multiplicar m por n (onde n é um inteiro não-negativo) fornece:

- `0`, no caso base (isto é, se n`==0`);

- m `+` $mult$`(`m`,`n`-1)`, no caso indutivo (isto é, se n`!=0`).

Vejamos agora, mais detalhadamente, a execução de uma chamada $multr$`(3,2)`. Cada chamada da função *multr* cria novas variáveis, de mesmo nome m e n. Existem, portanto, várias variáveis com nomes (m e n), devido às chamadas recursivas. Nesse caso, o uso do nome refere-se à variável local ao corpo da função que está sendo executado. As execuções das chamadas de funções são feitas, dessa forma, em uma estrutura de pilha. Chamamos, genericamente, de *estrutura de pilha* uma estrutura na qual a inserção (ou alocação) e a retirada (ou liberação) de elementos é feita de maneira que o último elemento inserido é o primeiro a ser retirado.

Como a execução de chamadas de funções é feita na forma de uma estrutura de pilha, em cada instante da execução de um programa, o último conjunto de variáveis alocados na pilha corresponde às variáveis e parâmetros da última função chamada. No penúltimo espaço são alocadas as variáveis e parâmetros da penúltima função chamada, e assim por diante. Cada espaço de variáveis e parâmetros alocado para uma função é chamado de *registro de ativação* dessa função.

O registro de ativação de cada função é desalocado (isto é, a área de memória correspondente na pilha é liberada para uso por outra chamada de função) no instante em que a execução da função termina. Como o término da execução da última função chamada precede o término da execução da penúltima, e assim por diante, o processo de alocação e liberação de registros de ativação de chamadas de funções se dá como em uma estrutura de pilha.

As variáveis que podem ser usadas no corpo de uma função são apenas os parâmetros dessa função e as variáveis internas aí declaradas (chamadas de variáveis locais da função), além das variáveis externas, declaradas no mesmo nível da definição da função, chamadas de *variáveis globais*.

Variáveis declaradas internamente a uma função são criadas cada vez que uma chamada a essa função é executada, enquanto variáveis globais são criadas apenas uma vez, quando a execução do programa é iniciada ou anteriormente, durante a carga do código do programa na memória.

No momento da criação de uma variável local a uma função, se não foi especificado explicitamente um valor inicial para essa variável, o valor nela armazenado será indeterminado, isto é, será o valor já existente nos bytes alocados para essa variável na *pilha* de chamadas de funções.

Na Figura 4.2, representamos a estrutura de pilha criada pelas chamadas à função *multr*. Os nomes m e n referem-se, durante a execução, às variáveis mais ao topo dessa estrutura. Uma chamada recursiva que vai começar a ser executada é circundada por uma caixa. Nesse caso, o resultado da expressão, ainda não conhecido, é indicado por "`...`".

Uma chamada de função corresponde então nos seguintes passos, nesta ordem:

1. Avaliação das expressões especificadas na chamada da função.

   Cada expressão usada em uma chamada em computação, correspondente a cada *parâmetro*, é chamada de *parâmetro real*. O resultado obtido pela avaliação de cada parâmetro real é chamado de *argumento*.

   Por exemplo, na chamada $mult$`(3+1,2)`, `3+1` e `2` são parâmetros reais. Os argumentos são `4` e `2`, obtidos pela avaliação das expressões `3+1` e `2`, respectivamente.

2. Criação de registro de ativação da função, contendo espaço para os parâmetros da função (e possivelmente variáveis declaradas localmente).

   No nosso exemplo, é criado registro de ativação contendo espaço para os parâmetros m e n.

3. Atribuição dos argumentos aos parâmetros correspondentes.

   No nosso exemplo $mult$`(3+1,2)`, o argumento `4` (obtido pela avaliação do parâmetro real `3+1`) é atribuído ao parâmetro m.

   Essa atribuição do argumento ao parâmetro é chamada em computação de *passagem de parâmetro*, no caso uma *passagem de parâmetro por valor*. O significado disso é o já explicado: o parâmetro real é avaliado, resultado em um valor, chamado de argumento, e copiado para o parâmetro. O parâmetro é também chamado de *parâmetro formal*.

4. Execução do corpo da função.

5. Liberação do espaço de memória alocado para o registro de ativação da função.

Tabela 4.2: Passos na execução de *multr*`(3,2)`

Comando/ Expressão	**Resultado** (expressão)	**Estado** (após execução/avaliação)		
$\boxed{multr\ \texttt{(3,2)}}$ ...		$m \mapsto 3$ $n \mapsto 2$		
n `== 0`	falso	$m \mapsto 3$ $n \mapsto 2$		
`return` m+ $\boxed{multr\texttt{(}m\texttt{,}n\texttt{-1)}}$	...	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
n `== 0`	falso	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
`return` m+ $\boxed{multr\texttt{(}m\texttt{,}n\texttt{-1)}}$	...	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	$m \mapsto 3$ $n \mapsto 0$
n `== 0`	verdadeiro	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	$m \mapsto 3$ $n \mapsto 0$
`return 0`		$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
`return` m `+ 0`		$m \mapsto 3$ $n \mapsto 2$		
`return` m `+ 3`				
multr `(3,2)`	`6`			

É preciso, é claro, que o resultado da função fique disponível para ser usado, ou seja armazenado em uma variável que exista depois que o espaço de memória alocado para o registro de ativação tiver sido liberado, após a chamada à função.

As definições de *exp* e *expr* seguem o mesmo padrão de definição das funções *mult* e *multr*, respectivamente. A definição de *expr* é baseada na seguinte definição indutiva da exponenciação:

$$\begin{aligned} m^0 &= 1 \\ m^n &= m \times m^{n-1} \qquad \text{se } n > 0 \end{aligned}$$

Eficiência

Uma característica importante da implementação de funções, e de algoritmos em geral, é a sua *eficiência*. A definição alternativa da função de exponenciação apresentada na Figura 4.2 ilustra bem como implementações de uma mesma função, baseadas em algoritmos distintos, podem ter eficiência diferente. Essa nova definição é uma implementação de m^n baseada na seguinte definição indutiva:

$$\begin{aligned} m^0 &= 1 \\ m^n &= (m^{n/2})^2 && \text{se } n \text{ é par} \\ m^n &= m \times m^{n-1} && \text{se } n \text{ é ímpar} \end{aligned}$$

A avaliação de *exp2*`(`m`,`n`)` produz o mesmo resultado que a avaliação de *exp*`(`m`,`n`)` e de *expr*`(`m`,`n`)`, mas de maneira mais eficiente — em termos do número de operações necessárias, e portanto em termos do tempo de execução. Para perceber isso, observe que a avaliação de *exp2*`(`m`,`n`)` é baseada em chamadas recursivas que dividem o valor de n por `2` a cada chamada; a avaliação de *exp*`(`m`,`n`)`, por outro lado, diminui o valor de n de 1 a cada iteração (assim como *expr*`(`m`,`n`)` diminui o valor de n de 1 a cada chamada recursiva).

Por exemplo, as chamadas recursivas que ocorrem durante a avaliação da expressão *exp2*`(2,20)` são, sucessivamente, as seguintes:

```
exp2(2,20)
exp2(2,10)
exp2(2,5)
exp2(2,4)
exp2(2,2)
exp2(2,1)
exp2(2,0)
```

A diferença em eficiência se torna mais significativa quando o expoente n é maior. São realizadas da ordem de $log_2(n)$ chamadas recursivas durante a execução de *exp2*(*m*,*n*) — uma vez que n é em média dividido por 2 em chamadas recursivas —, ao passo que a execução de *exp*(*m*,*n*) requer n iterações.

Um exercício interessante é aplicar a mesma técnica usada na definição de *exp2* (ou seja, usar divisão por 2 caso o segundo operando seja par) para obter uma definição mais eficiente para a multiplicação (procure resolver esse exercício antes de ver sua solução, no Exercício Resolvido 2).

Questões sobre eficiência de algoritmos são importantes em computação, envolvendo aspectos relativos ao tempo de execução e ao consumo de memória (referidos abreviadamente como aspectos de tempo e espaço). A análise da eficiência de algoritmos é abordada superficialmente ao longo deste texto, sendo um estudo mais detalhado desse tema deixado para cursos posteriores.

4.2 Fatorial

Nesta seção, exploramos um pouco mais sobre recursão e iteração, por meio do exemplo clássico do cálculo do fatorial de um número inteiro não-negativo n, usualmente definido como:

$$n! = n \times (n-1) \times (n-2) \times \ldots 3 \times 2 \times 1$$

Duas formas comuns para implementação dessa função são mostradas a seguir:

```
int fat (int n) {
  int f=1, i;
  for (i=1; i<=n; i++) f *= i;
  return f;
}

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

```
int exp2 (int m, int n) {
  if (n == 0) return 1;
  else if (n % 2 == 0) { // n é par
         int x = exp2(m, n/2);
         return x*x; }
       else return m * exp2(m,n-1);
}
```

Figura 4.2: Implementação mais eficiente da operação de exponenciação

A função *fatr* espelha diretamente a seguinte definição indutiva, que especifica precisamente o significado dos três pontos (`...`) na definição de $n!$ dada acima:

$$\begin{array}{ll} n! = 1 & \text{se } n = 0 \\ n! = n \times (n-1)! & \text{em caso contrário} \end{array}$$

A versão iterativa, *fat*, adota a regra de calcular o fatorial de n usando um contador (i), que varia de `1` a n, e uma variável (f), que contém o produto $1 \times 2 \times \ldots i$, obtido multiplicando o valor de i por f a cada iteração. Mudando o valor do contador e o valor do produto a cada iteração (para valores de i que variam de 1 a n), obtemos o resultado desejado na variável f (ou seja, $n!$), ao final da repetição. Verifique que, no caso da execução de uma chamada *fat*`(0)`, nenhuma iteração do comando `for` é executada, e o resultado retornado pela função é, corretamente, igual a `1` (valor inicialmente atribuído à variável f).

O exemplo a seguir mostra como pode ser implementada uma versão recursiva do processo iterativo para cálculo do fatorial. Nesse caso, o contador de repetições (i) e a variável (f), que contém o produto $1 \times 2 \times \ldots i$, são passados como argumentos da função definida recursivamente, sendo atualizados a cada chamada recursiva.

```
int fatIter (int n, int i, int f) {
  if (i > n) return f;
  else return fatIter(n,i+1,f*i);
}

int fatr1 (int n) {
  return fatIter (n,1,1);
}
```

4.3 Obtendo Valores com Processos Iterativos

Outros exemplos de problemas cuja solução requer o uso de recursão ou iteração são abordados a seguir. De modo geral, na solução de tais problemas, o valor calculado em uma iteração ou chamada recursiva de uma função é obtido pela aplicação de funções a valores obtidos na iteração ou chamada recursiva anterior.

O exemplo mais simples é o cálculo do somatório de termos de uma série aritmética. Por exemplo, considere o cálculo do somatório dos termos da série aritmética de passo 1, $\sum_{i=1}^{n} i$, para um dado n, implementada a seguir pela função *pa1*:

```
int pa1 (int n) {
  int s = 0, i;
  for (i=1; i<=n; i++) s += i;
  return s;
}
```

Assim como para o cálculo do fatorial, duas versões recursivas, que usam processos de cálculo diferentes, podem ser implementadas. Essas definições são mostradas a seguir. A primeira (*pa1r*) espelha diretamente a definição indutiva de $\sum_{i=1}^{n} i$, e a segunda (*pa1rIter*) espelha o processo iterativo, de maneira análoga à usada na definição de *fatIter*.

```
int pa1r (int n) }
  if (n==0) return 0;
  else return n + pa1r(n-1);
}

int pa1Iter (int n, int i, int s) {
  if (i > n) return s;
  else return pa1Iter(n, i+1, s+i);
}

int pa1rIter (int n) {
  return pa1Iter(n,1,0);
}
```

É simples modificar essas definições para o cálculo de somatórios dos termos de séries aritméticas com passo diferente de 1. Essas modificações são deixadas como exercício para o leitor.

Nenhuma dessas implementações constitui a maneira mais eficiente para calcular a soma de termos de uma série aritmética, pois tal valor pode ser calculado diretamente, usando a fórmula:[1]

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Essas definições servem, no entanto, para a ilustrar a implementação de somatórios semelhantes, como nos exemplos a seguir.

Considere a implementação de uma função para cálculo de:

$$\sum_{i=0}^{n} x^i$$

Também nesse caso, podemos usar um comando de repetição, como na implementação de *pa1*, sendo adequado agora usar uma variável adicional para guardar a parcela obtida em cada iteração. Essa variável é chamada *parc* na implementação apresentada a seguir, que calcula o somatório das parcelas da série geométrica, $\sum_{i=0}^{n} x^i$, para um dado x e um dado n:

```
int pg (int n, int x) {
  int s = 1, parc = x, i;
  for (i=1; i<=n; i++) {
      s += parc; parc *= x;
  }
  return s;
}
```

O uso de *parc* em *pg* evita que seja necessário calcular x^i (ou seja, evita que uma operação de exponenciação tenha que ser efetuada) a cada iteração. Em vez disso, a parcela da i-ésima iteração (contida em *parc*), igual a x^{i-1}, é multiplicada por x para obtenção da parcela da iteração seguinte.

[1] O leitor com interesse em matemática pode procurar deduzir essas fórmulas. Conta a história que a fórmula $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ foi deduzida e usada por Gauss quando ainda garoto, em uma ocasião em que um professor de matemática pediu-lhe, como punição, que calculasse a soma dos 1000 primeiros números naturais. Deduzindo a fórmula em pouco tempo, Gauss respondeu a questão prontamente.

De fato, é possível deduzir a fórmula rapidamente, mesmo sem uso de papel e lápis, depois de perceber que a soma

$$\begin{array}{cccccccccc} & 1 & + & 2 & + & \ldots & + & (n-1) & + & n \\ + & n & + & (n-1) & + & \ldots & + & 2 & + & 1 \end{array}$$

fornece o mesmo resultado que somar n termos iguais a $n+1$.

A dedução da fórmula $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$ pode ser feita com artifício semelhante, e é deixada como exercício para o leitor.

Para a implementação de um somatório, devemos, portanto, decidir se cada parcela a ser somada vai ser obtida a partir da parcela anterior (e nesse caso devemos declarar uma variável, como a variável *parc*, usada para armazenar o valor calculado para a parcela em cada iteração), ou se cada parcela vai ser obtida por meio do próprio contador de iterações. Como exemplo desse segundo caso, considere a implementação do seguinte somatório:[2]

$$\sum_{i=1}^{n} \frac{1}{i}$$

Nesse caso, a parcela a ser somada a cada iteração, `1/i`, é obtida a partir de i, e não da parcela "anterior", `1/(i-1)`:

```
float somaSerieHarmonica (int n) {
  float s = 0.0f, i;
  for (i=1; i<=n; i++) s += 1/(float)i;
  return s;
}
```

Na maioria das vezes, uma parcela pode ser calculada de maneira mais fácil e eficiente a partir da parcela calculada anteriormente. Em vários casos, esse cálculo não usa a própria parcela anterior, mas valores usados no cálculo dessa parcela. Por exemplo, considere o seguinte somatório, para cálculo aproximado do valor de π:

$$\pi = 4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots)$$

Nesse caso, devemos "guardar" não o valor mas o sinal da parcela anterior (para invertê-lo) e o denominador usado para cálculo dessa parcela (ao qual devemos somar 2 a cada iteração):

```
float piAprox (int n) {
  float s = 0.0f, denom = 1.0f; int sinal = 1, i;
  for (i=1; i<=n; i++) {
      s += sinal/denom;
      sinal = -sinal; denom += 2;
  }
  return 4 * s;
}
```

Em nosso último exemplo, vamos definir uma função para calcular um valor aproximado para e^x, para um dado x, usando a fórmula:

$$e^x = 1 + (x^1/1!) + (x^2/2!) + \ldots$$

Para calcular a parcela a ser somada em cada iteração i do comando de repetição, a implementação usa o denominador $i!$ e o numerador x^i, calculado na parcela anterior. Optamos pelo uso de um comando `while` (em vez do `for`), para ilustração:

[2] A seqüência de números $1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{n}, \ldots$ é denominada *série harmônica*.

```
float eExp (float x, int n) {
  float s = 1.0f; int i=1;
  float numer = x; int denom = 1;
  while (i<=n) {
    s += numer/denom;
    i++;
    numer *= x; denom *= i;
  }
  return s;
}
```

A decisão entre usar um comando `for` ou um comando `while` em `C` é, na maioria das vezes, uma questão de estética ou de gosto. Existe uma diferença quando um comando `continue` é usado internamente a um comando `for` ou um comando `while` (veja Exercício Resolvido 10).

Note que os dois comandos a seguir são equivalentes — se nenhum comando `continue` é usado em c_1:

```
for (c1; b; c2) c

c1; while (b) { c; c2; }
```

Outro comando de repetição disponível em `C` é o comando `do-while`. Esse comando tem a forma:

```
do c; while (b);
```

onde c é um comando e b é uma expressão de tipo *boolean*. Esse comando é bastante semelhante ao comando `while`. No caso do comando `do-while`, no entanto, o comando c é executado uma vez, antes do teste de terminação (b). Por exemplo, considere o seguinte trecho de programa, que imprime os inteiros de `1` a `10`, no dispositivo de saída padrão:

```
int i = 1;
do { printf("%d ",i);
     i++;
   } while (i <= 10);
```

4.3.1 Não-terminação

Pode ocorrer, em princípio, que a avaliação de certas expressões que contêm símbolos definidos recursivamente, assim como a execução de certos comandos de repetição, não termine. Considere, como um exemplo extremamente simples, a seguinte definição:

```
int infinito() { return infinito() + 1; }
```

Essa declaração especifica que a função *infinito* retorna um valor do tipo `int`. Qual seria esse valor? A avaliação de uma chamada à função *infinito* nunca termina, pois envolve, sempre, uma nova chamada a essa função. O valor representado por uma chamada a essa função não é, portanto, nenhum valor inteiro. Em computação, qualquer tipo, predefinido ou declarado em um programa, em geral inclui um valor especial, que constitui um valor indefinido do tipo em questão e que representa expressões desse tipo cuja avaliação nunca termina ou provoca a ocorrência de um erro (como, por exemplo, uma divisão por zero).

Muitas vezes, um programa não termina devido à aplicação de argumentos a funções cujo domínio não engloba todos os valores do tipo da função. Essas funções são comumente chamadas, em computação, de *parciais*. Por exemplo, a definição de *fat*, dada anteriormente, especifica que essa função recebe como argumento um número inteiro e retorna também um número inteiro. Mas note que, para qualquer n `< 0`, a avaliação de *fat*`(`n`)` não termina. A definição de *fat* poderia, é claro, ser modificada de modo a indicar a ocorrência de um erro quando o argumento é negativo.

Considere agora a seguinte definição de uma função — *const1* — que retorna sempre `1`, para qualquer argumento:

```
int const1 (int x) { return 1; }
```

A avaliação da expressão:

const1`(`*infinito*`())`

nunca termina, apesar de *const1* retornar sempre o mesmo valor (`1`), para qualquer argumento (o qual, nesse caso, não é usado no corpo da função). Isso ocorre porque, em `C`, assim como na grande maioria das linguagens de programação, a avaliação dos argumentos de uma função é feita antes do início da execução do corpo da função.

4.4 Correção e Entendimento de Programas

Como mencionado no início do capítulo, os conceitos de recursão e iteração constituem, juntamente com as noções de composição sequencial e seleção, as ferramentas fundamentais para construção de algoritmos e programas, a partir de um conjunto apropriado de operações ou comandos básicos.

A solução de qualquer problema que envolva a realização de uma ou mais operações repetidas vezes pode ser expressa, no paradigma de programação imperativo, por meio de um comando de repetição ou usando funções com definições recursivas.

A escolha de um ou outro método, recursivo ou iterativo, é muitas vezes uma questão de estilo, mas diversos aspectos podem influir na decisão de escolher um ou outro método, principalmente a clareza ou facilidade de entendimento ou de convencimento ou demonstração da correção, e a eficiência do código gerado.

Vamos dar nesta seção uma introdução a aspectos relativos a um entendimento mais preciso e detalhado sobre o comportamento de programas e sobre como convencer (a si próprio e a outros) e demonstrar a correção de programas. Esse assunto é importante particularmente para cursos de ciência da computação e cursos relacionados. Na prática, a demonstração de correção e o convencimento de que um programa se comporta como esperado (em particular no caso de programas imperativos, que envolvem mudanças de estado) são obtidos informalmente. Estes devem ser baseados no entanto em argumentação mais precisa e detalhada, como as apresentadas a seguir. Uma argumentação formal, precisa e detalhada, requer não somente bastante estudo, tempo e esforço, mas também ferramentas adequadas para o processo de formalização, no sentido de prover suporte amigável mas principalmente de modo a se integrar de maneira harmoniosa com a linguagem de programação utilizada. Essa integração harmoniosa de métodos para formalização de propriedades de programas com linguagens de programação ainda é objeto de pesquisa na ciência da computação, e certamente a linguagem `C` apresenta características que tornam essa integração difícil.

Tal dificuldade pode ser sentida notadamente devido a que em `C` é possível:

- usar livremente ponteiros e valores de tipo *união* (`union` em `C`), que tornam possível a alteração do valor armazenado em uma variável sem uso do nome desta variável;

- usar expressões que têm efeito colateral, ou seja, expressões que modificam o valor armazenado em alguma variável, além de retornar um valor. Trechos de programas com expressões que têm efeito colateral podem ser transformados, em geral sem grande dificuldade, em trechos que primeiro usam comandos para provocar o efeito colateral (modificando o valor armazenado em variáveis) e em seguida usam as variáveis com os valores modificados.

Por motivos como esses, devemos notar que o material apresentado nesta seção é bastante introdutório. Começamos com definições recursivas (seção 4.4.1) e depois abordamos programas que envolvem mudanças de estado, com o uso de comandos de repetição e de atribuição (seção 4.4.2).

4.4.1 Definições Recursivas

Definições recursivas de funções são baseadas no princípio de indução, e o entendimento, a correção e demonstrações de propriedades de tais definições recursivas podem ser obtidas por meio de provas por indução.

Como um exemplo simples, vamos mostrar, por indução, que as duas definições recursivas *fatr* e *fatr1* definidas na seção 4.2 (copiadas abaixo) retornam o mesmo resultado ($n!$) quando aplicadas ao mesmo argumento (n).

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

int fatIter (int n, int i, int f) {
  if (i > n) return f;
  else return fatIter(n,i+1,f*i);
}
int fatr1 (int n) {
  return fatIter (n,1,1);
}
```

Mais precisamente, vamos mostrar o seguinte.

Lema 1. *Para todo n inteiro maior ou igual a zero, temos que $fatr(n) = fatr1(n) = n!$.*

Prova: O lema envolve consiste de fato em duas propriedades, que vamos chamar de P_1 e P_2:

$$P_1(n) = (n \in \mathbb{I} \wedge n \geq 0) \to fatr(n) = n!$$
$$P_2(n) = (n \in \mathbb{I} \wedge n \geq 0) \to fatr1(n) = n!$$

onde supomos que $\mathbb{I}$ representa o conjunto dos valores de tipo `int` e $n!$ é definido, para todo $n \in \mathbb{I}, n \geq 0$, como:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n * (n-1)! & \text{caso contrário} \end{cases}$$

A prova de P_1 é direta, por indução sobre n, uma vez que a definição de *fatr* consiste simplesmente em no uso de uma notação diferente (a linguagem `C`) usada para se escrever a função fatorial (!). A prova de P_1 é incluída a seguir como ilustração.

Caso base ($n = 0$): Nesse caso, temos:

$$\begin{array}{lll} fatr(0) & = \texttt{1} & \text{(def. de } fatr\text{)} \\ & = 0! & \text{(def. de } 0!\text{)} \end{array}$$

Caso indutivo ($P_1(n)$ implica $P_1(n+1)$, para $n \geq 0$, $n+1 \in \mathbb{I}$):

$$\begin{array}{llll} fatr(n+1) & = (n+1) \texttt{*} fatr(n) & \text{(def. de } fatr,\ n+1 > 0\text{)} & (1) \\ & = (n+1) \texttt{*} (\text{n}!) & \text{(hip. de indução)} & (2) \\ & = (n+1) \times (\text{n}!) & \text{(hip.: } \texttt{*} \text{ implementa } \times\text{)} & (3) \\ & = (\text{n+1})! & \text{(def. de !)} & (4) \end{array}$$

Para obter (3), precisamos supor que não ocorre "overflow", isto é, que a operação de multiplicação usada em `C` (`*`) se comporta como a operação matemática de multiplicação ($\times$). Em implementações que usam 32 bits para armazenamento de números inteiros, essa suposição não é válida (ocorre *overflow*) para $n \geq 17$. Portanto, de fato, o lema só é válido, em implementações que usam 32 bits para armazenamento de valores de tipo `int`, para n inteiro maior ou igual a zero e menor ou igual a 16.

Para provar P_2, observemos que, em qualquer chamada a *fatIter* com parâmetros n, i e f:

1. A condição

$$(n \geq 0) \wedge (i \leq n+1) \wedge \big(f = (i-1)!\big)$$

deve ser verdadeira, no início da execução de uma chamada à função.

Uma condição que deve ser verdadeira no início da execução de um comando é chamada de *pré-condição*.

Analogamente, uma condição que é verdadeira no final da execução de um comando, se a pré-condição for verdadeira, é chamada de *pós-condição*.

O programa deve ser feito de modo a garantir que a pré-condição seja verdadeira na primeira chamada à função. No caso de *fatIter*, por exemplo, deve ser garantido sempre que $n > 0$, caso contrário a execução de *fatIter* não terminará.

Sendo a pré-condição verdadeira, podemos verificar facilmente que ela continuará a ser verdadeira, no início da execução da chamada recursiva. Isto porque, na chamada recursiva, sendo i' e f' os parâmetros de nomes i e f em uma chamada que faz esta chamada recursiva, temos que $i' = i + 1$ e $f' = f * i$; como $f = (i-1)!$, temos que $f' = i! = (i'-1)!$.

A pré-condição é, assim, uma condição *invariante* em chamadas recursivas a *fatIter*, desde que seja verdadeira na primeira chamada à função.

2. A expressão

$$n - i + 1$$

é sempre não-negativa, se o for na primeira chamada à função, e decresce em cada chamada recursiva à função.

Tal expressão, que é decrescente mas permanece não-negativa, se for não-negativa na primeira iteração ou chamada recursiva de uma função, é chamada de uma expressão *variante*. Entenda-se *expressão não-negativa decrescente* quando se falar de uma "expressão variante".

A existência de uma expressão variante garante que uma função recursiva terminará, caso tal expressão seja não-negativa na primeira chamada à função.

As duas condições acima garantem que, quando $i > n$, temos que $i = n+1$ e, portanto, $f = n!$.

4.4.2 Comandos de Repetição

Para expressar o comportamento de comandos, podemos usar o que se chama de *semântica axiomática*, que usa *fórmulas de correção de comandos*, escritas comumente na forma:

$$\{p\}\ c\ \{q\}$$

onde c é um comando e p e q são *asserções* (ou condições, ou fórmulas proposicionais). A asserção p é chamada *pré-condição* e q *pós-condição* associada ao comando c.

A pré-condição especifica o que deve ser válido no estado anterior, e a pós-condição o que deve ser válido no estado posterior, à execução do comando.

Mais precisamente, dizemos que:

- $\{p\}\ c\ \{q\}$ é *parcialmente* correta se toda execução de c que for iniciada em um estado em que p é verdadeiro e terminar, termina em um estado em que q é verdadeiro.

- $\{p\}\ c\ \{q\}$ é *totalmente* correta se toda execução de c que for iniciada em um estado em que p é verdadeiro terminar em um estado em que q é verdadeiro.

Ou seja, correção total inclui a condição de que o comando termina, ao contrário da correção parcial.

Vamos mostrar a seguir que a definição iterativa *fat* dada na seção 4.2 (copiada abaixo) computa o fatorial do argumento fornecido.

```
int fat (int n) {
  int f=1, i;
  for (i=1; i<=n; i++) f *= i;
  return f;
}
```

Mais precisamente, vamos demonstrar o seguinte (onde $\mathbb{I}$, introduzido acima, é o conjunto dos valores de tipo `int`).

Lema 2. *Em um programa sintaticamente correto, o resultado da execução de fat*(n) *é igual a* $n!$*, para todo* $n \in \mathbb{I} \geq 0$.

Para provar este lema, vamos transformar o comando `for` acima em um comando `while` equivalente e usar as semânticas axiomáticas da composição sequencial de comandos e dos comandos de atribuição e `while`, explicadas a seguir.

Semântica axiomática da composição sequencial de comandos:

$$\frac{\begin{array}{c}\{p\}\, c_1;\ \{q\} \\ \{q\}\, c_2;\ \{p'\}\end{array}}{\{p\}\, c_1\texttt{;}\ \ c_2\texttt{;}\ \{p'\}}$$

A semântica da composição sequencial de c_1 com c_2 é auto-explicativa: simplesmente especifica que a pós-condição p' deve ser obtida a partir da pós-condição p por uma asserção que é tanto pós-condição de c_1 e pré-condição de c_2.

Semântica axiomática do comando de atribuição: Para expressar a semântica do comando de atribuição de forma axiomática, precisamos supor que um comando de atribuição é da forma x `=` e onde e é uma expressão sem efeito colateral, isto é, que não causa mudança no valor armazenado em nenhuma variável. Desta forma, temos:

$$\frac{}{\{p\texttt{[}x := e\texttt{]}\}\ x\texttt{=}e\texttt{;}\ \{p\}}$$

A notação $p\texttt{[}x := e\texttt{]}$ especifica uma asserção que difere de p pela substituição textual de todas as ocorrências de x por e. Se $p\texttt{[}x := e\texttt{]}$ é válido antes do comando de atribuição

$$x\texttt{=}e\texttt{;}$$

então, depois da execução desse comando, p é válido.

Por exemplo, $\{x = 10\}\ x = x + 1\ \{x = 11\}$ é válido, pois $(x = 11)\texttt{[}x := x + 1\texttt{]}$ é equivalente a $x + 1 = 11$ (pela substituição textual de x por $x + 1$), ou seja, $x = 10$.

A semântica do comando `while` pode ser descrita como a seguir:

$$\frac{\begin{array}{l}\{p \wedge b\}\ c;\ \{p\} \\ \{p \wedge b \wedge \alpha = e\}\ c;\ \{e < \alpha\} \\ (p \wedge b) \to (e \geq 0)\end{array}}{\{p\}\ \texttt{while}\ b\ \texttt{do}\ c;\ \{p \wedge (\neg b)\}}$$

onde e é uma expressão inteira e α é uma variável nova, que não ocorre em p, b, e e c.

Podemos descrever a regra acima como a seguir:

- a fórmula de correção $\{p \wedge b\}$ c; $\{p\}$ estabelece que p é uma condição *invariante* da iteração (ou, em outras palavras, um invariante durante a execução do comando `while`, ou, de forma abreviada, um invariante do comando `while`).

- as premissas nas quais e ocorre garantem a terminação da iteração. O propósito de α é guardar o valor de e antes da execução de c (uma vez que α não ocorre em c, o valor de α é o mesmo antes e depois da execução de c). A premissa $\{p \wedge b \wedge \alpha = e\} c \{e < \alpha\}$ estabelece que o valor de e decresce a cada iteração. Juntamente com a premissa $(p \wedge b) \rightarrow (e \geq 0)$, nenhuma computacão infinita de c pode ser realizada, uma vez que e decresce e é maior ou igual a zero sempre que $p \wedge b$ for verdadeiro.

Podemos agora mostrar a prova do nosso lema.

Prova: Todo comando `for` em `C` tem o seguinte formato (veja seção 4.1):

```
for (e0; e1; e2) c
```

para algum e_0, e_1, e_2, c. O fato de e_0 e e_2 serem expressões e não comandos em `C` é uma idiossincrasia da definição da linguagem `C`. Na ausência de um comando `continue` como parte do comando c, todo comando `for` é equivalente a um comando `while`, da forma:

```
e0; while (e1) { c; e2; }
```

Obs.: no caso do comando `continue` ocorrer em c, a execução deve continuar em e_2 e não em e_1, segundo a semântica do comando `for`.

Portanto,

```
int fat (int n) {
  int f=1, i;
  for (i=1; i<=n; i++) f *= i;
  return f;
}
```

é equivalente a:

```
int fat (int n) {
  int f=1, i;
  i=1;
  while (i<=n)
    f *= i;
    i++;
  }
  return f;
}
```

Para provar o lema, anotamos as proposições invariantes em cada parte da iteração, como a seguir:

```
int fat (int n) {
  // {n ≥ 0}        (1)
  int f=1, i;
  i=1;
  // {(f = 1) ∧ (i = 1) ∧ (n ≥ 0)}        (2)
  // {(f = (i − 1)!) ∧ (i ≥ 1) ∧ (n ≥ 0) ∧ (i ≤ n + 1)}        (3)
  while (i<=n)
    // {(f = (i − 1)!) ∧ (i ≥ 1) ∧ (n ≥ 0) ∧ (i ≤ n) ∧ (α = n − i)}        (4)
    f *= i;
    // {f = i! ∧ (i ≥ 1) ∧ (n ≥ 0) ∧ (i ≤ n)}        (5)
    i++;
  // {f = (i − 1)! ∧ (i − 1 ≥ 1) ≥ (n ≥ 0) ∧ (i − 1 ≤ n)} ∧ (n − i < α)        (6)
  }
  // {f = n! ∧ (i = n + 1) ∧ (n ≥ 0)}        (7)
  return f;
}
```

A prova segue imediatamente, usando a semântica dos comandos `while`, do comando de atribuição e da composição sequencial de comandos:

- (1) segue de (2) usando a semântica do comando de atribuição e da composição sequencial,
- (2) implica (3),
- (3) e $i \leq n$ implica (4),
- (4) segue de (5) usando a semântica do comando de atribuição,
- (5) segue de (4) usando a semântica do comando de atribuição,
- (5) segue de (6) usando a semântica do comando de atribuição,
- (6) e a negação de $i \leq n$ implica (7).

4.4.3 Semântica Axiomática

Reunimos abaixo a semântica axiomática de comandos e da composição sequencial de comandos, usando fórmulas de correção $\{p\}\, c\, \{q\}$.

...

4.4.4 Exemplos

...

4.5 Exercícios Resolvidos

1. Defina recursivamente o significado do comando `while`.

Solução: Esse comando pode ser definido recursivamente pela seguinte equação:

```
while ( b ) c = if ( b ) { c; while ( b ) c }
```

Obs.: O símbolo = é usado acima para definir uma equação matemática, e não como um comando de atribuição.

2. Defina uma função *mult2* que use a mesma técnica empregada na definição de *exp2*. Ou seja, use divisão por 2 caso o segundo operando seja par, para obter uma implementação da operação de multiplicação mais eficiente do que a apresentada em *multr*.[3]

Solução:

```
static int mult2 (int m, int n) {
  if (n == 0) return 0;
  else if (n % 2 == 0) { // n é par
          int x = mult2 (m,n/2);
             return x + x; }
          else return m + mult2 (m,n-1);
  }
```

3. Dê uma definição recursiva para uma função *pgr* que espelhe o processo iterativo de *pg*. Em outras palavras, defina recursivamente uma função que, dados um número inteiro x e um número inteiro não-negativo n, retorne o valor do somatório:

$$\sum_{i=0}^{n} x^i$$

obtendo cada termo do somatório a partir do termo anterior (pela multiplicação desse termo anterior por x) e passando como argumento, em cada chamada recursiva, o termo e o somatório obtidos anteriormente.

Escreva um programa que leia repetidamente pares de valores inteiros x e n, até que o valor de n seja zero ou negativo, e imprima o valor do somatório $\sum_{i=0}^{n} x^i$, usando a função *pgr*.

Solução:

```
int pgIter (int x, int n, int i, int s, int t) {
  if (i > n) return t;
  else { int sx = s*x;
         return pgIter(x,n,i+1,sx,t+sx);
       }
  }

  int pgr (int x, int n) {
       return pgIter (x,n,1,1,1);
  }

  int main () {
       int x,n;
       while (1) {
         scanf("%d %d",&x, &n);
         if (n <= 0) break;
         printf("%d",pgr(x,n));
       }
  }
```

4. Dê uma definição para função *exp2*, definida na Figura 4.2, usando um comando de repetição, em vez de recursão.

Solução:

[3] Exemplos do uso desse algoritmo são encontrados em um dos documentos matemáticos mais antigos que se conhece, escrito por um escrivão egípcio (A'h-mose), por volta de 1700 a.C.

```
int exp2 (int m, int n) {
  int r = 1;
  while (n != 0)
    if (n % 2 == 0) {
       n = n / 2;
       m = m * m;
    } else { r = r * m;
             n = n - 1; }
    return r;
  }
```

A definição recursiva apresentada na Figura 4.2 é mais simples, pois espelha diretamente a definição indutiva de *exp2*, dada anteriormente. A definição acima usa um esquema não-trivial (não diretamente ligado à definição da função) para atualização do valor de variáveis no corpo do comando de repetição.

5. Defina uma função para calcular o máximo divisor comum de dois números inteiros positivos.

Solução: O *algoritmo de Euclides* é um algoritmo clássico e engenhoso para cálculo do máximo divisor comum de dois números inteiros positivos:

$$mdc(a,b) = \begin{cases} a & \text{se } b = 0 \\ mdc(b, a\%b) & \text{caso contrário} \end{cases}$$

O operador % é usado acima como em `C`, ou seja, `a%b` representa o resto da divisão inteira de `a` por `b` (`a` e `b` são números inteiros). O algoritmo original de Euclides (escrito no famoso livro *Elementos*, por volta do ano 3 a.C.) usava $mdc(b, a - b)$ em vez de $mdc(b, a\%b)$, na definição acima, mas o uso da divisão torna o algoritmo mais eficiente.

Note que esse algoritmo funciona se $a \geq b$ ou em caso contrário. Se $a < b$, a chamada recursiva simplesmente troca a por b (por exemplo, $mdc(20, 30)$ é o mesmo que $mdc(30, 20)$).

Em Java, podemos escrever então:

```
int mdc (int a, int b) {
  if (b == 0) return a;
  else return mdc(b, a%b);
}
```

Podemos também escrever a função *mdc* usando um comando de repetição, como a seguir:

```
int mdc (int a, int b) {
  int t;
  if (a<b) return mdc(b,a);
  while (b>0) { t = a; a = b; b = t % b; };
  return a;
}
```

6. Defina uma função *calc* que receba como argumentos um caractere, que indica uma operação aritmética (`'+'`, `'-'`, `'*'` e `'/'`), e dois valores de ponto flutuante, e retorne o resultado da aplicação da operação sobre os dois argumentos. Por exemplo: *calc* `('+', 1.0, 1.0)` deve retornar `2.0` e *calc* `('*', 2.0, 3.0)` deve retornar `6.0`.

Solução: Vamos ilustrar aqui o uso do comando `switch`, existente em Java, que permite escolher um dentre vários comandos para ser executado. O comando `switch` tem a forma:

```
switch (e) {
  case e1:  c1;
  case e2:  c2;
  ...
  case en:  cn;
}
```

O comando `switch` pode ser visto como uma seqüência de comandos de seleção `if`, cada um realizando um teste correspondente a um caso do comando `switch`.

O significado de um comando `switch` é o seguinte: a expressão e é avaliada e é executado o primeiro comando c_i, na ordem $c_1, \ldots, c_n$, para o qual o valor fornecido por e é igual ao valor de e_i (caso tal comando exista), sendo também executados todos os comandos seguintes (c_{i+1}, ..., c_n), se existirem, nessa ordem.

A execução de qualquer desses comandos pode ser finalizada, e geralmente deve ser, por meio de um comando `break`. No entanto, na solução desse exercício o uso de um comando `break` não é necessário, pois toda alternativa contém um comando `return`, que termima a execução da função.

Supõe-se também, na função *op* definida abaixo, que o caractere passado como argumento para *op* é sempre igual a `'+'`, `'*'`, `'-'` ou `'/'`.

```
double op (char c, double a, double b) {
  switch (c) {
    case '+' :  { return a + b; }
    case '*' :  { return a * b; }
    case '-' :  { return a - b; }
    case '/' :  { return a / b; }-
    }
  }
```

Se o valor de e não for igual a nenhum dos valores e_i, para $i \leq n$, um caso *default* pode ser usado, tal como nesse exemplo. O caso `default` pode ser usado no lugar de `case` e_i, para qualquer e_i, para algum $i = 1, \ldots, n$, sendo em geral usado depois do último caso. Se `default` não for especificado (é comum dizer "se não existir nenhum caso `default`"), o comando `switch` pode terminar sem que nenhum dos comandos c_i, para $i = 1, \ldots, n$, seja executado (isso ocorre se o resultado da avaliação de e não for igual ao valor de nenhuma das expressões e_i, para $i = 1, \ldots, n$).

A necessidade do uso de um comando `break`, sempre que se deseja que seja executado apenas um dos comandos do comando `switch`, correspondente a um determinado caso (expressão), é hoje em geral reconhecida como um ponto fraco do projeto da linguagem Java (tendo sido herdado da linguagem C).

A expressão e, no comando `switch`, deve ter tipo `int`, `short`, `byte` ou `char`, devendo o seu tipo ser compatível com o tipo das expressões e_1, ..., e_n. As expressões $e_1, \ldots, e_n$ têm que ser valores constantes (e distintos).

Um comando `switch` pode ser também precedido de um *rótulo* — um nome seguido do caractere "`:`". Nesse caso, um comando `break` pode ser seguido desse nome: isso indica que, ao ser executado, o comando `break` causa a terminação da execução do comando `switch` precedido pelo rótulo especificado.

7. Escreva uma definição para a função *raizq*, que calcula a raiz quadrada de um dado valor x, com erro de aproximação menor que `0.0001`.

```
const double e = 0.0001;

double raizq (double x) {
  double y = x;
  while (!fim(y,x)) y = melhore(y,x);
  return y;
}

int fim (double y, double x) {
  return math.abs(y * y - x) < e;
}

double melhore (double y, double x) {
  return (y + x/y) / 2;
}
```

Figura 4.3: Cálculo da raiz quadrada, usando o método de Newton

Solução: A raiz quadrada de x é um número y tal que $y^2 = x$. Para certos números, como, por exemplo, 2, a sua raiz quadrada é um número que teria que ser representado com infinitos algarismos na sua parte decimal, não sendo possível obter essa representação em um tempo finito. Desse modo, o cálculo desses valores é feito a seguir, com um erro de aproximação inferior a um valor preestabelecido — nesse caso, `0.0001`.

Vamos chamar de e o valor `0.0001`. Denotando por $|x|$ o valor absoluto de x, o valor y a ser retornado por *raizq*(x) deve ser tal que:

$$y \geq 0 \text{ e } |y^2 - x| < e$$

Para definir a função *raizq*, vamos usar o método de aproximações sucessivas de Newton. Para a raiz quadrada, o método de Newton especifica que, se y_i é uma aproximação para $\sqrt{x}$, então uma aproximação melhor é dada por:

$$y_{i+1} = (y_i + x/y_i)/2$$

Por exemplo, sejam $x = 2$ e $y_0 = 2$. Então:

$$\begin{array}{ll} y_1 = (2 + 2/2)/2 & = 1.5 \\ y_2 = (1.5 + 2/1.5)/2 & = 1.4167 \\ y_3 = (1.4167 + 2/1.4167)/2 & = 1.4142157\ldots \end{array}$$

Repetindo esse processo, podemos obter aproximações para a raiz quadrada de 2 com qualquer precisão desejada (sujeitas, é claro, às limitações da representação de números do computador). A implementação da função *raizq* pode ser feita usando um comando de repetição, como mostrado na Figura 4.3.

8. Em 1883, o matemático francês Édouard Lucas inventou a seguinte pequena *estória*:

> Um templo, na cidade de Hanói, contém três torres de diamante, em uma das quais Deus colocou, quando criou o mundo, 64 discos de ouro, empilhados uns sobre os outros, de maneira que os discos diminuem de tamanho da base para o topo, como mostrado na Figura 4.4 a seguir. Os monges do templo trabalham sem cessar para transferir, um a um, os 64 discos da torre em que foram inicialmente colocados para uma das duas outras torres, mas de forma que um disco nunca pode ser colocado em cima de outro menor. Quando os monges terminarem de transferir todos os discos para uma outra torre, tudo virará pó, e o mundo acabará.

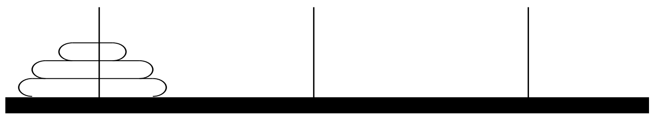

Figura 4.4: Torres de Hanói (com 3 discos)

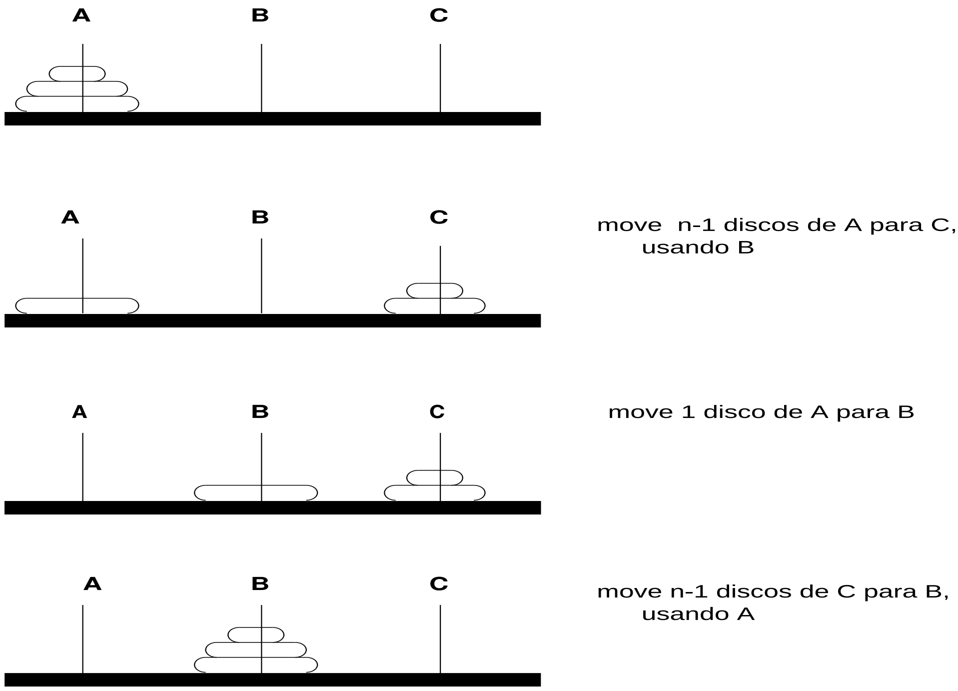

Figura 4.5: Solução do problema das torres de Hanói (com 3 discos)

A questão que se coloca é: supondo que os monges trabalhem tão eficientemente quanto possível, e consigam transferir 1 disco de uma torre para outra em 1 segundo, quanto tempo decorreria (em segundos) desde a criação até o fim do mundo?

Solução: Uma solução recursiva para o problema é baseada na idéia ilustrada na Figura 4.5. Nessa solução, supõe-se, como hipótese indutiva, que se sabe como transferir $n-1$ discos de uma torre para outra (sem colocar um disco em cima de outro menor); o caso base consiste em transferir um disco de uma torre para outra vazia. O número total de movimentações de n discos é definido indutivamente por:

$$\begin{aligned} f(1) &= 1 \\ f(n) &= 2 \times f(n-1) + 1 \end{aligned}$$

Cada valor da seqüência de valores definida por $f(n)$ representa, de fato, o menor número de movimentações requeridas para mover n discos de uma torre para outra, satisfazendo o requerimento de que um disco nunca pode ser colocado sobre outro de diâmetro menor. Para perceber isso, basta notar que, para mover um único disco d, digamos, da torre A para a torre B, é preciso antes mover todos os discos menores que d para a torre C.

Portanto, a resposta à questão proposta anteriormente é dada por $f(64)$. Um resultado aproximado é $1,844674 \times 10^{19}$ segundos, aproximadamente 584,5 bilhões de anos.

A definição de f estabelece uma *relação de recorrência* para uma seqüência de valores $f(i)$, para $i = 1, \ldots, n$ (uma relação de recorrência para uma seqüência é tal que cada termo é definido, por essa relação, em termos de seus predecessores). A primeira equação na definição de f é chamada *condição inicial* da relação de recorrência.

Podemos procurar obter uma fórmula que define diretamente, de forma não-recursiva, o valor de $f(n)$, buscando estabelecer um padrão que ocorre no cálculo de $f(n)$, para cada n:

$$\begin{array}{l}
f(1) = 1\\
f(2) = 2 \times f(1) + 1 = 2 \times 1 + 1 = 2 + 1\\
f(3) = 2 \times f(2) + 1 = 2(2 + 1) + 1 = 2^2 + 2 + 1\\
f(4) = 2 \times f(3) + 1 = 2(2^2 + 2 + 1) + 1 = 2^3 + 2^2 + 2 + 1\\
\ldots\\
f(n) = 2 \times f(n-1) + 1 = 2^{n-1} + 2^{n-2} + \ldots + 2 + 1
\end{array}$$

Podemos observar que $f(n) = 2^n - 1$ (pois $\sum_{i=0}^{n-1} 2^i = 2^n - 1$). O leitor com interesse em matemática pode provar (usando indução sobre n) que, para todo n, a definição recursiva de f de fato satisfaz a equação $f(n) = 2^n - 1$.

9. O problema descrito a seguir foi introduzido em 1202 por Fibonacci — também conhecido como Leonardo de Pisa, e considerado como o maior matemático europeu da Idade Média:

> Supondo que um par de coelhos — um macho e uma fêmea — tenha nascido no início de um determinado ano, que coelhos não se reproduzam no primeiro mês de vida, que depois do primeiro mês um par de coelhos dê origem, a cada mês, a um novo par — macho e fêmea — e que nenhuma morte ocorra durante um ano, quantos coelhos vão existir no final do ano?

Escreva um programa para solucionar esse problema, imprimindo o número de coelhos existentes no final do ano.

Solução: Note que: 1) o número de (pares de) coelhos vivos no final de um mês k é igual ao número de (pares de) coelhos vivos no final do mês $k-1$ mais o número de (pares de) coelhos que *nasceram* no mês k; e 2) o número de (pares de) coelhos que *nasceram* no mês k é igual ao número de (pares de) coelhos vivos no mês $k-2$ (pois esse é exatamente o número de coelhos que geraram filhotes no mês k).

Portanto, a quantidade de coelhos vivos ao fim de cada mês n é dada pelo número $fib(n)$ da seqüência de números definida a seguir, conhecida como seqüência de Fibonacci:

$$\begin{array}{lll}
fib(0) & = 0 & \\
fib(1) & = 1 & \\
fib(n) & = fib(n-1) + fib(n-2) & \text{se } n > 1
\end{array}$$

O problema pode então ser resolvido pelo programa a seguir:

```
int main () {
  printf("%d",fib(12));
}

int fib (int n) {
  if (n==0) return 0;
  else if (n==1) return 1;
  else return fib(n-1) + fib(n-2);
}
```

O número de pares de coelhos no final do décimo segundo mês, `fib(12)`, é igual a 144 (ou seja, o número de coelhos é igual a 288).

A função *fib* definido no programa acima é bastante ineficiente, pois repete muitos cálculos, devido às chamadas recursivas a `fib(n-1)` e `fib(n-2)`. Por exemplo, o cálculo de `fib(4)` envolve calcular `fib(2)` duas vezes. O método a seguir é claramente mais eficiente, podendo ser usado para o mesmo efeito, no programa acima, por meio da chamada `fib(12,0,1)`:

```
int fib (int n, int r1, int r) {
  if (n==1) return r;
  else return fib(n-1, r, r1+r);
}
```

Por questão de eficiência, o teste de igualdade com zero foi retirado, supondo-se então que o argumento de *fib* deve ser um inteiro positivo.

As chamadas a *fib* usam *r1* e *r* como acumuladores, comportando-se como mostrado a seguir:

1ª chamada	`fib(n, 0, 1)`	`fib(0) = 0`	`fib(1) = 1`
2ª chamada	`fib(n-1, 1, 1)`	`fib(1) = 1`	`fib(2) = 1`
...			
última chamada	`fib(1, r1, r)`	`fib(n-1) = r1`	`fib(n) = r`

A definição recursiva anterior de *fib* simula o mesmo processo iterativo do comando de repetição mostrado a seguir:

```
int fib (int n) {
  int r1 = 0, r = 1, t, i;
  for (i=2; i<=n; i++) {
      t = r1; r1 = r; r = r + t; }
  return r;
}
```

10. Os comandos `break` e `continue` podem ser usados no corpo de um comando de repetição. A execução de um comando `break` causa o término da repetição e a execução de um comando `continue` causa o início imediato de uma próxima iteração. O comando `continue` provê uma forma de saltar determinados casos em um comando de repetição, fazendo com que o controle passe para a iteração seguinte.

O exemplo da Figura 4.6 imprime a representação unária (usando o símbolo “`|`”) de todos os números inteiros de `1` até n que não são múltiplos de um determinado valor inteiro $m > 1$. O resultado da execução de `Exemplo_continue 20 5` é mostrado na Figura 4.7.

Os comandos `break` e `continue` podem especificar um rótulo, de mesmo nome do rótulo colocado em um comando de repetição. No caso de um comando `break`, o rótulo pode ser colocado também em um bloco ou em um comando `switch` (veja o Exercício Resolvido 6).

Como ilustra esse exemplo, um comando `continue` sem um rótulo indica o início da execução da próxima iteração do comando de repetição mais interno em relação a esse comando `continue`. Analogamente, um comando `break` sem um rótulo indica o término da execução do comando de repetição ou comando `switch` mais interno em relação a esse comando `break`.

O comando `break` provê uma forma de sair de um comando de repetição, que pode ser, em algumas situações, mais fácil (ou mais conveniente do que outras alternativas). Em alguns casos, pode ser mais fácil testar uma condição internamente ao comando de repetição e usar o comando `break`, em vez de incluir essa condição no teste de terminação do comando de repetição.

O uso do comando `break` é ilustrado pelo exemplo a seguir. Considere o problema de ler, repetidamente, um valor inteiro positivo e imprimir, para cada inteiro lido, seu fatorial. Se for lido um valor negativo, a execução do programa deve terminar.

11. Esse exercício ilustra o uso de entrada de dados até que uma condição ocorra ou até que se chegue ao final dos dados de entrada. O caractere `Control-d` é usado para especificar fim dos dados de entrada, em uma entrada de dados interativa no sistema operacional Linux e,

```
int main () {
  int n, m, col = 1, i;
  printf("Impressao de valores de 1 a n em notacao unaria, sem multiplos de m\n");
  printf("Digite n e m:  ");
  scanf("%d %d",&n, &m);

  for (i=1; i<=n; i++) {
      printf("\n");
      if (i % m == 0) continue;
      for (col=1; col<=i; col++) printf("|");
  }
  return EXIT_SUCCESS;
}
```

Figura 4.6: Exemplo de uso do comando `continue`

```
Impressao de valores de 1 a n em notacao unaria, sem multiplos de m
Digite n e m: 20 5

|
||
|||
||||

||||||
|||||||
||||||||
|||||||||

|||||||||||
||||||||||||
|||||||||||||
||||||||||||||

||||||||||||||||
|||||||||||||||||
||||||||||||||||||
||||||||||||||||||
```

Figura 4.7: Resultado do programa de exemplo de uso do comando `continue`

```
int fat(int x) {
  int fatx = 1, i; for (i=1; i<=x; i++) fatx *= i;
  return fatx;
}

int main () {
  int v;
  while (1) {
    printf("Digite um valor inteiro:  ");
    scanf("if (v < 0) break;
    printf("Fatorial de %d = }
    return SYSTEM_SUCCESS;
  }
```

Figura 4.8: Exemplo de uso do comando `break`

no sistema operacional Windows, `Control-z` no início da linha seguido da tecla `Enter` (ou `Return`).

Uma chamada à função *scanf* retorna o número de variáveis lidas, e pode ser usado para detectar fim dos dados de entrada a serem lidos (isto é, se não existe mais nenhum valor na entrada de dados a ser lido). O exemplo a seguir lê vários inteiros do dispositivo de entrada padrão e imprima a soma de todos os inteiros lidos. O programa termina quando não há mais valores a serem lidos.

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  int n, soma = 0, testeFim;
  while (1) {
    testeFim = scanf("%d",&n);
    if (testeFim != 1) break;
    soma = soma + n;
  }
  printf("Soma = %d\n", soma);
  return EXIT_SUCCESS;
}
```

O programa a seguir funciona de modo semelhante, mas lê vários inteiros *positivos* do dispositivo de entrada padrão e termina a execução quando quando não há mais valores a serem lidos ou quando um valor negativo ou zero for lido.

```
#include <stdio.h>
#include <stdlib.h>
int main () {
  int n, soma = 0, testeFim;
  while (1)) {
    testeFim = scanf("%d",&n);
    if (testeFim != 1 || n<=0) break;
    else soma = soma + n;
  }
  printf("Soma = %d\n", soma);
  return EXIT_SUCCESS;
}
```

12. Escreva um programa que leia, do dispositivo de entrada padrão, vários valores inteiros, positivos ou não, e imprima, no dispositivo de saída padrão, os dois maiores valores lidos.

A entrada termina com indicação de fim dos dados de entrada (em entrada interativa, `Control-z` seguido de `Enter` no Windows, ou `Control-d` no Linux).

Solução: São usadas duas variáveis inteiras *max1* e *max2* para armazenar os dois maiores valores lidos, e elas são atualizadas adequadamente, se necessário, após a leitura de cada inteiro. O valor inicial atribuído a essas variáveis é o valor *INT_MIN*, menor inteiro armazenável em uma variável de tipo `int`. Qualquer valor inteiro digitado será maior ou igual a esse valor e será, caso for maior, atribuído à variável. *INT_MIN* é definido em *limits.h*.

O valor retornado por *scanf* é usado para verificar fim dos dados de entrada (*scanf* retorna `-1` como indicação de fim dos dados de entrada).

O programa é mostrado a seguir.

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

int main() {
  int max1 = INT_MIN, max2 = INT_MIN, valor, fim;
  while (1) {
    fim = scanf("%d",&valor);
    if (fim == -1) break;
    if (valor > max1) { max2 = max1; max1 = valor; }
    else if (valor > max2) max2 = valor;
  }
  printf("Dois maiores = %d, %d\n", max1,max2);
}
```

13. Escreva um programa que leia, do dispositivo de entrada padrão, um texto qualquer, caractere a caractere e imprima, no dispositivo de saída padrão, i) o número de caracteres, ii) o número de palavras, e iii) o número de linhas do texto.

Considere que uma palavra é uma sequência de um ou mais caracteres que começa com qualquer caractere que não é um delimitador de palavras. Um delimitador de palavras é um caractere espaço (branco, i.e. `' '`), fim-de-linha (`'\n`) ou tab (caractere de tabulação, i.e. `\t`).

A entrada termina com indicação de fim dos dados de entrada (em entrada interativa, `Control-z` seguido de `Enter` no Windows, ou `Control-d` no Linux).

Solução: A solução mostrada a seguir usa *scanf* com formato `%c` para ler um caractere, e o valor retornado por *scanf* para verificar fim dos dados de entrada (*scanf* retorna `-1` para indicar fim dos dados).

A função *isletter* definida retorna verdadeiro (em C, valor inteiro diferente de zero) se e somente se o caractere passado como argumento é uma letra. Para isso, é testado se tal caractere está entre 'a' e 'z' ou entre 'A' e 'Z'.

Para contar palavras, é usado uma variável (*fora*) que indica se o caractere corrente, que está sendo lido, está fora ou dentro de uma palavra. O número de palavras (armazanado na variável *palavras*) é incrementado quando se está fora de uma palavra e um caractere não delimitador de palavras é lido (como especificado no enunciado, um delimitador de palavras é considerado como sendo um dos caracteres espaço, fim-de-linha ou tab).

O programa é mostrado a seguir.

```
#include <stdio.h>

int delim(char c) {
  return (c == ' ') || (c == '\t') || (c == '\n');
}

int main() {
  char c; int caracs = 0, palavras = 0, linhas = 0, isDelim, fim, fora=1;
  while (1) {
    fim = scanf("%c",&c);
    if (fim == -1) break;
    caracs++;
    isDelim = delim(c);
    if (isDelim) {
      if (c=='\n') linhas++;
      fora=1;
    }
    else if (fora) { palavras++; fora = 0; }
  }
  printf("Numero de caracteres,palavras,linhas = %d,%d,%d\n",
         caracs,palavras,linhas);
}
```

14. Esse é um exercício baseado no problema PAR (Par ou ímpar) , obtido de:

`http://br.spoj.pl/problems/PAR`

Há uma modificação motivada pelo fato de que não vamos usar ainda leitura de cadeias de caracteres, e por isso os nomes dos jogadores correspondentes a escolha "par"e "ímpar"são substituídos respectivamente por "Par"e "Impar". O problema é descrito a seguir.

O problema consiste em determinar, para cada jogada de partidas do jogo *Par ou Ímpar*, o vencedor da jogada.

A entrada representa uma sequência de dados referentes a partidas de *Par ou Ímpar*. A primeira linha de cada partida contém um inteiro n, que indica o número de jogadas da partida. As n linhas seguintes contêm cada uma dois inteiros a e b que representam o número escolhido por cada jogador ($0 \leq a \leq 5$ e $0 \leq B \leq 5$). O final da entrada é indicado por $n = 0$.

A saída deve conter para cada partida, uma linha no formato *Partida i*, onde i é o número da partida: partidas são numeradas sequencialmente a partir de 1. A saída deve conter também uma linha para cada jogada de cada partida, contendo *Par* ou *Impar* conforme o vencedor da partida seja o jogador que escolheu par ou ímpar, respectivamente. Deve ser impressa uma linha em branco entre uma partida e outra.

Por exemplo, para a entrada:

```
3
2 4
3 5
1 0
2
1 5
2 3
0
```

A saída deve ser:

```
Partida 1
Par
Par
Impar

Partida 2
Par
Impar
```

Solução: O programa abaixo define e usa a função *processaPartida* para separar o processamento (cálculo e impressão) de valores de cada partida, e a função *par*, que determina se um dado valor é par ou não. O número de cada partida é armazenado em uma variável, que tem valor inicial igual a 1 e é incrementada após o processamento de cada partida.

O programa usa também o recurso de definir a assinatura (ou interface, ou cabeçalho) de cada função definida, para usar (chamar) a função antes de defini-la. Na definição da interface o nome dos parâmetros é opcional, e é omitido no programa abaixo.

```
#include <stdio.h>
#include <stdlib.h>
void processaPartida(int,int);
int par(int);
int main() {
  int numPartida = 1, numJogadas;
  while (1) {
    scanf("%d", &numJogadas);
    if (numJogadas == 0) break;
    processaPartida(numPartida,numJogadas);
    numPartida++;
  }
}
void processaPartida(int numPartida, int numJogadas) {
  int mao1, mao2;
  printf("Partida %d\n", numPartida);
  for ( ; numJogadas>0; numJogadas--) {
    scanf("%d%d", &mao1, &mao2);
    printf("%s\n",par(mao1+mao2) ? "Par" : "Impar");
  }
  printf("\n");
}
int par(int valor) { return valor % 2 == 0; }
```

15. Nesse exercício vamos apresentar uma solução para o problema RUMO9S (Rumo aos 9s) , obtido de

`http://br.spoj.pl/problems/RUMO9s/`

A solução apresentada não usa arranjo nem cadeia de caracteres (abordados no próximo capítulo), para ler, armazenar e imprimir os valores de entrada. Em vez disso, caracteres são lidos um a um (números não podem ser lidos e armazenados como valores inteiros porque podem ter até 1000 dígitos decimais, e portanto não poder ser armazenadas como valores de tipo `int` ou `long int`). O problema é descrito a seguir.

Um número inteiro é múltiplo de nove se e somente se a soma dos seus dígitos é múltiplo de 9. Chama-se grau-9 de um número inteiro $n \geq 0$ o valor igual a 1 se $n = 9$, 0 se $n < 9$, e 1 mais o grau-9 da soma de seus dígitos, se $n > 9$.

Escreva um programa que, dado uma sequência de inteiros positivos, imprime se cada um deles é múltiplo de nove e, em caso afirmativo, seu grau-9. A entrada, no dispositivo de entrada padrão, contém uma sequência de inteiros positivos, um em cada linha, e termina com o valor 0. Exemplo:

Entrada:

```
999
27
9
998
0
```

Saída:

```
999 e' multiplo de 9 e seu grau-9 e' 3.
27 e' multiplo de 9 e seu grau-9 e' 2.
9 e' multiplo de 9 e seu grau-9 e' 1.
998 nao e' multiplo de 9.
```

Solução: A solução usa a função *isdigit* para testar se um dado caractere é um dígito (i.e. em `C`, um inteiro sem sinal, entre `'0'` e `'9'`).

A função *somaEImprimeCaracs* lê e imprime os dígitos contidos em uma linha da entrada padrão, e retorna o inteiro correspondente. Para isso, ela subtrai converte cada caractere lido no inteiro correspondente, subtraindo o caractere `'0'` (uma vez que os caracteres são ordenados, a partir de `'0'`, no código ASCII, usado em `C` para representação de caracteres, como valores inteiros).

```
#include <stdio.h>
#include <stdlib.h>
int somaEImprimeCaracs() {
  char c;
  scanf("%c",&c);
  if (c=='0') return 0;
  int soma=0;
  while (isdigit(c)) {
    soma += c - '0';
    printf("%c",c); scanf("%c",&c);
  }
  return soma;
}
int somaDigs(int n) {
  if (n<10) return n; else return (n%10 + somaDigs(n/10));
}
int grau9(int n) {
  if (n<10) return (n==9 ?  1 :  0);
  else { int grau = grau9(somaDigs(n));
         return (grau == 0?  0 :  1 + grau);
  }
}
int main() {
  int v, grau9v;
  while (1) {
    int n = somaEImprimeCaracs();
    if (n==0) break;
    int grau9n = grau9(n);
    if (grau9n==0) printf("is not a multiple of 9.\n", n);
    else printf("is a multiple of 9 and has 9-degree %d.\n", grau9n);
  }
  return 0;
}
```

4.6 Exercícios

1. Escreva três definições de função, chamadas *somaIter*, *somaRec* e *soma*, tais que, dados dois números inteiros positivos a e b, retorne o valor a + b. As duas primeiras definições devem usar apenas as operações mais simples de incrementar 1 e decrementar 1 (devem supor que as operações de adicionar e de subtrair mais de uma unidade não são disponíveis). A primeira definição deve usar um comando de repetição, e a segunda definição deve ser recursiva. A terceira definição deve usar o operador `+` de adição.

Inclua essas três definições em um programa de teste, e defina um programa de teste que leia vários valores a a b do dispositivo de entrada padrão, e não imprima nada se e somente se, para cada par de valores a e b lidos, os resultados retornados pelas execuções das três definições (*somaIter*, *somaRec* e *soma*) forem iguais. Se existir um par de valores a e b lido para o qual o resultado retornado pela execução das três definições não é igual, o programa deve terminar e uma mensagem deve ser impressa, informando o par de valores a, b para o qual o resultado da execução foi diferente, assim como o resultado retornado por cada definição, junto com o nome da função que retornou cada resultado.

A leitura deve terminar quando não houver mais valores a serem lidos (em entrada interativa, quando o caractere que indica fim dos dados for digitado).

2. Escreva um programa que leia pares de números inteiros *positivos* a,b e imprima, para cada par lido, o número de vezes que uma divisão inteira pode ser realizada, começando com o primeiro número como dividendo e usando sempre o segundo como divisor, e substituindo-se o dividendo pelo quociente na vez seguinte. A leitura deve terminar quando um dos valores lidos for menor ou igual a zero.

O programa deve definir e usar uma função *numdiv* que recebe dois números inteiros positivos e retorna o número de vezes que uma divisão exata pode ser realizada, neste processo de substituir o dividendo pelo quociente. O processo de divisões sucessivas deve terminar quando uma divisão exata não existe.

Exemplos: *numdiv*`(8,2)` deve retornar `3` e *numdiv*`(9,2)` deve retornar `0`.

3. O número de combinações de n objetos p a p — ou seja, o número de maneiras diferentes de escolher, de um conjunto com n elementos, um subconjunto com p elementos — denotado por $\binom{n}{p}$, é dado pela fórmula:

$$\frac{n(n-1)\dots(n-p+1)}{p!}$$

Por exemplo, o número de combinações de 4 objetos, 2 a 2, é igual a $\frac{4\times 3}{2!} = 6$ (se representamos os objetos por números, as combinações são `{1,2}`, `{1,3}`, `{1,4}`, `{2,3}`, `{2,4}` e `{3,4}`).

Defina uma função que, dados n e p, calcule o número de combinações de n objetos p a p.

Observação: A fórmula acima pode ser escrita também na forma:

$$\frac{n!}{p! \times (n-p)!}$$

No entanto, note que uma implementação baseada diretamente nessa última seria menos eficiente do que uma implementação baseada diretamente na primeira, uma vez que o número de operações de multiplicação necessárias para o cálculo seria maior nesse último caso.

Escreva um programa que leia vários pares de números inteiros positivos n, p e imprima, para cada par lido, o número de combinações existentes de n objetos p a p. A leitura deve terminar quando um dos valores lidos for menor ou igual a zero.

4. Escreva uma função para calcular qual seria o saldo de sua conta de poupança depois de 5 anos, se você depositou 1000 reais no início desse período e a taxa de juros é de 6% ao ano.

5. Generalize a questão anterior, de maneira que se possa especificar quaisquer valores inteiros como capital inicial, taxa de juros e prazo desejados.

Escreva um programa que leia, repetidamente, três valores inteiros positivos c, j, t que representam, respectivamente, o capital inicial, a taxa de juros anual e o número de anos de depósito, e imprima, para cada três valores lidos, o saldo final da conta, calculado usando a função acima. A leitura deve terminar quando um dos três valores lidos for menor ou igual a zero.

6. Defina funções que, dado o número de termos n, calcule:

(a) $\sum_{i=1}^{n} i^2$

(b) $\frac{1}{1} + \frac{3}{2} + \frac{5}{3} + \frac{7}{4} + \ldots$

(c) $\frac{1}{1} - \frac{2}{4} + \frac{3}{9} - \frac{4}{16} + \frac{5}{25} - \ldots$

(d) $\sum_{i=0}^{n} (\frac{i}{i!} - \frac{i^2}{(i+1)!})$

(e) $\frac{\pi}{4} = \frac{2 \times 4 \times 4 \times 6 \times 6 \times 8 \times \ldots}{3 \times 3 \times 5 \times 5 \times 7 \times 7 \times \ldots}$

Cada somatório deve ser implementado usando i) um comando de repetição e ii) uma função recursiva.

Escreva um programa que leia repetidamente pares de valores inteiros n, k e imprima, para cada par lido, o resultado de chamar a k-ésima função acima (k variando de 1 a 5) com o argumento n (que representa o número de termos). A leitura deve terminar quando $n \leq 0$ ou quando k não for um valor entre 1 e 5.

7. Escreva funções para calcular um valor aproximado do seno e cosseno de um ângulo dado em radianos, usando as seguintes fórmulas:

$$\operatorname{sen}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$$

Cada função deve receber o valor de x em radianos e o número de parcelas a serem usadas no somatório.

A definição não deve ser feita calculando o fatorial e a exponencial a cada parcela, mas sim de modo que o valor do numerador e do denominador de cada parcela sejam obtidos a partir dos valores respectivos da parcela anterior.

Escreva um programa que leia, do dispositivo de entrada padrão, um número inteiro positivo n, em seguida vários números de ponto flutuante (um a um), que representam valores de ângulos em graus e, para cada valor, imprima o seno e o cosseno desse valor, usando as funções definidas acima com o número de parcelas igual a n. Note que você deve converter graus em radianos nas chamadas às funções para cálculo do seno e cosseno, e que essas funções devem ter um parâmetro a mais que indica o número de parcelas a ser usado.

A entrada deve terminar quando um valor negativo ou nulo for lido.

8. Faça um programa que leia uma sequência de valores inteiros diferentes de zero, separados por espaços ou linhas, e imprima os valores pares dessa sequência. Um valor é par se o resto da divisão desse valor por 2 é igual a zero. A entrada termina quando um valor igual a zero for lido.

Por exemplo, para a entrada:

```
1 72 20 15 24 10 3 0 13 14
```

A saída deve conter (os números pares da entrada antes do primeiro zero, ou seja):

```
72 20 24 10
```

9. Faça um programa para imprimir a seguinte tabela:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
...									
10	20	30	40	50	60	70	80	90	100

10. Escreva um programa que leia um número inteiro positivo n e imprima um triângulo como o mostrado abaixo, considerando que o número de linhas é igual a n.

```
      *
     ***
    *****
   *******
  *********
 ***********
*************
```

11. Escreva um programa que leia um número inteiro positivo n e imprima um losango como o mostrado abaixo, considerando que o número de linhas é igual a n ($n = 13$ para o losango mostrado abaixo). Se n for par, as duas linhas no meio do losango devem ter o mesmo número de asteriscos.

```
      *
     ***
    *****
   *******
  *********
 ***********
*************
 ***********
  *********
   *******
    *****
     ***
      *
```

12. Defina uma função que converta o valor de uma temperatura dada em graus Fahrenheit para o valor correspondente em graus centígrados (ou Celsius). A conversão é dada pela fórmula:

$$T_C = \frac{5 \times (T_F - 32)}{9}$$

Use a função definida acima como parte de um programa que receba como argumentos o valor de uma temperatura inicial, o valor de uma temperatura final e um passo p (valor de incremento), e imprima uma tabela de conversão de graus Fahrenheit em graus centígrados, desde a temperatura inicial até o maior valor que não ultrapasse a temperatura final, de p em p graus Fahrenheit.

13. Os números mostrados na tabela a seguir formam a parte inicial do chamado *triângulo de Pascal* (nome dado em homenagem a Blaise Pascal (1623–1662), que escreveu um influente tratado sobre esses números). A tabela contém os valores das combinações de n elementos, p a p, para valores crescentes de n e p.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$	$\binom{n}{8}$	$\binom{n}{9}$	$\binom{n}{10}$
0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1

As entradas em branco nessa tabela têm, de fato, valor igual a zero, tendo sido deixadas em branco para evidenciar o triângulo formado pelas demais entradas da tabela.

Escreva um programa para imprimir o triângulo de Pascal, usando o fato de que

$$\binom{n}{p+1} = \frac{\binom{n}{p} \times (n-p)}{p+1}$$

Observação: A fórmula acima pode ser deduzida facilmente de

$$\binom{n}{p} = \frac{n!}{p\,! \times (n-p)!}$$

14. Escreva um programa que leia quatro valores inteiros positivos nA, nB, tA e tB — representando respectivamente as populações atuais de dois países A e B e as taxas de crescimento anual dessas populações — e determine o número de anos necessários para que a população do país A ultrapasse a de B, supondo que as taxas de crescimento dessas populações não variam e que nA `<` nB e tA `>` tB.

15. Escreva um programa que leia, do dispositivo de entrada padrão, um texto qualquer, caractere a caractere e imprima, no dispositivo de saída padrão, i) o número de vogais, ii) o número de consoantes e iii) o número de outros caracteres diferentes de vogais e consoantes presentes em palavras: considere que estes são todos os demais caracteres que não sejam espaço, fim-de-linha (`'\n'`) ou tab (caractere de tabulação, i.e. `'\t'`).

A entrada termina com indicação de fim dos dados de entrada (em entrada interativa, `Control-z` seguido de `Enter` no Windows, ou `Control-d` no Linux).

Dicas: Use *scanf* com formato `%c` para ler um caractere, e use o valor retornado por *scanf* para verificar fim dos dados de entrada: *scanf* retorna `-1` para indicar fim dos dados.

Defina e use função que retorna verdadeiro se e somente se o caractere passado como argumento é uma letra; para defini-la, teste se tal caractere está entre `'a'` e `'z'` ou entre `'A'` e `'Z'`.

16. Modifique o programa da questão anterior de modo a eliminar a suposição de que nA `<` nB e calcular, nesse caso, se a menor população vai ou não ultrapassar a maior e, em caso afirmativo, o número de anos necessário para que isso ocorra (em caso negativo, o programa deve dar como resultado o valor `0`).

17. Resolva o problema `BIT` disponível em:

`http://br.spoj.pl/problems/BIT/`

O enunciado é apresentado, de forma resumida, a seguir.

O problema consiste em escrever um programa para calcular e imprimir, para cada valor inteiro positivo lido, quantas notas de 50, 10, 5 e 1 reais são necessáias para para totalizar esse valor, de modo a minimizar a quantidade de notas.

A entrada é composta de vários conjuntos de teste. Cada conjunto de teste é composto por uma única linha, que contém um número inteiro positivo v, que indica o valor a ser considerado. O final da entrada é indicado por $v = 0$.

Para cada conjunto de teste da entrada seu programa deve produzir três linhas na saída. A primeira linha deve conter um identificador do conjunto de teste, no formato `"Teste n"`, onde `n` é o número do teste; os testes são numerados sequencialmente a partir de 1. Na segunda linha devem aparecer quatro inteiros, que representam o resultado encontrado pelo seu programa: o primeiro inteiro indica o número de notas de 50 reais, o segundo o número de notas de 10 reais, o terceiro o número de notas de 5 reais e o quarto o número de notas de 1 real. A terceira linha deve ser deixada em branco.

Por exemplo, para a entrada:

```
1
72
0
```

A saída deve ser:

```
Teste 1
0 0 0 1

Teste 2
1 2 0 2
```

18. Resolva o problema `ALADES` disponível em:

`http://br.spoj.pl/problems/ALADES/`

O enunciado é apresentado, de forma resumida, a seguir.

O problema consiste em escrever um programa que, dados valores de hora e minutos corrente e hora e minutos de um alarme, determinar o número de minutos entre os dois valores.

A entrada contém vários casos de teste. Cada caso de teste é descrito em uma linha, contendo quatro números inteiros h_1, m_1, h_2 e m_2, sendo que $h_1 : m_1$ representa hora e minuto atuais, e h_2:m_2 representa hora e minuto para os quais um alarme foi programado ($0 \leq h_1 < 24, 0 \leq m_1 < 60, 0 \leq h_2 < 24, 0 \leq m_2 \leq 60$). O final da entrada é indicado por uma linha que contém apenas quatro zeros, separados por espaços em branco. Os dados devem ser lidos da entrada padrão.

Para cada caso de teste da entrada, deve ser impressa uma linha, no dispositivo de saída padrão, contendo um número inteiro que indica o número de minutos entre os dois horários.

Por exemplo, para a entrada:

```
1 5 3 5
23 59 0 34
21 33 21 10
0 0 0 0
```

A saída deve ser:

```
120
35
1417
```

Escreva um programa que leia um texto (sequência de caracteres) do dispositivo de entrada padrão e imprima, no dispositivo de saída padrão, qual é a vogal ou quais são as vogais mais frequentes no texto: o programa deve imprimir quais foram as vogais com maior frequência, se existir uma ou mais de uma com a mesma maior frequência.

A ocorrência de uma vogal pode ser em letra minúscula ou maiúscula; em ambos os casos o número de ocorrências dessa vogal deve ser incrementado.

A impressão das vogais com maior frequência pode usar a vogal minúscula ou maiúscula (escolhendo é claro um dos casos para todas as vogais com maior frequência).

A leitura deve ser feita caractere a caractere, usando a função *getChar*. A leitura deve terminar quando *getChar* retornar *EOF*.

getChar retorna um inteiro: *EOF* quando não há mais dados a serem lidos, caso contrário retorna o caractere lido (i.e. a representação do caractere no código ASCII).

Use 5 variáveis para armazenar a frequência de cada vogal, e uma outra variável para indicar a maior frequência. Imprima as vogais com frequência igual a essa maior frequência.

19. O mínimo múltiplo comum (*mmc*) entre dois ou mais números é (como o próprio nome diz) o menor inteiro que é múltiplo de todos eles. Por exemplo, *mmc*`(4,6)` é igual a `12`. O máximo divisor comum (*mdc*) de dois ou mais números inteiros é (como o próprio nome diz) o maior divisor de todos eles (i.e. o maior valor que divide exatamente os números). Por exemplo, *mdc*`(4,6)` é igual a `2`.

Escreva um programa que leia uma sequência qualquer de números inteiros positivos e imprima o mínimo múltiplo comum entre eles.

O programa deve ler os valores da entrada padrão e imprimir o resultado na saída padrão. Ele deve funcionar para entradas não interativas. Ou seja, a entrada pode estar em arquivo, especificado por redirecionamento da entrada padrão. O programa deve terminar com o fim dos dados de entrada (i.e. o término da entrada é indicado pelo valor retornado por `scanf`).

O seu programa deve ser baseado nos fatos de que:

(a) $mmc(a,b) = \big(a/mdc(a,b)\big) \times b$
Ou seja, use o valor de $mdc(a,b)$ para calcular $mmc(a,b)$, dividindo a por $mdc(a,b)$ e multiplicando por b.

(b) *mmc* de três ou mais números pode ser calculado usando o fato de que: $mmc(a,b,c) = mmc(mmc(a,b),c)$.

Ou seja: para calcular *mmc* de três ou mais números, obtenha o *mmc* do resultado de calcular o *mmc* dos primeiros com o último.

(c) O cálculo do *mdc* de dois números a e b deve ser feito usando o algoritmo definido no Exercício Resolvido 5.

Capítulo 5

Arranjos

Abordamos a seguir, nos Capítulos 5 a **??**, a definição e o uso de estruturas de dados, que são valores compostos por valores mais simples: arranjos (capítulo 5), ponteiros (capítulo 6) e registros (ou, como são chamados em `C`, *estruturas*, capítulo 7). Uma introdução à definição e uso de estruturas de dados encadeadas, formadas com o uso de registros com ponteiros, são abordadas na seção 7.

Arranjos são estruturas de dados homogêneas, devido ao fato de que os componentes têm que ser todos de um mesmo tipo, enquanto estruturas de dados encadeadas e registros em geral são estruturas de dados heterogêneas, que podem envolver componentes de vários tipos, diferentes entre si. Como veremos na seção 5.5, cadeias de caracteres são, na linguagem `C`, arranjos terminados com um caractere especial (o caractere `'\0'`).

Arranjos são estruturas de dados muito usadas em programas. Um arranjo é uma forma de representar uma função finita (função de domínio finito — que em geral é vista como uma tabela ou uma seqüência finita de valores — com a característica de que o acesso aos seus componentes podem ser feitos de modo eficiente. Esta seção aborda a definição e uso de arranjos na linguagem `C`.

Um *arranjo* é é uma estrutura de dados formada por um certo número finito de componentes (também chamados de *posições* do arranjo) de um mesmo tipo, sendo cada componente identificado por um *índice*.

Um arranjo tem um *tamanho*, que é o número de componentes do arranjo. A uma variável de tipo *arranjo* de um tipo T e tamanho n correspondem n variáveis de tipo T.

Em `C`, os índices de um arranjo são sempre inteiros que variam de `0` a n`-1`, onde n é o tamanho do arranjo.

Se v é uma expressão que representa um arranjo de tamanho n, e i é uma expressão de tipo `int` com valor entre `0` e n`-1`, então v`[`i`]` representa um componente do arranjo v.

Nota sobre uso de índices fora do limite em `C`: A linguagem `C` não especifica que, em uma operação de indexação (uso de um valor como índice) de um arranjo, deva existir uma verificação de que esse índice é um índice válido. A linguagem simplesmente deixa a responsabilidade para o programador. Se o índice estiver fora dos limites válidos em uma indexação, uma área de memória distinta da área alocada para o arranjo será usada, ou o programa é interrompido, com uma mensagem de que um erro ocorreu devido a um acesso ilegal a uma área de memória que não pode ser usada pelo processo corrente. Se o índice for inválido mas estiver dentro da área reservada ao processo corrente, nenhum erro em tempo de execução será detectado. O erro devido a um acesso ilegal a uma área de memória não reservada ao processo corrente provoca a emissão da mensagem *segmentation fault*, e a interrupção da execução do processo corrente. O motivo de não existir verificação de que um índice de um arranjo está ou não entre os limites desse arranjo é, obviamente, eficiência (i.e. evitar gasto de tempo de execução). O programador deve estar ciente disso e atento de modo a evitar erros (i.e. evitar o uso de índices fora dos limites válidos em indexações de arranjos).

A eficiência que existe no acesso a componentes de um arranjo se deve ao fato de que arranjos são geralmente armazenados em posições contíguas da memória de um computador, e o acesso à i-ésima posição é feito diretamente, sem necessidade de acesso a outras posições. Isso espelha o funcionamento da memória de computadores, para a qual o tempo de acesso a qualquer endereço de memória é o mesmo, ou seja, independe do valor desse endereço. Um arranjo é, por isso, chamado de uma estrutura de dados *de acesso direto* (ou *acesso indexado*). Ao contrário, em uma estrutura de dados *de acesso sequencial*, o acesso ao i-ésimo componente requer o acesso aos componentes de índice inferior a i.

5.1 Declaração e Criação de Arranjos

Em C, arranjos são criados no instante da declaração de variáveis do tipo arranjo. Uma variável de tipo arranjo em C armazena na verdade não um valor de tipo arranjo mas o endereço do primeiro componente do arranjo que ela de fato representa. A versão C-99 da linguagem permite a declaração de arranjos dentro de funções com tamanho que é conhecido apenas dinamicamente, mas em geral uma variável de tipo arranjo tem um valor conhecido estaticamente (em tempo de compilação) ou é um parâmetro de uma função, sendo o tamanho nesse caso igual ao tamanho do argumento, especificado no instante da chamada à função. Usaremos, para criação de arranjos com tamanho conhecido apenas dinamicamente, a declaração de um ponteiro. Isso será explicado mais detalhadamente na seção 6.2.

O tamanho de um arranjo não faz parte do seu tipo (em geral, esse tamanho não é conhecido estaticamente), mas não podendo ser modificado. Considere os seguintes exemplos:

```
int ai[3];
char ac[4];
```

As declarações acima criam as variáveis *ai* e *ac*. A primeira é um arranjo de 3 inteiros e a segunda um arranjo de 4 caracteres.

A declaração dessas variáveis de tipo arranjo envolvem também a criação de um valor de tipo arranjo. Na declaração de *ai*, é criada uma área de memória com tamanho igual ao de 3 variáveis de tipo `int`. Similarmente, na declaração de *ac*, é criada uma área de memória com tamanho igual ao de 4 variáveis de tipo `char`. Os valores contidos nessas áreas de memória não são conhecidos: são usados os valores que estão já armazenados nessas áreas de memória.

5.2 Arranjos criados dinamicamente

A função *malloc*, definida na biblioteca *stdlib*, aloca dinamicamente uma porção de memória de um certo tamanho, passado como argumento da função, e retorna o endereço da área de memória alocada. Esse endereço é retornado com o valor de um ponteiro para o primeiro componente do arranjo. Ponteiros são abordados mais detalhadamente na seção 6. Por enquanto, considere apenas que *malloc* aloca uma área de memória — na *área de memória dinâmica* do processo corrente, chamada em inglês de *heap* — que será usada tipicamente por meio da operação de indexação do arranjo. Considere os seguintes comandos:

```
int *pi;
char *pc;
pi = malloc(3 * sizeof(int));
pc = malloc(4 * sizeof(char));
```

A chamada *malloc*`(3*sizeof(int))` alloca uma área de memória de tamanho `3*(sizeof(int))`, sendo `sizeof(int)` o tamanho de uma área de memória ocupada por um valor de tipo `int`. Similarmente, a chamada *malloc*`(4 * sizeof(char))` alloca uma área de memória de tamanho `4*(sizeof(char))`, sendo `sizeof(char)` o tamanho de uma área de memória ocupada por um valor de tipo `char`.

Após a atribuição à variável *pi* acima, *pi* contém o endereço da primeira posição de um arranjo com 3 componentes de tipo `int`. Analogamente, após a atribuição à variável *pc* acima, *ac* contém o endereço da primeira posição de um arranjo com 4 componentes de tipo `char`.

```
/*********************************************************
* Lê n, depois n inteiros do dispositivo de entrada padrão  *
* e imprime os n inteiros lidos em ordem inversa.           *
*********************************************************/

#include stdio.h;

int main() {
  int *arr, i=0, n;
  scanf("%d", &n);
  arr = malloc(n * sizeof(int));
  for (i=0; i<n; i++) scanf("%d", &arr[i]);
  for (i--; i>=0; i--) printf("%d ", arr[i]);
}
```

Figura 5.1: Exemplo de uso de comando `for` para percorrer arranjo

Note que, no caso da declaração de variáveis com um tipo que é explicitamente indicado como sendo um tipo arranjo, para o qual o tamanho é indicado explicitamente (como no exemplo anterior das variáveis *ai* e *ac*), o arranjo não é alocado na área dinâmica mas na área de pilha da função na qual a declaração ocorre (no caso, na área de memória alocada quando a execução da função *main* é iniciada).

O tamanho de uma área de memória pode ser obtido em `C` por meio do uso da função predefinida `sizeof`. A palavra reservada `sizeof` pode ser seguida de um nome de tipo (como nos exemplo acima) ou por uma expressão. O resultado retornado pela avaliação de `sizeof` é o tamanho em bytes do tipo ou expressão usada como "argumento" (tamanho do tipo ou tamanho do tipo da expressão, respectivamente). No caso de uso de um tipo, ele deve ser colocado entre parênteses, mas quando uma expressão é usada, ela pode seguir `sizeof` sem necessidade de parênteses (respeitando-se a precedência de operadores e chamadas de funções).

Por exemplo, depois da atribuição acima, `sizeof` *pi* retorna o mesmo que `sizeof(3 * sizeof(int))`.

A linguagem `C` usa colchetes em declaração de arranjos após o nome da variável (por exemplo, a declaração:

```
int a[5];
```

declara uma variável *a* de tipo arranjo, mas o tipoa `int` ocorre antes e a indicação de que esse tipo é um arranjo de componentes de tipo `int` ocorre após o nome da variável.

Isso é feito com o mero intuito de permitir declarações um pouco mais sucintas, como a seguinte, que cria uma variável *b* de tipo `int` e um arranjo *a* com componentes de tipo `int`:

```
int b, a[5];
```

5.3 Exemplo de Uso de Arranjo Criado Dinamicamente

Como exemplo do uso de arranjo criados dinamicamente, vamos considerar o problema de ler um inteiro não-negativo n, em seguida n valores inteiros e imprimir os n inteiros na ordem inversa à que foram lidos.

Por exemplo, se forem lidos o inteiro `3`, em seguida três inteiros i_1, i_2 e i_3, o programa deve imprimir i_3, i_2, i_1, nesta ordem. Uma solução é mostrada na Figura 5.1.

O programa da Figura 5.1 ilustra a operação básica de "percorrer" um arranjo para realização de alguma operação sobre os valores armazenados no arranjo. O programa declara o tipo da variável *arr* não como um arranjo de componentes de tipo `int`, mas como um ponteiro para variáveis do tipo `int`. Isso ocorre porque o tamanho do arranjo só é conhecido dinamicamente, sendo sua alocacão feita na área de memória dinâmica pela função *malloc*, que retorna um endereço para a

```
void preenche(int arr[], int tam, int valor) {
  // Preenche todas as posicoes de arr com valor
  int i;
  for (i=0; i<tam; i++) arr[i] = valor;
}

int iguais(int arr1[], int tam1, int arr2[], int tam2) {
  // Retorna verdadeiro sse arr1 = arr2, componente a componente.
  int i;
  if (tam1 == tam2)
    for (i=0; i<tam1; i++)
      if (arr1[i] == arr2[i]) return 0;
    return 1;
  else return 0;
}
```

Figura 5.2: Operações comuns em arranjos

área de memória alocada. No entanto, a operação de indexação de arranjos funciona normalmente também no caso de variáveis ou valores de tipo ponteiro. Mais detalhes sobre a relação entre arranjos e ponteiros em `C` estão na seção 6.

Outra observação importante é referente à necessidade de se especificar o tamanho do arranjo, ou seja, o número de valores inteiros a ser digitado, antes da leitura dos valores inteiros. Para evitar isso, há duas opções: i) não usar um arranjo, mas uma estrutura de dados encadeada, como descrito na seção 7, ou ii) adotar um certo valor como máximo para o número de valores a serem digitados (de modo a alocar um arranjo com tamanho igual a esse número máximo). Essa opção tem as desvantagens de que um número máximo pode não ser conhecido estaticamente ou ser difícil de ser determinado, e o uso de um valor máximo pode levar a alocação desnecessária de área significativa de memória, que não será usada (ou seja, o máximo pode ser um valor muito maior do que o de fato necessário).

Para percorrer um arranjo, é usado tipicamente um comando `for`, pois o formato desse comando é apropriado para o uso de uma variável que controla a tarefa de percorrer um arranjo: iniciação do valor da variável usada para indexar o arranjo, teste para verificar se seu valor é ainda um índice válido do arranjo, e atualização do valor armazenado na variável.

5.4 Operações Comuns em Arranjos

A Figura 5.2 apresenta exemplos de funções que realizam operações básicas bastante comuns sobre arranjos de componentes de um tipo específico, `int`. As funções são: i) preencher todos os componentes de um arranjo, passado como argumento, com um valor, também passado como argumento, e ii) testar igualdade de arranjos, passados como argumentos da função.

A primeira operação é uma função com "efeito colateral". Isso significa que ela não é uma função que tem como domínio e contra-domínio os tipos anotados na definição da função, mas precisa, para poder ser considerada como função, que o domínio e contra-domínio abranjam (além dos parâmetros explicitamente indicados na definição da função) também uma forma de representação do *estado* da computação. O estado da computação pode ser representado na forma de uma função que associa variáveis a valores armazenados nessas variáveis.

Nas duas funções acima, *preenche* e *iguais*, o tamanho do arranjo (número de posições alocadas) é um parâmetro da função. Isso é feito para evitar o uso da função predefinida `sizeof` de `C` com argumento que é um arranjo alocado dinamicamente: esta função, anteriormente à definição do padrão `C-99`, só podia ser usada quando o tamanho do arranjo era conhecido em tempo de compilação.

Note que o uso de `==` não é adequado para comparar igualdade de dois arranjos em `C` (i.e. com-

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int *arr1, *arr2, n1, n2, i;
  printf("Digite inteiro positivo n1, n1 valores inteiros, ");
  printf("inteiro positivo n2, n2 valores inteiros\n");
  scanf("%d", &n1);
  arr1 = malloc(n1 * sizeof(int));
  for (i=0; i<n1; i++) scanf("%d", &(arr1[i]));
  scanf("%d", &n2);
  arr2 = malloc(n2 * sizeof(int));
  for (i=0; i<n2; i++) scanf("%d", &arr2[i]);
  printf("Sequencia 1 de valores inteiros e' %s sequencia 2 de valores inteiros.",
         iguais(arr1,n1,arr2,n2) ?  "igual a":  "diferente da");
  system("PAUSE");
  return 0;
}
```

Figura 5.3: Exemplo de uso de função que testa igualdade entre arranjos

parar se dois arranjos têm o mesmo número de componentes e os conteúdos dos componentes em cada índice dos dois arranjos são iguais). O teste de igualdade de valores de tipo arranjo com `==` se refere a comparação apenas de ponteiros (i.e. comparação entre se os endereços da primeira posição do primeiro e do segundo arranjo são iguais).

O programa da Figura 5.3 ilustra o uso da função *iguais* definida acima, escrevendo um programa que lê, nesta ordem, um valor inteiro positivo n, $2 \times n$ valores inteiros, e em seguida imprime o resultado de testar se os n primeiros dos 2 *timesn* valores lidos são iguais aos n últimos (ou seja, se o primeiro é igual ao n-ésimo mais um, o segundo é igual ao n-ésimo mais dois, etc.).

5.5 Cadeias de caracteres

Em `C`, uma cadeia de caracteres — em inglês, um *string* — é um arranjo de caracteres que segue a convenção de que o último componente é o caractere `'\0'` (chamado de caractere nulo). O caractere `'\0'` é inserido automaticamente em literais de tipo cadeia de caracteres, escritos entre aspas duplas.

Por exemplo:

```
char str[] = "1234"
```

é o mesmo que:

```
char str[5] = "1234"
```

Note que são 5 componentes no arranjo *str*: o último componente, de índice 4, contém o caractere `'\0'`, e é inserido automaticamentre no literal de tipo cadeia de caracteres, e o tamanho do arranjo deve ser igual ao número de caracteres no literal mais 1.

O uso de cadeias de caracteres em `C`, e em particular a operação de ler uma cadeia de caracteres, deve levar em conta de que toda variável que é uma cadeia de caracteres deve ter um tamanho fixo. Para ler uma cadeia de caracteres, é necessário primeiro alocar espaço — um tamanho máximo — para armazenar os caracteres que vão ser lidos. No entanto, no caso de passagem de parâmetros para a função *main* (veja seção **??**), o sistema aloca, automaticamente, cadeias de caracteres de tamanho suficiente para armazenar os caracteres passados como parâmetros para a função *main*).

O uso de cadeias de caracteres em `C` envolve muitas vezes o uso de funções definidas na biblioteca *string*, dentre as quais destacamos as seguintes:

Assinatura	Significado
`int` *strlen*`(char *`*s*`)`	Retorna tamanho da cadeia *s*, excluindo caractere nulo no final de *s*
`char*` *strcpy*`(char *`*dest*`, const char *`*fonte*`)`	Copia cadeia apontada por *fonte*, incluindo caractere `'\0'` que indica terminação da cadeia, para *dest*; retorna *fonte*
`char*` *strcat*`(char *`*dest*`, const char *`*fonte*`)`	Insere cadeia apontada por *fonte*, incluindo caractere `'\0'` que indica terminação da cadeia, no final da cadeia apontada por *dest*, sobrepondo primeiro caractere de *fonte* com caractere nulo que termina *dest*; retorna cadeia *dest* atualizada
`int` *strcmp*`(const char *`*s1*`, const char *`*s2*`)`	Comparação lexicográfica entre cadeias de caracteres, sendo retornado `0` se as cadeias são iguais, caractere a caractere, valor negativo se *s1*`<`*s2*, lexicograficamente, e positivo caso contrário.

O uso do atributo `const` na declaração de parâmetros em assinaturas de funções acima significa que o parâmetro não é modificado no corpo da função, e é usado por questão de legibilidade.

A comparação do conteúdo de duas cadeias de caracteres não pode ser feita com o operador `==`, pois esse operador, se aplicado a valores de tipo `char*`, ou `char[]`, testa igualdade de ponteiros, e não do conteúdo apontado pelos ponteiros.

Em `C`, podem ocorrer erros difíceis de serem detectados se não for seguida a convenção de que uma cadeia de caracteres termina com o caractere nulo.

As funções acima não devem ser usadas se houver sobreposição de cadeias fonte e destino, sendo o comportamento dessas funções não especificado nesses casos.

A comparação lexicográfica entre duas cadeias *s1* e *s2* termina assim que ocorre uma diferença, e resulta em que *s1* `<` *s2* se o tamanho de *s1* é menor que o de *s2* ou se o caractere diferente de *s1* é menor do que o *s2*. A comparação entre caracteres é feita pelo valor da representação no código ASCII. Por exemplo: `"ab"` é menor que `"abc"` e que `"ac"`, e maior que `"aa"` e `"a"`.

5.5.1 Conversão de cadeia de caracteres para valor numérico

As seguintes funções da biblioteca *stdlib* podem ser usadas para converter uma cadeia de caracteres em um valor numérico:

Assinatura	Significado
`int` *atoi*`(char *)`	Converte cadeia de caracteres para valor de tipo `int`
`double` *atof*`(char *)`	Converte cadeia de caracteres para valor de tipo `float`
`int` *atol*`(char *)`	Converte cadeia de caracteres para valor de tipo `long`

Por exemplo:

```
char *s1 = "1234";
char *s2 = "12.34";
char *s3 = "1234";
char *s4 = "123quatro";
char *s5 = "xpto1234";

int i;
float f;

i = atoi(s1); // i = 1234
f = atof(s2); // f = 12.34
i = atoi(s3); // i = 1234
i = atoi(s4); // i = 123
i = atoi(s5); // i = 0
```

Note que:

- espaços à esquerda na cadeia de caracteres são ignorados;
- caracteres inválidos após um numeral válido são ignorados;
- se a conversão não puder ser realizada, é retornado `0`.

5.5.2 Conversão para cadeia de caracteres

Para conversão de um valor numérico em uma cadeia de caracteres, a função *sprintf*, similar a *printf*, pode ser usada. Um uso de *sprintf* tem o seguinte formato:

$$sprintf\texttt{(}str\texttt{,}\ formato\texttt{,}\ v_1\texttt{,}\ \texttt{...,}\ v_n\ \texttt{)}$$

onde *formato* é um literal de tipo cadeia de caracteres de controle da operação de escrita na cadeia de caracteres *str* e v_1`,...,`v_n são argumentos (valores a serem escritos).

As especificações de controle em *formato* são feitas como no caso de *printf*, usando o caractere `%` seguido de outro caractere indicador do tipo de conversão a ser realizada.

A função *sprintf* tem um comportamento semelhante ao de *printf*, com a diferença de que os valores são escritos na cadeia de caracteres *str*, em vez de no dispositivo de saída padrão.

O tamanho da cadeia *str* deve ser suficiente para conter todos os resultados da conversão dos valores para uma cadeia de caracteres.

sprintf não pode ser usada quando se deseja o valor da cadeia de caracteres correspondente a um inteiro (pois *sprintf* é um comando, não retorna nenhum valor). Quando um valor é desejado, uma função pode ser definida pelo programador, ou pode ser usada a função *itoa*, disponível em grande parte das implementações da biblioteca `stdlib`, embora não seja definida na linguagem C padrão (ANSI C). A função *itoa* tem a seguinte assinatura:

```
char* itoa(int valor, char* str, int base)
```

itoa converte *valor* para uma cadeia de caracteres, terminada com o caractere `NULL`, usando a base *base*, armazena essa cadeia em *str*, e retorna *str*.

Se a base for 10 e *valor* for negativo, a cadeia resultante é precedida do sinal `'-'`. Em qualquer outro caso, simplesmente se supõe que o valor é positivo.

str deve ser um arranjo com um tamanho grande o suficiente para conter a cadeia.

5.5.3 Passando valores para a função *main*

A função *main* pode receber como argumento várias cadeias de caracteres, separadas entre si por espaços ou outros caracteres delimitadores de palavras (como tab, i.e. `'\ t'`). O interpretador da linha de comandos do sistema operacional, quando chamado com um nome de um programa seguido de uma cadeia de caracteres, percorre essa cadeia de caracteres colocando cada sequência de caracteres, separada da seguinte por um ou mais delimitadores, em uma posição de um arranjo que é passado como argumento para a função *main*, precedido do número de valores contidos neste arranjo.

Por exemplo, para um programa com nome *prog*, o comando:

```
prog abc xy 123
```

faz com que o interpretador da linha de comandos percorra a cadeia de caracteres `"abc xy 123"` e coloque cada sequência de caracteres que está separada da seguinte por um ou mais delimitadores em uma posição de um arranjo, e passe como argumento para o método *main* do programa (`prog`) o valor 3 — que é igual ao tamanho do arranjo (número de cadeias de caracteres) — e esse arranjo. Assim, o arranjo passado contém `"abc"` na variável de índice `0`, `"xy"` na variável de índice `1` e `"123"` na variável de índice 2.

5.5.4 Exercícios Resolvidos

1. Escreva um programa que leia um valor inteiro positivo n, em seguida uma cadeia de caracteres de tamanho menor que n e imprima a frequência de todos os caracteres que ocorrem nesta cadeia.

   Por exemplo, para a entrada:

   ```
   10
   112223a
   ```

   a saída deve ser:

   ```
   1 aparece 2 vezes
   2 aparece 3 vezes
   3 aparece 1 vez
   a aparece 1 vez
   ```

   A ordem de impressão dos caracteres e sua frequência não é importante, mas cada caractere deve aparecer, com sua frequência de ocorrência, apenas uma vez na saída, e somente se essa frequência for diferente de zero.

   Solução: Cada caractere é representado no código ASCII por um valor inteiro compreendido entre `0` e `127`. Para armazenar a informação sobre o número de ocorrências de cada um desses caracteres, podemos usar um arranjo de valores inteiros com `128` componentes, fazendo corresponder a cada caractere representado pelo número inteiro i a posição i desse arranjo. Tal arranjo pode ser declarado da seguinte forma:

   ```
   int max = 128;
   int *contchar = malloc(max * sizeof(char));
   ```

   Um trecho de programa que armazena em *contchar*`[`i`]` o número de ocorrências de cada caractere de uma cadeia de caracteres *str* pode ser escrito como a seguir, onde *tam_str* é o número de caracteres em *str*:

   ```
   void contFreqChar(const char str[], int contChar[], int tamanho_str) {
     int i;
     for (i=0; i<tamanho_str; i++)
       contChar[str[i]]++;
   }
   ```

O uso do atributo `const` na declaração do parâmetro *str* da função *contFreqChar* é usado apenas por questão de legibilidade, para indicar que a cadeia de caracteres *str* não é modificada no corpo da função.

Um exemplo de definição de uma função *main* que usa a função *contFreqChar* é mostrada a seguir. Os valores de entrada são lidos, a função *contFreqChar* é chamada e a frequência de cada caractere que ocorre na cadeia de caracteres especificada na entrada é impressa.

```
int main() {
  int tam;
  scanf("%d",&tam);
  char *s = malloc(tam * sizeof(char));
  scanf("%s",s);
  printf("Na cadeia de caracteres %s\n",s);
  int max = 128; int *contChar = malloc(max * sizeof(int));
  int i,freq;
  for (i=0; i<max; i++) contChar[i] = 0;
  contFreqChar(s,contChar,tam);
  for (i=0; i<max; i++)
    if (contChar[i] = 0) {
      freq = contChar[i];
      printf("%c aparece %d %s\n",(char)i,freq,freq==1?"vez":"vezes");
    }
  }
```

2. Escreva um programa para determinar a letra ou algarismo que ocorre com maior freqüência em uma cadeia de caracteres dada, com um tamanho máximo previamente fornecido, e imprimir esse algarismo no dispositivo de saída padrão.

   Solução: Um caractere alfanumérico é um caractere que pode ser um algarismo ou uma letra. A solução consiste em, inicialmente, determinar a freqüência de ocorrência de cada caractere alfanumérico, de maneira análoga à do exercício anterior. Ou seja, a solução armazena cria arranjos com componentes correspondentes a cada caractere alfanumérico. Cada componente contém a frequência de ocorrência do caractere alfanumérico correspondente. Em seguida, o índice do componente de maior valor (isto é, maior frequência) desse arranjo é determinado e o caractere alfanumérico correspondente a esse índice é impresso. Vamos usar três arranjos, um arranjo para dígitos — com índices de `0` a `9` —, e os outros para letras minúsculas e maiúsculas. O índice do arranjo de dígitos correspondente a um dígito *d* é obtido por *d* `- '0'`, usando o fato de que a representação dos dígitos são crescentes, a partir do valor da representação do caractere `'0'`. Analogamente para letras minúsculas e maiúsculas, que têm valores de representação crescentes a partir, respectivamente, dos valores das representações dos caractere `'a'` e `'A'`.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h> // define isdigit

int letraMinusc(char c) { return (c >= 'a' && c <= 'z'); }
int letraMaiusc(char c) { return (c >= 'A' && c <= 'Z'); }

void contFreqAlfaNums(const char s[], int tam, int contDigLets[],
                      int tamDigs, int tamLets) {
  int i; char c;
  for (i=0; i<tam; i++) {
    c = s[i];
    if (isdigit(c)) contDigLets[c - '0']++;
    else if (letraMinusc(c)) contDigLets[tamDigs + (c - 'a')]++;
    else if (letraMaiusc(c)) contDigLets[tamDigs + tamLets + (c - 'A')]++;
  }
}

char alfaNumMaisFreq(const char s[], int tams) {
  int tamDigs = 10, tamLets = 26, tam = tamDigs + 2*tamLets,
      *contAlfaNums = malloc (tam * sizeof(int)), i;
  for (i=0; i<tam; i++) contAlfaNums[i] = 0;
  contFreqAlfaNums(s,tams,contAlfaNums,tamDigs,tamLets);
  i = maisFreq(contAlfaNums,tam);
  return (i<tamDigs ?  i + '0' :
          i<tamDigs + tamLets ?  i + 'a' :  i + 'A');
  }

  int maisFreq(const int freq[], int tam) {
    int i, maxi, maxFreq = 0;
    for (i=0; i<tam; i++)
      if (freq[i] > maxFreq) {
        maxi = i; maxFreq = freq[maxi];
      }
    return maxi;
  }

  int main() {
    int tamstr;
    scanf("%d",&tamstr);
    char *str = malloc(tamstr * sizeof(char));
    scanf("%s",str);
    printf("Caractere alfanumerico mais frequente em %s e':  %c\n",str,
           maisFreq(str,tamstr));
  }
```

A função *maisFreq* recebe como argumento uma cadeia de caracteres, em que cada caractere representa um algarismo, e retorna o algarismo mais freqüente nessa cadeia. Quando existir mais de um algarismo com a maior freqüência de ocorrência, o método retorna, dentre esses, aquele que ocorre primeiro na cadeia. Por exemplo, *maisFreq*`("005552")` retorna `'5'`, e *maisFreq*`("110022")` retorna `'1'`.

3. Escreva um programa para resolver o exercício 15, página 72, considerando a condição de que inteiros podem ter até 1000 dígitos decimais.

   Solução: Usamos uma cadeia de caracteres de até 1001 dígitos para armazenar inteiros que podem ter até 1000 dígitos, e mais o caractere '\0' para indicar terminação da cadeia. A soma dos dígitos de um inteiro de até 1000 dígitos sempre pode ser armazenada em um valor de tipo `int`.

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

int somaDigsInt (int n) { return somaDigsInt1(n,0); }
int somaDigsInt1 (int n, int soma) {
  if (n==0) return soma;
  else return somaDigsInt(n/10, n%10 + soma);
}

int somaDigs (char* digs, int i) {
  if (digs[i] == '\0') return 0;
  else return (digs[i] - '0') + somaDigs(digs,i+1);
}

int grau9Int (int n) {
  if (n <= 9) return (n == 9? 1 : 0);
  else {
    int grau = grau9Int(somaDigsInt(n));
    return (grau == 0 ? grau : 1 + grau);
  }
}

int grau9 (char *digs) { return grau9Int(somaDigs(digs,0)); }

int main() {
  const int numDigs = 1001;
  char v[numDigs];
  while (1) {
    scanf("%s", &v);
    int i=0; while (i < numDigs && v[i] == '0') i++;
    if (i < numDigs && v[i]=='\0') break; // Termina se inteiro lido igual a zero

    int grau9v = grau9(v);
    printf("%s is%s a multiple of 9", v, grau9v==0 ? "not": );
    if (grau9v==0) printf("\n");
    else printf(" and has 9-degree %d\n",grau9v);
  }
  return 0;
}
```

5.5.5 Exercícios

1. Escreva uma função *inverte* que estenda o Exercício 5 da seção 3.12 para qualquer cadeia de caracteres s. A função *inverte* tem como parâmetro adicional (além da cadeia de caracteres) o tamanho n da cadeia passada como argumento. O programa que chama a função *inverte* deve ler, antes de cada cadeia de caracteres, um valor que, deve-se supor, é maior que o tamanho da cadeia.

2. Defina uma função *decPraBin* que receba um número inteiro não-negativo como argumento e retorne uma cadeia de caracteres que é igual à representação desse número em notação binária.

Por exemplo, ao receber o número inteiro `8`, a função deve retornar `"1000"`.

Para calcular o tamanho da cadeia de caracteres a ser alocada, defina e use uma função *numDiv2* que, ao receber um número inteiro positivo n como argumento, retorne $m + 1$ tal que $2^m \leq n < 2^{m+1}$. Esse número $(m+1)$ é igual ao número de vezes que n pode ser dividido por 2 até que o quociente da divisão seja zero. Por exemplo, $2^3 \leq 8 < 2^4$, e 4 é o número de caracteres da cadeia `"1000"`, necessários para representação de `8` na base 2.

Note que, para cada n, deve ser alocada uma cadeia de caracteres de tamanho *tam*`+2`, onde *tam* é o resultado de *numDiv2*`(`n`)`, para conter, além dos caracteres necessários para representação de n na base 2, o caractere `'\0'` (usado em `C` para terminação de cadeias de caracteres).

Escreva um programa que leia vários números inteiros positivos do dispositivo de entrada padrão e imprima, para cada inteiro lido, a sua representação em notação binária, usando a função definida no item anterior (o programa que contém a função *main* deve conter também a definição das funções definidas acima).

A execução deve terminar quando um inteiro negativo ou zero for lido.

3. Defina uma função que receba como argumento um número inteiro não-negativo b, em notação binária, e retorne o valor inteiro (de tipo `int`) correspondente, em notação decimal.

   Por exemplo, ao receber o número inteiro `1000`, a função deve retornar o valor `8`.

   Seu programa pode ler b como um valor inteiro — e supor que o valor lido pode ser armazenado como um valor de tipo `int` — ou como uma cadeia de caracteres — e supor que o tamanho máximo da cadeia é de `30` dígitos binários.

   No caso de leitura como um valor de tipo `int`, cada dígito deve ser obtido como resto de divisão por `10`. Por exemplo, para obter cada dígito de `101`, obtenha o `1` mais à direita como resto da divisão de `101` por `10`; depois obtenha o quociente da divisão de `101` por `10` (que é igual a `10`) e repita o processo.

   Escreva um programa que leia, do dispositivo de entrada padrão, várias cadeias de caracteres que representam números inteiros positivos em notação binária, e imprima, para cada valor lido, a sua representação em notação decimal, usando a função definida acima (o programa que contém a função *main* deve conter também a definição da função definida acima).

   A execução deve terminar com o fim dos dados de entrada.

4. Escreva um programa que leia, do dispositivo de entrada padrão, um valor inteiro positivo t, em seguida uma cadeia de caracteres s de tamanho menor que t e, em seguida, várias cadeias de caracteres $s_1, \ldots, s_n$, também com tamanho menor que t, e imprima, para cada cadeia s_i, para i entre 1 e n, uma mensagem que indica se s contém s_i ou não. A entrada deve terminar com o fim dos dados de entrada, isto é, quando fim-de-arquivo for detectado; em entrada interativa, quando *scanf* retornar `-1`, devido ao fato de o usuário digitar *Control-d* (no Unix) ou *Control-z* (no Windows).

   Por exemplo, se a entrada for:

   ```
   1000
   abcdefghijklmnopqrstuvwxyz123456789
   nopqr
   789
   xya
   abc
   ```

   A saída deve ser uma mensagem como a seguir:

   ```
   nopqr Sim
   789 Sim
   xya Nao
   abc Sim
   ```

5. Escreva um programa que leia um inteiro positivo n, em seguida vários pares s_1, s_2 de cadeias de caracteres, de tamanho menor que n, e imprima, para cada par lido, uma cadeia de caracteres que é a *concatenação* de s_1 e s_2 (a concatenação de s_1 com s_2 é a cadeia formada pelos caracteres de s_1 seguidos pelos caracteres de s_2).

   Não se esqueça de considerar que, em C, cadeias de caracteres são armazenadas de modo a terminar com o caractere `'\0'`.

6. Escreva um programa que leia um valor n, duas cadeias de caracteres s_1 e s_2, ambas com tamanho menor que n, e imprima o resultado de remover de s_2 todos os caracteres que aparecem em s_1.

   Por exemplo, para a entrada:

   ```
   100
   abci adefghiabaf
   ```

   A saída deve ser:

   ```
   defghf
   ```

5.6 Arranjo de arranjos

Considere o trecho de programa a seguir:

```
int **a, n=4, m=3;
a = malloc(n * sizeof(int*));
int i;
for (i=0; i<n; i++)
  a[i] = malloc(m * sizeof(int));
```

Após a execução desse trecho de programa, a representa um arranjo com quatro componentes, sendo cada componente um arranjo com três componentes. Tal arranjo é algumas vezes chamado de uma *matriz*, no caso uma matriz 4 por 3. Um arranjo de arranjos é também chamado de *arranjo multidimensional*.

Um arranjo pode ter como componentes arranjos de tamanhos diferentes, alocados dinamicamente, como ilustra o exemplo a seguir.

```
int **a;
a = malloc(2 * sizeof(int*));
...
a[0] = malloc(10 * sizeof(int));
...
a[1] = malloc(40 * sizeof(int));
...
```

O arranjo a é um arranjo, alocado dinamicamente, de tamanho 2, contendo dois arranjos alocados dinamicamente, sendo que o primeiro deles, `a[0]`, tem 10 componentes, enquanto o segundo, `a[1]`, tem 40 componentes.

5.7 Inicialização de Arranjos

Variáveis devem em geral ser inicializadas na sua declaração. Do contrário, o valor armazenado será definido pelo valor que estiver na memória, durante a execução do programa, quando a variável é criada. Isso pode provocar a ocorrência de erros em outras partes do programa, que podem ser difíceis de detectar.

Para variáveis de tipo arranjo, existe uma notação especial em `C`, que infelizmente só pode ser usada em declarações de variáveis, que consiste em enumerar, entre os caracteres `''` e `''`, todos os valores componentes do arranjo, separados por vírgulas.

Por exemplo, pode-se escrever:

```
char* diasDaSemana[] = { "Dom", "Seg", "Ter", "Qua", "Qui", "Sex", "Sab" };
```

para declarar uma variável *diasDaSemana* que é um arranjo de 7 componentes, sendo cada componente uma cadeia de caracteres. O tamanho do arranho não precisa ser especificado, sendo determinado automaticamente de acordo com o número de componentes especificado no valor usado para inicialização do arranjo. Ou seja, a declaração acima é equivalente pode ser feita como a seguir:

```
char* diasDaSemana[7] = { "Dom", "Seg", "Ter", "Qua", "Qui", "Sex", "Sab" };
```

No entanto, se o tamanho for explicitamente especificado, como acima, a variável de tipo arranjo não poderá posteriormente conter um arranjo com tamanho diferente de `7`, ou seja, a variável só poderá ser indexada com valores entre `0` e `6`.

Em casos em que não se pode inicializar um arranjo no instante de sua declaração (pois o valor inicial ou o tamanho de um arranjo ainda não são conhecidos), é em geral conveniente inicializar a variável de tipo arranjo com o valor `NULL` (ponteiro nulo).

5.8 Exercícios Resolvidos

1. Escreva um programa que leia notas de alunos obtidas em tarefas de uma certa disciplina, e imprima a nota total de cada aluno e a média das notas dos alunos nessa disciplina. A entrada dos dados contém, primeiramente, o número n de alunos da turma, depois o número k de tarefas da disciplina, e em seguida o número de cada aluno (de 1 a n) e as k notas desse aluno. Cada valor é separado do seguinte por um ou mais espaços ou linhas.

Solução: A solução define uma função *calcNotas* que recebe um arranjo com as notas de cada aluno em cada tarefa, e retorna um arranjo com as notas finais de cada aluno. Em `C`, o arranjo passado como argumento, alocado dinamicamente, é um ponteiro para ponteiros-para-inteiros.

Outra função, chamada *media*, recebe um arranjo de inteiros (em `C`, um ponteiro para inteiros) e o tamanho do arranjo, e retorna um valor de tipo `float` que é igual à média dos inteiros contidos no arranjo.

Na função *main*, um arranjo de notas de cada aluno em cada avaliação é preenchido com valores lidos, o método *calcNotas* é chamado para cálculo das notas finais, e as notas finais calculadas, assim como a média, calculada por chamada à função *media*, são impressas.

As notas de cada aluno são representadas em `C` como ponteiros para ponteiros-para-inteiros.

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

int* calcNotas(int **notas, int numAlunos, int numAvaliacoes) {
  int i, j, *notasFinais;
  notasFinais = malloc(numAlunos * sizeof(int));

  // Inicializa notas finais de cada aluno com 0 for (i=0; i < numAlunos; i++)
    notasFinais[i] = 0;

  for (i=0; i < numAlunos; i++)
    for (j=0; j < numAvaliacoes; j++)
      notasFinais[i] += notas[i][j];

  return notasFinais;
}

float media(int vs[], int n) {
  int i, soma=0;
  for (i=0; i<n; i++) soma += vs[i];
  return ((float)soma)/n;
}

int main() {
  int n,k;
  scanf("%d",&n);
  scanf("%d",&k);

  int **notas = malloc(n*sizeof(int*));
  int i,j;

  for (i=0; i<n; i++) {
    notas[i] = malloc(k*sizeof(int));
    for (j=0; j<k; j++)
      scanf("%d",&(notas[i][j]));
    }

    int *notasFinais = calcNotas(notas,n,k);

    for (i=0; i<n; i++)
      printf("Nota final do aluno %d = %d\n", i+1, notasFinais[i]);
    printf("Media da turma = %f", media(notasFinais,n));
  }
```

5.9 Exercícios

1. Considere que uma empresa comercial, que tem n lojas especializadas de certo tipo de material, te contratou para fazer o seguinte programa em `C`.

A empresa tem dados armazenados sobre o número de vendas realizadas em cada loja. Não importa qual tipo de material, a empresa está interessada apenas no número de unidades vendidas.

A empresa quer um programa que leia, do dispositivo de entrada padrão, o valor de n, em seguida n valores $v_1, \ldots, v_n$ que correspondem ao número de unidades vendidas em um mês nas lojas de 1 a n, respectivamente, e imprima, no dispositivo de saída padrão, quais foram as lojas de 1 a n nas quais o número de unidades vendidas foi maior ou igual à média de unidades vendidas em suas lojas.

2. Escreva um programa que leia um texto qualquer, caractere a caractere, e imprima:

- o número de algarismos que ocorrem no texto,
- o número de letras que ocorrem no texto, e
- o número de linhas do texto que contêm pelo menos um caractere.

3. Escreva um programa que leia um valor inteiro positivo n, em seguida uma matriz quadrada $n \times n$ de valores inteiros, e imprima uma mensagem indicando se a matriz quadrada é ou não um *quadrado mágico*.

Um *quadrado mágico* é uma matriz quadrada na qual a soma dos números em cada linha, coluna e diagonal é igual.

4. Os votos de uma eleição são representados de modo que `0` significa voto em branco, `1` a n significam votos para os candidatos de números `1` a n, respectivamente, e qualquer valor diferente desses significa voto nulo.

A eleição tem um vencedor se o número de votos em branco mais o número de votos nulos é menor do que 50% do total de votos, sendo vencedores, nesse caso, todos os candidatos com número de votos igual ao maior número de votos.

Escreva um programa que leia os votos de uma eleição e determine se há um vencedor e, em caso positivo, determine o número de vencedores da eleição, quais são esses vencedores e o número de votos dos mesmos.

5. Escreva um programa que leia um valor inteiro positivo n, em seguida um valor inteiro positivo k, depois uma sequência s de k valores inteiros diferentes de zero, cada valor lido separado do seguinte por um ou mais espaços ou linhas, e imprima n linhas tais que: a $1^{\underline{a}}$ linha contém os valores de s divisíveis por n, a $2^{\underline{a}}$ linha contém os valores de s para os quais o resto da divisão por n é 1, etc., até a n-ésima linha, que contém os valores de s para os quais o resto da divisão por n é $n - 1$.

Dica: use um arranjo de n posições com elementos que são arranjos de k valores inteiros, sendo um valor igual a zero indicativo de ausência de valor naquela posição.

Exemplo: Considere a entrada:

```
4 12 13 15 16 17
```

Para essa entrada, a saída deve ser como a seguir:

```
Valores divisiveis por 4:  12 16
Valores com resto da divisao por 4 igual a 1:  13 17
Valores com resto da divisao por 4 igual a 2:
Valores com resto da divisao por 4 igual a 3:  15
```

6. Reescreva o programa referente a impressão do Triângulo de Pascall (exercício 13 do capítulo anterior, página 76), usando um arranjo e o fato de que cada valor em uma posição j (diferente da primeira, que é 1) de qualquer linha (diferente da primeira) do Triângulo de Pascal pode ser obtido somando-se os valores nas posições j e $j - 1$ da linha anterior.

Por exemplo, o valor contido na terceira coluna da linha correspondente a $n = 5$ no Triângulo de Pascal (página 76) é 10. Ele é igual a 6+4: 6 é o valor na mesma coluna da linha anterior (correspondente a $n = 4$), e 4 o valor anterior a 6 nesta mesma linha (correspondente a $n = 4$).

7. Escreva um programa que leia, repetidamente, do dispositivo de entrada padrão, os seguintes valores, nesta ordem:

(a) um número inteiro positivo n,

(b) n inteiros positivos $v_1, \ldots, v_n$,

(c) 3 inteiros positivos i, j, k.

O programa deve imprimir, para cada n lido, uma linha com os valores $v_i, v_{i+k}, v_{i+2\times k}, \ldots, v_{i+p\times k}$ tais que $i + p \times k \leq j$. Os valores devem ser separados por espaços e impressos no dispositivo de saída padrão.

Por exemplo, para a entrada:

```
10 11 25 33 40 50 69 73 85 96 101 3 2 7 0
```

a saída deve ser:

```
33 50 73
```

Isso ocorre porque o primeiro valor impresso é v_3 $(i = 3)$, o seguinte é v_5 $(j = 2,\ 5 = i + j)$, e o seguinte e último é v_7 $(k = 7 = i + 2 \times j)$.

8. Escreva um programa que leia, repetidamente, um valor inteiro n positivo ou nulo, em seguida, se o valor for positivo, uma sequência de caracteres de tamanho máximo n em uma linha da entrada padrão, e imprima, para cada linha lida, uma linha com os mesmos caracteres mas com as letras minúsculas transformadas em letras maiúsculas. O programa deve terminar quando o valor n lido for nulo (igual a zero).

Escreva o seu programa definindo e usando:

- uma função *minusc* que recebe um valor de tipo `char` e retorna se esse valor é ou não uma letra minúscula, ou seja, se é ou não um valor maior ou igual a `'a'` e menor ou igual a `'z'`).
- uma função *capitaliza* que recebe um valor de tipo `char` e, se o valor for uma letra minúscula, retorna a letra maiúscula correspondente, senão o próprio valor recebido.

  A função *capitaliza* é definida para você abaixo:

```
char capitaliza (char c) {
  return (minusc(c) ?  c - 'a' + 'A' : c);
}
```

9. Escreva um programa que leia, do dispositivo de entrada padrão, um texto qualquer, caractere a caractere, e imprima, no dispositivo de saída padrão, i) o número de palavras que têm um gênero, masculino ou feminino, ii) o número de palavras do gênero masculino, e iii) o número de palavras do gênero feminino.

Você pode considerar, para simplificar o problema, que:

- Uma palavra é uma sequência de caracteres quaisquer seguida de um delimitador. Um delimitador é um dos caracteres `' '`, `'\t'`, `'\n'` ou `EOF`.

  A entrada termina com `EOF` (caractere indicador de fim dos dados de entrada: em entrada interativa, `Control-z` seguido de `Enter` no Windows, ou `Control-d` no Linux).
- Uma palavra têm gênero masculino se seu último caractere for a letra `'o'` ou se a penúltima letra for `'o'` e a última letra for `'s'`, ou se a palavra é igual a `'O'` ou `"Os"`. Analogamente, uma palavra tem gênero feminino se seu último caractere for a letra `'a'` ou se a penúltima letra for `'a'` e a última letra for `'s'`, ou se a palavra é igual a `'A'` ou `"As"`.

Dica: Use a função *getchar* para ler um caractere, e use o valor retornado por *getChar* para verificar fim dos dados de entrada: *getChar* retorna o valor *EOF* (igual a `-1`) para indicar fim dos dados.

10. Escreva um programa que leia, caractere a caractere, do dispositivo de entrada padrão, um texto qualquer, e imprima, no dispositivo de saída padrão, o número de palavras do texto.

Dica: para contar o número de palavras, use uma variável booleana para indicar se a posição corrente de leitura corresponde a uma posição interna a uma palavra ou externa. Uma posição fora de uma palavra é uma posição correspondente a um delimitador.

Lembre-se que, em `C`, uma variável booleana é uma variável inteira: o valor `0` representa falso e qualquer valor diferente de zero verdadeiro

11. Estenda o exercício 9 para imprimir também o número total de palavras do texto.

12. Um armazém trabalha com n mercadorias diferentes, identificadas por números de 1 a n.

Cada funcionário do armazém tem salário mensal estipulado como 20% do total da receita que ele consegue vender, mais um bônus igual a 10% da receita obtida com a mercadoria que teve a maior receita com as vendas.

Escreva um programa em `C` para calcular o valor do salário de um funcionário em um determinado mês; o programa deve ler o valor n que indica o número de mercadorias, em seguida uma sequência de pares i seguido de v_i, sendo i um número de mercadoria (de 1 a n) e v_i uma receita obtida no mês pelo funcionário por uma operação de venda da mercadoria i, e imprimir o salário desse funcionário, nesse mês.

Por exemplo, se a entrada for:

```
3
1 10.0
2 20.0
3 30.0
1 30.0
```

a saída deve ser: `22.0`

Isso ocorre porque o total das vendas foi `90.0` e a maior receita por mercadoria, obtida com a mercadoria 1, foi `40.0` (`30.0 + 10.0`), e `90.0 * 0.2` (vinte por cento do total de vendas) somado com `0.1 * 40.0` (dez por cento da total de vendas com a mercadoria mais vendida) é igual a `22.0`.

Capítulo 6

Ponteiros

Ponteiros são representações de endereços na memória do computador. Eles constituem um recurso de programação de baixo nível, que espelha a representação de estruturas de dados em memórias de computadores. Seu uso é devido em grande parte a eficiência (ou seja, tem o objetivo de fazer com que programas sejam executadas rapidamente ou usando poucos recursos de memória) ou, algumas vezes, a necessidade de acesso ao hardware. O uso de ponteiros poderia ser em grande parte das vezes ser substituído por uso de um estilo de programação baseado em abstrações mais próximas do domínio do problema e não da implementação de uma solução do problema em um computador, abstrações essas tanto de dados quanto de definição de funções sobre esses dados.

O uso de ponteiros deve ser feito com cuidado, para evitar erros em tempo de execução que podem ser difíceis de entender e de corrigir.

Uma variável é um lugar da memória ao qual foi dado um nome (o nome da variável). Toda variável tem um endereço, que é o endereço do lugar da memória que foi alocado para a variável. Um ponteiro é um endereço (da memória), ou um tipo (de valores que são ponteiros para variáveis de um determinado tipo; por exemplo, o tipo `int *` é um tipo ponteiro, de valores que são ponteiros para variáveis de tipo `int`; dizemos apenas: tipo "ponteiro para `int`"). Além disso, quando o contexto deixar claro, usamos também ponteiro para denotar variável que contém um endereço.

Um tipo ponteiro é indicado pelo caractere `*` como sufixo de um tipo, indicando o tipo ponteiro para áreas de memória deste tipo. Por exemplo:

```
int *p;
```

declara uma variável de nome p e tipo `int *`, ou seja, ponteiro para `int`.

O caractere `*` é considerado em um comando de declaração de variáveis em `C` como qualificador da variável que o segue, de modo que, por exemplo:

```
int *p, q;
```

declara um ponteiro p e uma variável q de tipo `int`, e não dois ponteiros.

O uso de ponteiros é baseado principalmente no uso dos operadores `&` e `*`, e da função *malloc*.

O operador `&`, aplicado a uma variável, retorna o endereço dessa variável. Por exemplo, a sequência de comandos:

```
int *p, q;
p = &q;
```

declara p como uma variável de tipo `int *`, q como uma variável de tipo `int` e armazena o endereço de q em p.

O operador `*` é um operador chamado de "derreferenciação": aplicado a um ponteiro p, o resultado é a variável apontada por p (se usado como um valor, o resultado é o valor contido na variável apontada por p).

O uso de `*` em uma declaração significa declaração de um ponteiro; o uso de `*` precedendo um ponteiro em uma expressão significa "derreferenciação".

Considere, por exemplo, o problema de trocar o conteúdo dos valores contidos em duas variáveis a e b de tipo `int`. Para fazer isso precisamos passar para uma função *troca* o endereço das variáveis, i.e. devemos chamar *troca*`(&a,&b)`, onde a função *troca* é definida como a seguir:

```
void troca(int* x, int* y) {
  int t = *x;
  *x = *y;
  *y = t;
}
```

Note que uma chamada *troca*`(`a`,`b`)` em vez de *troca*`(&`a`,&`b`)` fará com que a execução do programa use endereços de memória que são valores inteiros contidos em a e b, e isso fará com que o programa provavelmente termine com um erro devido a tentativa de acesso a endereço inválido de memória.

Note também que a função *scanf* modifica o valor de uma variável, e é por isso que o endereço da variávell é que deve ser passado como argumento de *scanf*.

Ocorre um erro durante a execução de um programa quando um ponteiro é “derreferenciado” e o ponteiro representa um endereço que não está no conjunto de endereços válidos que o programa pode usar. Esse erro é comumente chamado em inglês de *segmentation fault*, ou seja, erro de segmentação. Isso significa que foi usado um endereço que está fora do *segmento* (trecho da memória) associado ao processo que está em execução.

O endereço `0` é também denotado por *NULL* (definido em *stdlib*) e é usado para indicar um “ponteiro nulo”, que não é endereço de nenhuma variável. É em geral também usado em inicializações de variáveis de tipo ponteiro para as quais não se sabe o valor no instante da declaração.

6.1 Operações de soma e subtração de valores inteiros a ponteiros

Em `C`, é possível incrementar (somar um) a um ponteiro. Sendo p um ponteiro para valores de um tipo t qualquer, p`+1` representa o “endereço seguinte ao endereço denotado por p e, analogamente, p`-1` representa o “endereço anterior ao endereço denotado por p.

Com isso, é possível realizar operações aritméticas quaisquer com um ponteiro e um inteiro. Essas operações aritméticas são realizadas de modo a levar em conta o tamanho do tipo do valor denotado por um ponteiro: somar o inteiro 1 a um ponteiro significa somar uma unidade ao ponteiro igual ao tamanho do tipo de variáveis apontadas por esse ponteiro.

Por exemplo, um valor inteiro ocupa, em muitas implementações, 4 bytes (32 bits). Nessas implementações, se p é do tipo `*int` (isto é, é um ponteiro para variáveis de tipo `int`), p`+1` representa um endereço 4 bytes maior do que o endereço denotado por p.

6.2 Ponteiros e arranjos

Em `C`, o nome de uma variável de tipo arranjo pode ser usado como um ponteiro para a primeiro posição do arranjo, com a diferença que o valor dessa variável não pode ser modificado (a variável de tipo arranjo corresponde uma variável de tipo ponteiro declarada com o atributo `const`).

Por exemplo, quando se declara o arranjo:

```
int a[10];
```

um arranjo de 10 posições é alocado e o nome a representa um ponteiro para a primeira posição desse arranjo. Ou seja, o nome a representa o mesmo que `&`a`[0]`.

Sendo i uma expressão qualquer de tipo `int`, a expressão a`[`i`]` denota o mesmo que `(`a`+`i`)` — ou seja, a i-ésima posição do arranjo a, i posições depois da 0-ésima posição. Se usada em um contexto que requer um valor, essa expressão é “derreferenciada”, fornecendo o valor `*(`a`+`i`)`. Portanto, em

`C` a operação de indexação de arranjos é expressa em termos de uma soma de um inteiro a um ponteiro.

Assim, quando um arranjo a é passado como argumento de uma função f, apenas o endereço `&`a`[0]` é passado.

Capítulo 7

Registros

Um registro (ou, como é chamado em C, uma *estrutura*) é um tipo, e também um valor desse tipo, que é um produto cartesiano de outros tipos, chamados de campos ou componentes do registro, com notações especiais para definição dos componentes do produto e para acesso a esses componentes.

Em matemática, e em algumas linguagens de programação, a forma mais simples de combinar valores para formação de novos valores é a construção de pares. Por exemplo, `(10,'*')` é um par, formado por um primeiro componente, `10`, e um segundo componente, `'*'`. Um par é um elemento do *produto cartesiano* de dois conjuntos — o par `(10,'*')` é um elemento do produto cartesiano do conjunto dos valores inteiros pelo conjunto dos valores de tipo `char`.

Naturalmente, além de pares, é também possível formar triplas, quádruplas, quíntuplas etc. — usualmente chamadas de *tuplas* — que são elementos de produtos cartesianos generalizados, ou seja, elementos de um produto de vários conjuntos.

Em linguagens de programação (como C por exemplo), no entanto, é mais comum o uso de *registros*, em vez de tuplas. Um *registro* é uma representação de um valor de um produto cartesiano, assim como uma tupla, mas cada componente, em vez de ser identificado pela sua posição (como no caso de tuplas), é identificado por um nome — usualmente chamado de *rótulo*. Cada componente tem um nome a ele associado.

Por exemplo, um valor como:

$$\{ x \texttt{ = 10, } y \texttt{ = '*'}\}$$

representa, em C, um registro com dois componentes, em que um deles tem rótulo x e o outro tem rótulo y. Nesse exemplo, o valor do componente é separado do rótulo pelo símbolo "=". O registro $\{y$ `= '*',` x `= 10`$\}$ representa o mesmo valor que o registro $\{x$ `= 10,` y `= '*'`$\}$. Ou seja, a ordem em que os componentes de um registro é escrita não é relevante para determinação do valor representado, ao contrário do que ocorre com relação à ordem dos componentes de uma tupla. Deve existir, é claro, uma operação para selecionar um componente de um registro, assim como ocorre no caso de tuplas.

No entanto, em C a especificação de valores de tipo registro deve seguir uma ordem para os valores dos campos, que é a ordem em que os campos aparecem na definição do tipo registro, e só podem ser usados em declarações de variáveis, como veremos no exemplo a seguir.

Uma declaração de um tipo registro consiste de uma sequência de campos, cada um dos quais com um tipo e um nome. Por exemplo:

```
struct contaBancaria {
  int numero;
  char *idCorrentista;
  float saldo;
};
```

Em C, a definição acima consiste na definição de um tipo, de nome *account*, ao qual se pode referir usando a palavra `struct` seguida do nome *account*.

Por exemplo, a seguinte declaração cria uma variável desse tipo:

```
struct contaBancaria conta;
```

O tipo *contaBancaria*, assim como a variável *conta*, têm três campos ou componentes: *contaBancaria*, *numeroDaConta* e *saldo*.

O acesso aos componentes é feito usando-se um valor de tipo registro seguido de um ponto e do nome do campo. Por exemplo, *conta.numero* tem tipo `int`. Analogamente, *conta.idCorrentista* tem tipo `char*` e *conta.saldo* tem tipo `float`.

Esses componentes denotam (podem ser usados ou modificados como) uma variável comum do tipo do campo.

Valores de tipo registro podem ser construídos em `C` mas apenas na inicialização de uma variável de tipo registro, de modo semelhante ao que ocorre no caso de arranjos. Um valor de tipo registro especifica valores a cada um dos campos do registro, entre chaves, como mostrado no exemplo a seguir.

O exemplo seguinte ilustra uma declaração de uma variável do tipo *contaBancaria*, definido acima, especificando um valor inicial para a variável:

```
struct contaBancaria conta = { 1, "MG1234567", 100.0 };
```

A atribuição de um valor de tipo registro a outro copia, como esperado, o valor de todos os campos do registro. Considere o seguinte exemplo:

```
#include <stdio.h>

struct Ponto { int x; int y; } ;

int main() {
  struct Ponto p = {1,2}, q;
  q = p;
  q.x = 2;
  printf("p.x = %d\nq.x = %d\n", p.x, q.x);
}
```

Esse programa imprime:

```
p.x = 1
q.x = 2
```

7.0.1 Declarações de tipos com `typedef`

O uso de nomes para introdução de novos tipos é bastante útil, para documentação e legibilidade do programa, e isso ocorre particularmente no caso de tipos registro e outros tipos Por exemplo, para dar um nome para um tipo que representa coordenados do plano cartesiano, ou dados de uma conta bancária, pode-se definir e usar tipos *Ponto* e *ContaBancaria* como a seguir:

```
struct Ponto { int x; int y; };
typedef struct Ponto Ponto;

struct contaBancaria {
  int numero;
  char *idCorrentista;
  float saldo;
};
typedef struct contaBancaria contaBancaria;

int main() {
  Ponto p, q;
  ContaBancaria c; // aqui vem uso de variaveis p,q,c ...
}
```

7.0.2 Ponteiros para registros

Ponteiros para registros podem ser usadas para passar valores de tipo registro sem ter que copiar o registro, e também de modo a permitir a alteração de campos de registros.

Um ponteiro para um registro pode ser derreferenciado como normalmente, usando o operador `*`, mas existe em `C` a possibilidade de usar o operador `->`, que além da derreferenciação faz também acesso a um campo de um registro. Por exemplo:

```
struct Ponto { int x; int y; };
typedef struct Ponto Ponto;

void moveParaOrigem (Ponto *p) {
  p -> x = 0; // O mesmo que:  (*p).x = 0;
  p -> y = 0;
}
```

7.0.3 Estruturas de dados encadeadas

Em computação, uma estrutura de dados encadeada consiste de uma sequência de registros de tipo T que contêm um campo que é uma ponteiro que pode ser nulo ou um ponteiro para um próximo registro de tipo T.

Por exemplo, uma lista encadeada de registros com campos de tipo `int` pode ser formada com registros do seguinte tipo:

```
struct ListaInt {
  int val;
  struct ListaInt *prox;
};
```

Listas encadeadas são estruturas de dados flexíveis, pois não requerem tamanho máximo, como arranjos. Elas podem crescer e decrescer de tamanho à medida que dados vão sendo inseridos e removidos.

A desvantagem, em relação ao uso de arranjos, é que o acesso a um componente da estrutura de dados requer um tempo que depende da posição desse componente na estrutura: o acesso a cada componente depende de acesso a cada um dos componentes anteriores a ele na lista.

Árvores binárias podem ser formadas de modo similar. Por exemplo, uma árvore binária com nodos que contêm campos de tipo `int` pode ser formada com registros do seguinte tipo:

```
struct ArvBinInt {
  int val;
  struct ArvBinInt *esq;
  struct ArvBinInt *dir;
};
```

O seguinte exemplo ilustra o uso de uma lista encadeada para evitar a restrição de se ter que especificar um número máximo de valores, necessário para uso de arranjo. Considere o problema do Exercício Resolvido 1, da seção 5.5.4, que propõe que um texto qualquer seja lido e seja impressa a frequência de todos os caracteres que ocorrem no texto. Considere que o problema não especifica o tamanho do texto. A solução a seguir usa uma lista encadeada de caracteres para armazenar o texto, em vez de uma cadeia de caracteres.

Em casos como esse, pode ser mais adequado usar um arranjo flexível, com um tamanho máximo que pode ser aumentado, testando, antes de cada inserção de um novo valor no arranjo, se esse tamanho máximo foi atingido. Se o tamanho máximo for atingido, um novo arranjo é alocado com um tamanho maior (por exemplo, o dobro do tamanho anterior), o arranjo antigo é copiado para o novo, e o novo arranjo passa a ser usado, no lugar do antigo. Arranjos flexíveis são estruturas de dados bastante usadas em programas escritos em linguagens como, por exemplo, `Java` e `C++`.

7.0.4 Exercícios Resolvidos

1. Escreva um programa que funcione como o exemplo fornecido na seção 5.3 mas sem a condição de que o número de valores a serem impressos, em ordem impressa, seja fornecido. Ou seja, escreva um programa que leia do dispositivo de entrada padrão qualquer número de valores inteiros, separados por um ou mais espaços ou linhas, e imprima esses valores na ordem inversa em que foram lidos.

   Solução:

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

struct nodoLista { int val; struct nodoLista *prev; };
typedef struct nodoLista nodoLista;

int main() {
  int val, numValLidos;
  struct nodoLista *cur, *prev = NULL;
  while (1) {
    numValLidos = scanf("%d",&val);
    if (numValLidos != 1) break;
    cur = malloc(sizeof(nodoLista));
    cur -> val = val;
    cur -> prev = prev;
    prev = cur;
  }
  while (cur != NULL) {
    printf("%d ", cur -> val);
    cur = cur -> prev;
  }
}
```

A solução usa a notação *ponteiro* `->` *campo*, que é uma abreviação para `(*`*ponteiro*`).`*campo*.

2. Escreva um programa que leia, do dispositivo de entrada padrão, resultados de partidas de um campeonato e imprima, no dispositivo de saída padrão, a lista dos nomes dos times que obtiveram maior número de pontos nesse campeonato. Cada vitória vale 3 pontos e cada empate vale 1 ponto.

   A entrada consiste dos seguintes dados, nesta ordem:

   (a) uma linha contendo um número inteiro n, que especifica o número de times do campeonato;

   (b) n linhas contendo dois valores i e s_i, onde i é um número inteiro entre 1 e n e s_i é o nome do time i; o nome de um time é uma cadeia de caracteres de tamanho máximo 30; a ordem em que essas n linhas aparecem na entrada deve ser irrelevante;

   (c) várias linhas contendo 4 números inteiros não-negativos t_1 v_1 t_2 v_2, que indicam o resultado da partida entre o time t_1 e t_2: t_1 marcou v_1 gols e t_2 marcou v_2 gols; os resultados terminam com o fim da entrada (EOF).

   Os times na lista impressa devem estar separados por vírgula (se houver mais de um time com mais pontos), e a ordem é irrelevante. Por exemplo, se a entrada for:

```
3
1 America
2 Atletico
3 Cruzeiro
1 1 2 2
1 2 3 3
2 1 3 1
```

   A saída deve ser: `Atletico, Cruzeiro`

   Isso porque o `America` perdeu do `Atletico` (1x2) e do `Cruzeiro` (2x3), e o `Atletico` empatou com o `Cruzeiro` (1x1).

   Solução: A solução apresentada usa arranjos para armazenar nomes de times e para armazenar pontos acumulados em partidas. Esses arranjos são indexados com o número do time menos 1 (porque os números de time variam entre 1 e n e arranjos em `C` sempre têm índice inicial igual a `0`). O programa constrói uma lista de vencedores (times com maior número de pontos) à medida em que tal maior número de pontos é calculado. A lista de vencedores — chamda de *maiores* — é inicialmente nula. Para cada time, do primeiro ao último, se o número de pontos é maior do que o maior calculado até cada instante desta iteracão, o maior é atualizado, senão, se o número de pontos for igual, este é inserido na lista de maiores.

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

struct Maiores { int num; struct Maiores* prox; };
typedef struct Maiores Maiores;

int main() {
  int n, num;
  scanf("%d", &n);

  char** times = malloc(n * sizeof(char*));
  int i; const int tamMaxNomeTime = 31; // 31 devido a terminacao com '\0'
  for (i=0; i<n; i++) {
    scanf("%d", &num);
    times[num-1] = malloc(tamMaxNomeTime*sizeof(char));
    scanf("%s", times[num-1]);
  }

  int numValLidos, num1, num2, gols1, gols2, *pontos;
  pontos = malloc(n * sizeof(int));
  for (i=0; i<n; i++) pontos[i] = 0;
  while (1) {
    numValLidos = scanf("%d%d%d%d",&num1, &gols1, &num2, &gols2);
    if (numValLidos != 4) break;
    if (gols1>gols2) pontos[num1-1] += 3; else
    if (gols2>gols1) pontos[num2-1] += 3;
    else { pontos[num1-1]++; pontos[num2-1]++; }
  }

  Maiores* maiores = NULL;
  int maior = 0;
  for (i=0; i<n; i++)
    if (pontos[i] > maior) {
      maiores = malloc(sizeof(Maiores));
      maiores->num = i;
      maiores->prox = NULL;
      maior = pontos[i];
    }
    else if (pontos[i] == maior) { // novo elemento em maiores
      Maiores* novo = malloc(sizeof(Maiores));
      novo->prox = maiores;
      novo->num = i;
      maiores = novo;
    }
  printf("%s", times[maiores->num]);
  maiores = maiores->prox;
  while (maiores!=NULL) {
    printf(", %s",times[maiores->num]);
    maiores = maiores -> prox;
  }
  printf("\n");
  return 0;
}
```

3. Escreva um programa que leia uma sequência de valores inteiros quaisquer e imprima esses

valores em ordem não-decrescente (cada valor seguinte é maior ou igual ao anterior). Valores iguais devem aparecer tantas vezes quantas existirem na entrada.

Por exemplo, considere a entrada:

```
4 3 1 5 5 2 3
```

A saída deve ser:

```
1 2 3 3 4 5 5
```

Solução: Vamos definir e usar uma função para ordenação de valores conhecida como *ordenação por seleção*. O algoritmo simplesmente seleciona a cada iteração o maior elemento (a menos de igualdade) e insere o valor selecionado no início de uma lista l de valores ordenados; no final a lista l estará ordenada (no final o menor elemento será inserido no início da lista).

7.0.5 Exercícios

1. Escreva um programa que leia um valor inteiro positivo n, em seguida leia, caractere a caractere, uma cadeia de caracteres s de um tamanho qualquer, maior do que n, e imprima os n últimos caracteres de s, armazenando para isso a cadeia s como uma uma lista encadeada de caracteres onde um apontador é usado para apontar para o caractere anterior da cadeia.

2. Escreva um programa que leia um valor inteiro positivo n e, em seguida, uma sequência s de valores inteiros diferentes de zero, cada valor lido separado do seguinte por um ou mais espaços ou linhas, e imprima n linhas tais que: a 1ª linha contém os valores de s divisíveis por n, a 2ª linha contém os valores de s para os quais o resto da divisão por n é 1, etc., até a n-ésima linha, que contém os valores de s para os quais o resto da divisão por n é $n - 1$. A entrada termina quando um valor igual a zero for lido. A ordem dos valores impressos em cada linha não é relevante.

   Use um arranjo de n posições com elementos que são registros representando listas encadeadas de valores inteiros.

   Exemplo: Considere a entrada:

   ```
   4 12 13 15 16 17
   ```

   Para essa entrada, a saída deve ser como a seguir:

   ```
   Valores divisiveis por 4:  16 12
   Valores com resto da divisao por 4 igual a 1:  17 13
   Nenhum valor com resto da divisao por 4 igual a 2
   Valores com resto da divisao por 4 igual a 3:  15
   ```

3. Reescreva o programa do exercício anterior de modo a fazer com que a ordem dos valores impressos seja a mesma ordem que os valores ocorrem na entrada. Para isso, use um arranjo de ponteiros para a última posição de cada lista encadeada.

7.1 Notas Bibliográficas

Existe uma vasta literatura sobre algoritmos e estruturas de dados em computação, que estendem o que foi abordado neste capítulo principalmente com o estudo mais detalhado de algoritmos para busca, inserção, remoção e ordenação de valores nessas estruturas de dados. Além de aspectos de implementação de estruturas de dados e de operações para manipulação das mesmas, essa literatura aborda em geral diferentes aplicações dos algoritmos e discute também aspectos de eficiência (ou *complexidade*, como é usual dizer em computação) de algoritmos.

Livros didáticos dedicados a esses temas incluem [26, 21, 18], os dois primeiros já traduzidos para a língua portuguesa e o terceiro escrito em português. [9] é um livro interessante, que adota a abordagem de programação funcional.

Capítulo 8

Exercícios

Este capítulo descreve a solução de diversos exercícios, que mostram como usar e decidir quando usar os diversos comandos e estruturas de dados abordados neste livro.

Os enunciados dos exercícios são obtidos da página Web `http://br.spoj.pl/problems/`. *SPOJ* (*Sphere Online Judge*) é um sistema disponível na Internet (na página `http://br.spoj.pl/`) que permite o registro de novos problemas e a submissão de soluções de problemas registrados. Há dezenas de milhares de usuários e milhares de problemas já registrados. A solução pode ser submetida em dezenas de linguagens de programação, incluindo, é claro, `C`.

8.1 ENCOTEL

Considere que uma representação alfanumérica de um número de telefone é uma sequência de caracteres tal que cada caractere pode ser: uma letra maiúscula (de `A` a `Z`), um hifen (`-`) ou um dígito `1` ou tt 0, sendo que letras maiúsculas representam dígitos de `2` a `9`, de acordo com a tabela abaixo.

Letras	Número
ABC	2
DEF	3
GHI	4
JKL	5
MNO	6
PQRS	7
TUV	8
WXYZ	9

Escreva um programa que leia várias linhas, cada linha contendo uma tal representacão alfanumérica de número de telefone, e imprima uma sequência de representações para os números de telefone, novamente uma em cada linha, que substitua letras maiúsculas por dígitos de acordo com a tabela mostrada.

Considere que cada representacão alfanumérica possui entre 1 e 30 caracteres. A entrada é terminada por fim de arquivo (EOF).

Por exemplo, para a entrada:

```
1-HOME-SWEET-HOME
MY-MISERABLE-JOB
```

A saída deve ser:

```
1-4663-79338-4663
69-647372253-562
```

A solução mostrada abaixo usa um arranjo que armazenada, para cada letra maiúscula, seu código, segundo a tabela apresentada. De fato, como em `C` o primeiro índice de um arranjo tem que ser `0`, para cada letra maiúscula corresponde um índice entre `0` e o número de letras maiúsculas menos um.

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

const int n = 26; // Numero de letras maiusculas

char* mkCodLetras() {
  char i='A', codigo='2', *cod=malloc(n*sizeof(char));
  int j, k; while (i<='W') {
    k = (i == 'P' || i == 'W')? 4 :  3;
    for (j=0; j<k; j++) cod[i-'A'+j] = codigo;
    codigo++; i+=k;
  }
  return cod;
}

int main() {
  const int max = 31; // 31 devido a terminacao com '\0'
  char *codLetras = mkCodLetras(), *exp = malloc(max*sizeof(char));
  int numValLidos;
  while (1) {
    numValLidos = scanf("%s", exp);
    if (numValLidos != 1) break; int i = 0, c_A; char c;
    while (exp[i] != '\0') {
      c = exp[i]; c_A = c-'A';
      printf("%c",c_A>=0 && c_A<n?  codLetras[c_A] : c);
      i++;
    }
    printf("\n");
  }
}
```

Essa solução evita escrever um programa, relativamente ineficiente e mais longo, que testa, após a leitura de cada caractere, se o caractere é uma letra pertencente a um grupo específico de letras na tabela mostrada, para impressão do dígito correspondente a esse grupo de letras na tabela.

8.2 PAPRIMAS

x Um número primo é um número que possui somente dois divisores: ele mesmo e o número 1. Exemplos de números primos são: 1, 2, 3, 5, 17, 101 e 10007.

Neste problema você deve ler um conjunto de palavras, onde cada palavra é composta somente por letras no intervalo a-z e A-Z. Cada letra possui um valor específico, a letra a vale 1, a letra b vale 2 e assim por diante, até a letra z, que vale 26. Do mesmo modo, a letra A vale 27, a letra B vale 28 e a letra Z vale 52.

Você deve escrever um programa para determinar se uma palavra é uma palavra prima ou não. Uma palavra é uma palavra prima se a soma de suas letras é um número primo.

Entrada: A entrada consiste de um conjunto de palavras. Cada palavra está sozinha em uma linha e possui L letras, onde $1 \leq L \leq 20$. A entrada é terminada por fim de arquivo (EOF).

Saída: Para cada palavra você imprimir: *It is a prime word.*, se a soma das letras da palavra é um número primo, caso contrário você deve imprimir *It is not a prime word.*.

Exemplo: Para a entrada:

UFRN
contest
AcM

a saída dever ser:

It is a prime word.
It is not a prime word.
It is not a prime word.

Uma solução completa é mostrada na Figura 8.1. A função *main* lê diversas palavras, até encontrar fim-de-arquvo, e imprime mensagem indicando se cada palavra lida é uma palavra prima ou não. O tamanho de cada palavra é restrito a um tamanho máximo de 20 caracteres.

A função *primo* verifica se um dado número n é primo ou não. Essa função pode ser implementada de diversos modos. A Figura 8.1 usa o método simples de divisões sucessivas, por todos os números menores que $\sqrt{n}$. Para valores grandes de n, esse método gasta mais tempo do que outros algoritmos existentes. Os programas mostrados a seguir usam o algoritmo conhecido como *crivo de Eratóstenes*. O leitor interessado pode consultar e.g.:

`http://en.wikipedia.org/wiki/Prime_number#Verifying_primality`

```
const int max = 21; // Um a mais do que o tamanho máximo de uma palavra,
                    // pois um caractere (o caractere '\0') é usado
                    // para indicar final da cadeia de caracteres
int prima(char* palavra);
int ler(char* palavra);

int main () {
  char palavra[max];
  while (ler(palavra))
    printf("It is%s a prime word.\n", prima(palavra) ?  "" : " not");
}
int minusc (char let) { return (let >= 'a' && let <= 'z'); }
int valor (char let) {
  return (minusc(let) ?  let-'a'+1
                      :  let-'A'+('z'-'a'+1)+1);
}
int prima (char* palavra) {
  int i, somaLet=0;
  for (i=0; i < max && palavra[i] != '\0'; i++)
    somaLet += valor(palavra[i]);
  return primo(somaLet);
}
int primo (int n) {
  if (n%2 == 0) return 0;
  int k = 3, sqrtn = sqrt((double)n);
  // Se existir divisor próprio de n, tem que existir divisor próprio menor que √n.
  while (k <= sqrtn) {
    if (n%k == 0) return 0;
    k += 2;
  }
  return 1;
}
int ler(char* palavra) {
  // Retorna verdadeiro sse leitura com sucesso.
  return (scanf("%s",palavra) == 1);
}
```

Figura 8.1: Solução de PAPRIMAS com teste simplificado de primalidade

Uma solução que usa um teste de primalidade baseado no algoritmo conhecido como *crivo de Eratóstenes* é mostrada a seguir:

```
#include <stdio.h> // define scanf, printf
#include <stdlib.h> // define malloc

const int max = 21;
int prima(char* palavra, int* primos, int);
int ler(char* palavra);
int* listaDePrimos(int); // Criada com o algoritmo Crivo de Eratóstenes

int main () {
  const int m = max*(2*('z'-'a'+1));
  // Primalidade deve ser testada para valores menores ou iguais a m
  char palavra[max]; int *primos = listaDePrimos(m);
  while (ler(palavra))
    printf("It is%s a prime word.\n", prima(palavra,primos,m) ?  "" : " not");
}
int minusc (char let) { ...como na Figura 8.1 ...  }
int valor (char let) { ...como na Figura 8.1 ...  }
int* listaDePrimos(int m) {
  int p=2, m2=m+2, nums[m2], *primos = malloc(m),
      // nums[0],nums[1] não usados (por simplicidade, i.e.
      // para que índice de nums corresponda a número entre 2 e m);
      i,j,k;
  for (i=2,j=0; i<m2; i++,j++) { nums[i]=0; primos[j] = 0; }
  j = 0;
  do {
    // marca múltiplos de p
    for (i=p; i<m2; i+=p) nums[i] = 1;

    // procura próximo valor não marcado a partir de j
    for (k=p+1; k<m2; k++) if (!nums[k]) break;
    p = k; // p é o próximo primo
    primos[j] = p;
    j++;
  } while (p<m2);
  return primos;
}
int primo (int n, int* primos, int m) {
  int i;
  for (i=0; i<m; i++)
    if (primos[i] >= n) return (primos[i]==n);
    else if (primos[i] == 0) return 0;
  return 0;
}
int ler (char* palavra) { ...como na Figura 8.1 ...  }
int prima (char* palavra, int* primos, int m) {
  int i, somaLet=0;
  for (i=0; i < max && palavra[i] != '\0'; i++)
    somaLet += valor(palavra[i]);
  return primo(somaLet,primos,m);
}
```

O algoritmo do *Crivo de Eratóstenes*, inventado por Eratóstenes em 350 A.C., cria uma lista de todos os primos menores que um valor máximo m. Para o problema PAPRIMAS, existe tal valor máximo, que pode ser definido como vinte vezes o valor máximo possível para uma eletra (uma vez que podem ocorrer no máximo 20 letras em uma palavra). O algoritmo de Eratóstenes consiste no seguinte:

1. Criar lista l com todos os inteiros de 2 a m.
2. Atribuir, inicialmente, 2 a p.
3. Repetir o seguinte, até que não exista valor não removido em l a partir de p:
   - remover de l todos os múltiplos de p;
   - em seguida, fazer p igual ao número seguinte a p, em l, ainda não removido.
4. Retornar a lista dos números não removidos.

O programa cria um arranjo contendo todos os primos de 3 até m e, para verificar se um dado número n é primo, percorre esse arranjo usando uma *pesquisa sequencial* (de componente a componente), até encontrar um número primo igual ou maior a n ou até chegar ao final do arranjo.

Essa pesquisa sequencial em um arranjo ordenado é ineficiente. Um algoritmo mais eficiente, chamado de *pesquisa binária*, é usado a seguir.

```
struct ListaETamanho {
  int* lista;
  int tam;
};
typedef struct ListaETamanho ListaETamanho;

ListaETamanho listaDePrimos (int m) {
  int p=2, m2=m+2, nums[m2], *primos = malloc(m),
      i,j,k;
  for (i=2,j=0; i<m2; i++,j++) { nums[i]=0; primos[j] = 0; }
  j = 0;
  do {
    for (i=p; i<m2; i+=p) nums[i] = 1;
    for (k=p+1; k<m2; k++) if (!nums[k]) break;
    p = k; // p é o próximo primo
    primos[j] = p;
    j++;
  }
  while (p<m2);
  ListaETamanho l;
  l.lista = primos;
  l.tam = j;
  return l;
}
int primo (int n, ListaETamanho primos) {
  int linf=0, lsup=primos.tam-1, meio = (linf+lsup)/2;
  while (linf<lsup-1) {
    if ((primos.lista)[meio] > n) { lsup = meio; meio = (linf+lsup)/2;} else
    if ((primos.lista)[meio] < n) { linf = meio; meio = (linf+lsup)/2;} else
    return 1;
  } return ((primos.lista)[linf]== n || (primos.lista)[lsup]==n);
}
```

O algoritmo de pesquisa binária compara o elemento que está sendo procurado com o valor que está na metade do arranjo ordenado, permitindo assim que a busca prossiga ou na metade inferior ou na superior do arranjo, conforme o elemento a ser procurado seja menor ou maior, respectivamente, do que o valor que está na metade do arranjo. Se o elemento a ser procurado é igual ao elemento na metade, então, é claro, a busca termina com sucesso.

São mostradas apenas as funções modificadas, que são *listaDePrimos* e *primo*. A função *listaDePrimos* é modificada apenas para retornar, além do arranjo contendo os primos, o número de primos de fato armazenado nesse arranjo. A função *primo* percorre esse arranjo usando pesquisa binária.

8.3 ENERGIA

8.4 CIRCUITO

8.5 POLEPOS

Referências Bibliográficas

[1] Adele Goldberg, David Robinson. *Smalltalk-80: The Language*. Addison-Weslay, 1989.

[2] Andrew Tanenbaum. *Organização Estruturada de Computadores*. LTC (tradução), 2001.

[3] Ângelo Guimarães, Newton Lages. *Algoritmos e Estruturas de Dados*. Ltc Editora, 1988.

[4] Brian Kernighan, Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[5] Bryan O'Sullivan, Don Stewart, John Goerzen. *Real World Haskell*. O'Reilly, 2008.

[6] Cordelia Hall, John O'Donnell. *Discrete Mathematics Using a Computer*. Springer, 2000.

[7] David Patterson, John Hennessy. *Computer Organization and Design: The Hardware Sofwtare Interface*. Morgan Kaufmann, 2ª edition, 2002.

[8] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[9] Fethi Rabhi and Guy Lapalme. *Algorithms: A functional programming approach*. Addison-Wesley, 1999.

[10] James Gosling et al. *The Java Language Specification*. Addison-Wesley, 2ª edition, 2000.

[11] J. Ichbiah. Rationale for the design of the Ada programming language. *ACM SigPlan Notices*, 14(6B), 1979.

[12] Kathleen Jensen, Nicklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, 1991 (edições anteriores: 1974, 1985).

[13] Keith Devlin. *Mathematics: The New Golden Age*. Penguin Books & Columbia University Press, 1999.

[14] Leon Sterling, Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1994.

[15] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[16] Nicklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.

[17] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1985.

[18] Nivio Ziviani. *Projeto de Algoritmos com Implementções em Pascal e C*. Pioneira Thomson, 2004.

[19] Ole-Johen Dahl, Björn Myrhaug, Kristen Nygaard. Simula — an Algol-based Simulation Language. *Communications of the ACM*, 9(9):671–678, 1966.

[20] Lawrence Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. Second edition.

[21] Robert Sedgewick and Kevin Wayne. *Algorithms in C: Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley, 3ª edition, 1998.

[22] Roger Penrose. *The Emperor's New Mind: Concerning Computers, Minds and The Laws of Physics*. Oxford University Press, 1989.

[23] Samuel Harbison. *Modula-3*. Prentice Hall, 1992.

[24] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[25] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3ª edition, 1991.

[26] T. H. Cormen and others. *Introduction to Algorithms*. MIT Press, 2001.

[27] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2ª edition, 1999.

[28] Ulf Nilsson, Jan Maluszynski. *Logic Programming and Prolog*. John Wiley & Sons, 2ª edition, 1995.

[29] Daniel Velleman. *How to Prove It: A Structured Approach*. Cambridge University Press, 2ª edition, 2006.

[30] William Stallings. *Arquitetura e Organização de Computadores: Projeto para o Desempenho*. Pearson Education do Brasil (tradução), 5ª edition, 2002.

Apêndice A

Escolha da linguagem `C`

O livro adota a linguagem de programação `C`. A escolha dessa linguagem foi motivada pela necessidade de homogeneização no ensino de disciplinas introdutórias de programação de computadores nos cursos de várias universidades (por exemplo, nas disciplinas de *Algoritmos e Estruturas de Dados I* do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais). O uso da linguagem `C` apresenta, reconhecidamente, vantagens no ensino de tais disciplinas para cursos como os de Engenharia Elétrica e Engenharia de Controle e Automação, por se tratar de linguagem adequada à chamada *programação de sistemas*, na qual se faz acesso direto a dispositivos e recursos de hardware. Para tais sistemas, a programação na linguagem `C` é adequada pois a linguagem permite acesso direto aos dispositivos e recursos de hardware, e portanto bibliotecas e programas que fazem tais acessos diretos ao hardware podem ser mais facilmente encontrados e usados. Para outros cursos, o uso da linguagem `C` é controverso, pelo fato de existir na linguagem uma preocupação central com eficiência, e possibilidade de acesso direto a áreas de memória, o que leva, principalmente, a duas consequências indesejáveis do ponto de vista de um aprendizado em programação:

1. Ausência, em muitos casos, de verificação dos chamados *erros de tipo*. Tais erros ocorrem devido ao uso de valores em contextos inadequados, ou seja, em contextos nos quais não faz sentido usar tais valores, e portanto nos quais tais valores não deveriam ser usados.

   Grande parte do desenvolvimento das linguagens de programação nos dias atuais é relacionado ao objetivo de tornar as linguagens cada vez mais seguras, no sentido de possibilitar a deteção de um número cada vez maior de erros de tipo, e ao mesmo tempo dando flexibilidade ao programador, de modo que ele não tenha que especificar ou mesmo se preocupar com a anotação explícita de tipos em seus programas, e procurando manter o algoritmo que permite essa deteção de erros simples e eficiente.

2. Mecanismos adequados na linguagem para suporte a abstrações, conceitos e construções comumente usados em programas, de forma simples e segura. Exemplos de tais mecanismos são:

   - Tipos algébricos: tipos que permitem representar disjunção (*"ou"*) entre tipos e estruturas de dados, de modo seguro e simples. Tipos algébricos são usados comumente, para representar estruturas muito comuns em computação, tais como:
     - *enumerações*, denotadas por tipos que consistem em uma enumeração de todos os possíveis elementos do tipo, como por exemplo um tipo booleano (cada valor consistindo de falso *ou* verdadeiro), ou estações do ano (primavera, verão, outono ou inverno), etc.;
     - *alternativas* que estendem enumerações de modo a permitir diferentes modos de construir valores, tal como por exemplo uma forma geométrica, que pode ser um círculo com o tamanho de seu raio, *ou* um retângulo com os tamanhos de seus dois lados etc.
     - *estruturas recursivas* que estendem alternativas por permitir recursividade na sua definição, como por exemplo uma lista, que pode ser vazia *ou* um elemento seguido

do restante da lista (tal restante consistindo, recursivamente, de uma lista), uma árvore binária, que pode ser uma folha contendo um valor de um determinado tipo *ou* um par formado por duas subárvores, etc.;

- Polimorfismo: tanto de valores (estruturas de dados) que podem ser instanciados para quaisquer tipos, mantendo a mesma forma, quanto de funções que realizam operações sobre tais estruturas de dados, que funcionam do mesmo modo, independentemente da instância sobre a qual realizam a operação.

  A ausência de suporte a polimorfismo em `C` impede, em particular, que existam bibliotecas com funções de manipulação de estruturas de dados de propósito geral como pilhas, filas, dicionários, conjuntos etc., comuns em linguagens mais modernas.

Leitores interessados em tais assuntos podem consultar as notas bibliográficas do Capítulo 2.

Em conclusão, existe um compromisso entre a intenção de formar durante o tempo da graduação um programador pronto para as necessidades da indústria e o objetivo didático primordial que consiste em oferecer uma formação sólida aos graduandos nos aspectos teóricos relacionados à programação de computadores. Pelo primeiro motivo, este livro foi escrito na linguagem `C`. Pelo segundo, se oferecem seções especiais que abordam temas destinados aos estudantes que procurem maior aprofundamento nos conteúdos lecionados.

Índice Remissivo