Algoritmos

Prof. Wander Gaspar, D.Sc.

29 de janeiro de 2016

Sumário

Prefácio

Notas de aula da disciplina **Algoritmos** ministrada aos alunos do **Bacharelado em Sistemas de Informação** do CES/JF.

Horário das aulas no primeiro semestre letivo de 2016: segunda-feira e quarta-feira de 20:50 às 22:30. As avaliações estão agendadas conforme cronograma seguinte:
Primeira avaliação.: 06/abril
Segunda avaliação.: 18/maio
Terceira avaliaçã.o: 29/junho

É de competência do professor lançar semanalmente a frequência no sistema acadêmico. A disciplina possui 4 créditos, o que equivale a 72 horas/aula. O limite máximo de faltas, que corresponde a 25 porcento, representa 18 horas/aula.

Contato com o professor pelos e-mails `wandergaspar@pucminas.cesjf.br` e `wandergaspar@gmail.com`

Capítulo 1

Introdução aos Algoritmos

Não se pode criar experiência. É preciso passar por ela. – Albert Einstein

1.1 Introdução à Organização de Computadores

1.1.1 Modelo de von Neumann

O modelo lógico de computador proposto por von Neumann pode ser descrito pela Figura 1.1.

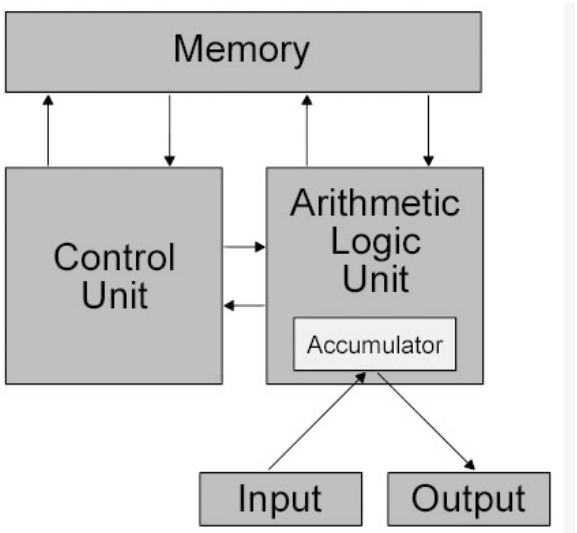

Figura 1.1: Modelo de Arquitetura de von Neumann

- **Unidade de controle:** a unidade de controle controla o funcionamento da unidade lógica e aritmética e da memória. Além disso, ela distribui e organiza tarefas,

transfere informações da entrada para a memória e da memória para a saída.

- **Memória:** a memória é o dispositivo de armazenamento de informação do computador. Nela são armazenados tanto as instruções de um programa quanto os dados necessários para a sua execução. A memória é dividida em espaços com endereços.

- **Unidades de entrada e saída:** a unidade de entrada traduz informação (por exemplo, letras, números, imagens, marcas magnéticas) de uma variedade de dispositivos de entrada em impulsos elétricos que a CPU entenda. A unidade de saída traduz os dados processados, enviados pela CPU em forma de impulsos elétricos, em palavras ou números, que são impressos por impressoras ou mostrados em monitores de vídeo.

1.1.2 Bases de Numeração

Por questões de Engenharia (confiabilidade, menor consumo de energia, menor dissipação de calor), os computadores utilizam a base numérica *binária* (dígitos 0 e 1) em lugar da base numérica decimal (dígitos 0 e 9).

Outras bases numéricas também utilizadas em Ciência da Computação são a *octal* (dígitos 0 a 7) e *hexadecimal* (dígitos 0 a 9 e A a F). Isso se deve à fácil conversão entre a base binária e as bases octal e hexadecimal associado à uma representação mais concisa e inteligível (Figura 1.2).

Cada 4 dígitos em um número na representação binária corresponde a um número na representação hexadecimal. Por exemplo, $(101111001101)_2 = (BCD)_{16}$ e $(13A)_{16} = (000100111010)_2$.

Analogamente, cada 3 dígitos binários correspondem a um dígito octal. Por exemplo, $(101111001101)_2 = (5715)_8$. Em decimal, $5 \times 8^3 + 7 \times 8^2 + 1 \times 8^1 + 5 \times 8^0 = (3021)_{10}$.

1.1.3 Bit

Bit (simplificação para dígito binário, *BInary digiT* em inglês) é a menor unidade de informação que pode ser codificada e manipulada em um computador digital. Um bit tem um único valor, 0 ou 1, ou verdadeiro ou falso, ou neste contexto, quaisquer dois valores mutuamente exclusivos.

Fisicamente, o valor de um bit é armazenado como uma carga elétrica acima ou abaixo de um nível padrão em um único capacitor dentro de um dispositivo de memória. Mas, bits podem ser representados fisicamente por vários meios: luz (em fibras óticas), ondas eletromagnéticas (rede *wireless*), polarização magnética (discos rígidos).

Base 2	Base 8	Base 10	Base 16
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
10000	20	16	10
10001	21	17	11

Figura 1.2: Bases de Numeração utilizadas em Computação

1.1.4 Byte

Denomina-se *byte* (contração de *BinarY term*) à unidade básica de tratamento de informação em computadores digitais. Por padronização de mercado, 1 byte equivale a 8 bits contíguos. A primeira codificação de 1 byte = 8 bits deveu-se à empresa IBM em 1960 com a criação da tabela ASCII.

Como cada bit pode conter 2 valores binários diferentes, 1 byte (8 bits) pode representar 256 valores numéricos distintos na base 2. A Tabela 1.1 apresenta os valores mínimo e máximo que podem ser armazenados em 1 byte.

Tabela 1.1: Valores mínimo e máximo em 1 byte

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

$$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 255$$

Cada byte pode armazenar um caractere (letra, número ou símbolo), de acordo com alguma tabela de codificação.

1.1.5 Codificação ASCII

ASCII (acrônimo para *American Standard Code for Information Interchange*, que em português significa "Código Padrão Americano para o Intercâmbio de Informação") é uma codificação de caracteres de oito bits criado pela IBM. Originalmente ASCII usava apenas 7 bits (capaz de representar 128 caracteres) mas posteriormente estendeu-se para 8 bits para suportar os caracteres acentuados usados em diversos idiomas como o português.

A Figura 1.3 apresenta a codificação ASCII para os principais caracteres imprimíveis dos idiomas ocidentais.

1.1.6 Grandezas Derivadas do Byte

Uma vez que os computadores utilizam aritmética binária (base 2), os valores numéricos são expressos em potências de dois e não em potências de dez, embora sejam usados os mesmos nomes para os múltiplos.

Assim, 1 quilobyte (KB) equivale a 2^{10} bytes = 1.024 bytes, 1 megabyte (MB) equivale a 2^{20} bytes = 1.048.576 bytes e 1 gigabyte (GB) equivale a 2^{30} bytes = 1.073.741.824 bytes.

1.1.7 Palavra

Uma palavra (*word*) é um valor fixo para um determinado processador. Assim, um Pentium possui uma palavra de 32 bits. Isso significa que 1 palavra de 32 bits são processados por vez nesse processador e indica a sua capacidade de processamento.

1.2 Memória

Memória é o componente de um computador cuja função é armazenar informações a serem manipuladas pelo sistema. A Figura 1.4 apresenta uma analogia entre uma memória de computador e um arquivo de fichas em uma empresa.

Na realidade, existem diversos tipos de memória em um computador:

- **Memória principal:** (memória RAM *Random Access Memory* Memória de Aceso Randômico).
- **Memória cache:** construída com tecnologia RAM, inserida entre a UCP e a memória principal.
- **Registradores:** dispositivos de armazenamento existentes no interior dos processadores, com o objetivo de conter os dados, instruções e endereços de memória RAM a serem processados pela UCP.

Binário	Decimal	Hexa	Glifo
0010 0000	32	20	
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binário	Decimal	Hexa	Glifo
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binário	Decimal	Hexa	Glifo
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

Figura 1.3: Caracteres imprimíveis da tabela ASCII

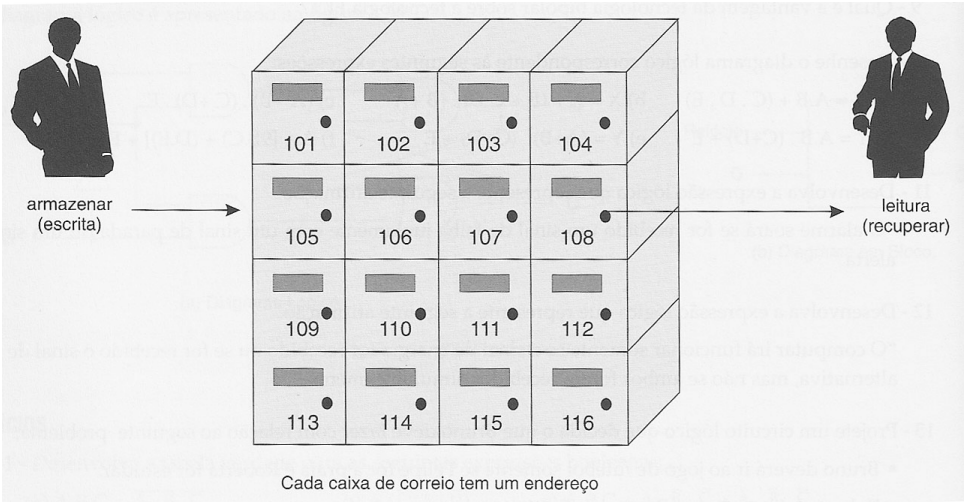

Figura 1.4: Analogia entre memória de computador e fichário manual

- **Memória secundária:** discos rígidos, mídias óticas, *pen drives*.

A A Figura 1.5 apresenta a relação entre o custo e o desempenho dos tipos de memória existentes.

A memória principal é endereçada por bytes (células), começando pelo endereço 0 (byte 0) até o último byte disponível. A A Figura 1.6 apresenta um exemplo de alocação de bytes na memória principal de um computador.

1.3 Programação de Computadores

Uma *linguagem de programação* é uma codificação criada para instruir um computador a realizar uma determinada tarefa. Para que um computador execute qualquer tarefa é necessário fornecer um código, escrito em uma linguagem de programação.

1.3.1 Linguagem de Máquina

O tipo mais primitivo de linguagem de programação é chamado de linguagem de máquina, assim chamado porque é a única codificação que o computador entende diretamente. Os primeiros computadores baseados no modelo de Von Neumann utilizavam unicamente a programação em linguagem de máquina. A Figura 1.7 apresenta um trecho de um programa codificado em linguagem de máquina.

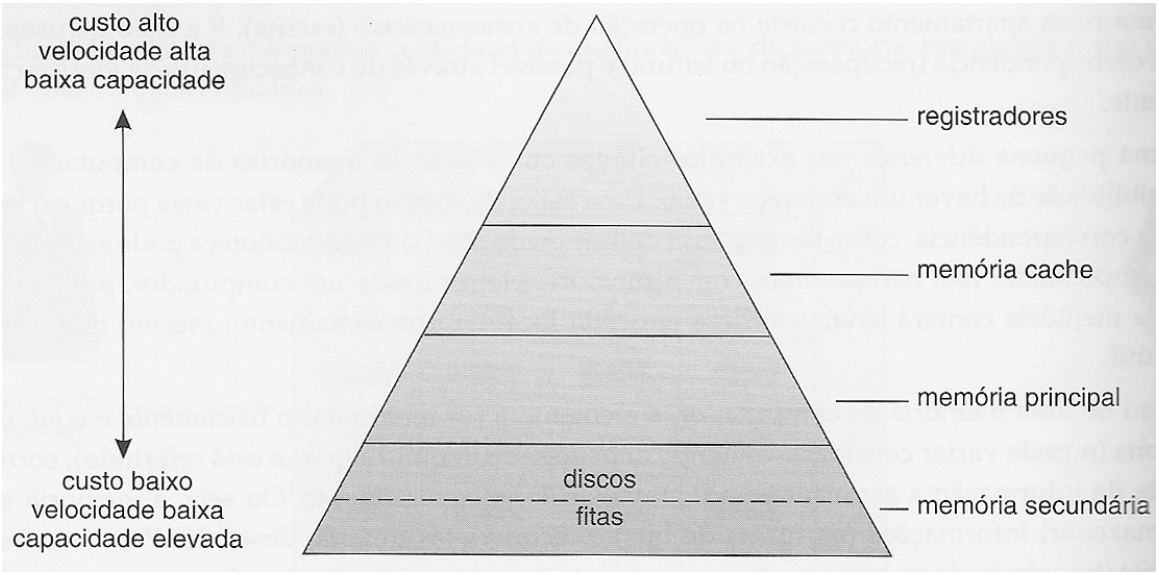

Figura 1.5: Relação custo *versus* desempenho dos tipos de memória

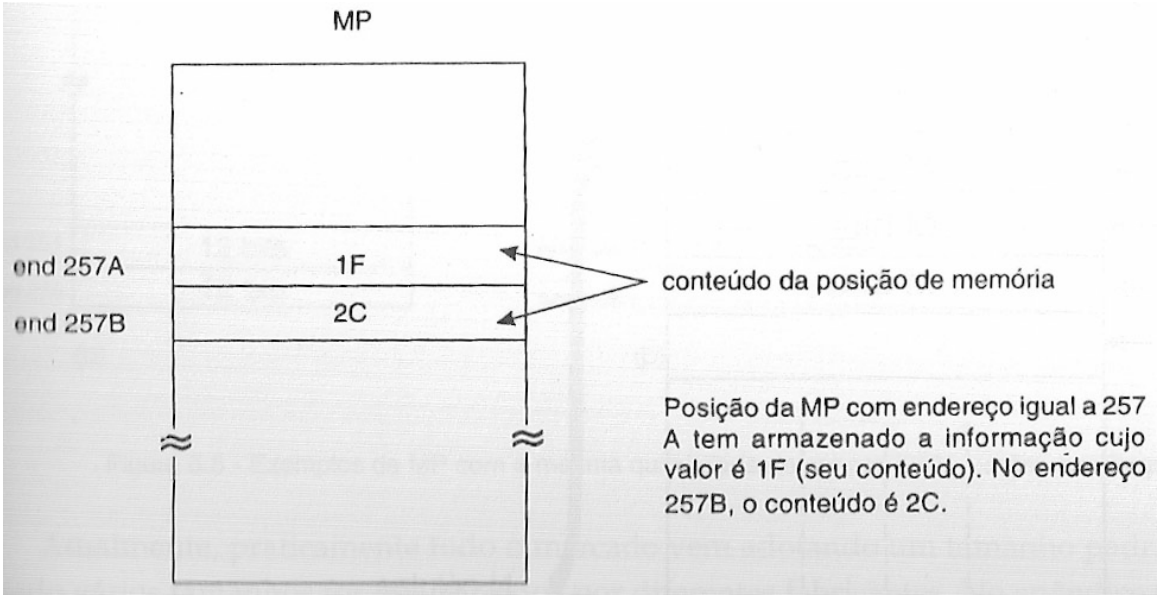

Figura 1.6: Alocação de bytes na memória do computador

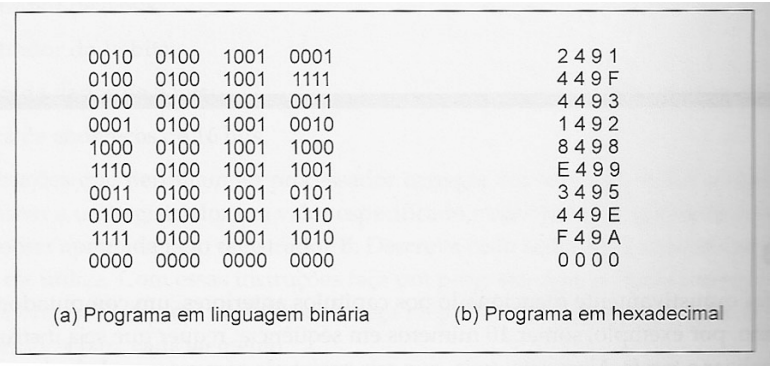

Figura 1.7: Trecho de programa em linguagem de máquina

1.3.2 Linguagem de Montagem

Uma linguagem de montagem ou *assembly* é uma notação legível por humanos para o código de máquina de um computador. A linguagem de montagem foi a primeira evolução no sentido de tornar a programação de computadores mais fácil e menos tediosa. A Figura 1.8 apresenta um trecho de um programa codificado em linguagem *assembly*.

Figura 1.8: Trecho de programa em linguagem de montagem

Por exemplo, enquanto um computador sabe o que a instrução 10110000 01100001 faz, para os programadores é mais fácil recordar a representação equivalente em uma instrução mnemônica da linguagem de montagem MOV AL 61. Tal instrução ordena que o valor hexadecimal 61 (97, em decimal) seja movido para o registrador AL do processador.

1.3.3 Linguagem de Alto Nível

Uma linguagem de alto nível, ou orientada ao problema, permite que o programador especifique a realização de uma tarefa pelo computador de uma forma muito mais próxima da linguagem humana do que da linguagem de máquina ou de montagem.

Uma das principais metas das linguagens de programação de alto nível é permitir

que programadores tenham uma maior produtividade, permitindo expressar suas intenções mais fácil e rapidamente. Linguagens de programação de alto nível também tornam os programas menos dependentes de computadores ou ambientes computacionais específicos (propriedade chamada de portabilidade). Isto acontece porque programas escritos em linguagens de programação de alto nível são traduzidos para o código de máquina específico do computador em que será executado.

Desde o surgimento do *Fortran*, centenas de linguagens de programação já foram desenvolvidas. A Figura 1.9 apresenta algumas das linguagens mais bem sucedidas ao longo dos últimos 50 anos.

Linguagem	Data	Observações
FORTRAN	1957	FORmula TRANslation — primeira linguagem de alto nível. Desenvolvida para realização de cálculos numéricos.
ALGOL	1958	ALGOrithm Language — linguagem desenvolvida para uso em pesquisa e desenvolvimento, possuindo uma estrutura algorítmica.
COBOL	1959	COmmon Business Oriented Language — primeira linguagem desenvolvida para fins comerciais.
LISP	1960	Linguagem para manipulação de símbolos e listas.
PL/I	1964	Linguagem desenvolvida com o propósito de servir para emprego geral (comercial e científico). Fora de uso.
BASIC	1964	Desenvolvida em universidade, tornou-se conhecida quando do lançamento do IBM-PC, que veio com um interpretador da linguagem, escrito por Bill Gates e Paul Allen.
PASCAL	1968	Primeira linguagem estruturada — designação em homenagem ao matemático francês Blaise Pascal que, em 1642, foi o primeiro a planejar e construir uma máquina de calcular.
C	1967	Linguagem para programação de sistemas operacionais e compiladores.
ADA	1980	Desenvolvida para o Departamento de Defesa dos EUA.
DELPHI	1994	Baseada na linguagem Object Pascal, uma versão do Pascal orientada a objetos.
JAVA	1996	Desenvolvida pela Sun, sendo independente da plataforma onde é executada. Muito usada para sistemas Web.

Figura 1.9: Linguagens de alto nível bem sucedidas no últimos 50 anos

1.3.4 Compilação

Compilação é o processo de análise de um programa escrito em uma linguagem de alto nível, chamado programa fonte (ou código fonte) e sua conversão (ou tradução) em um programa equivalente em linguagem binária de máquina (programa objeto ou código objeto).

O processo de compilação de um programa fonte em programa objeto é feito pelo *compilador* da linguagem. Assim, um compilador C converte um código fonte escrito em linguagem C para a linguagem de máquina do computador subjacente. A Figura 1.10 apresenta o processo de compilação.

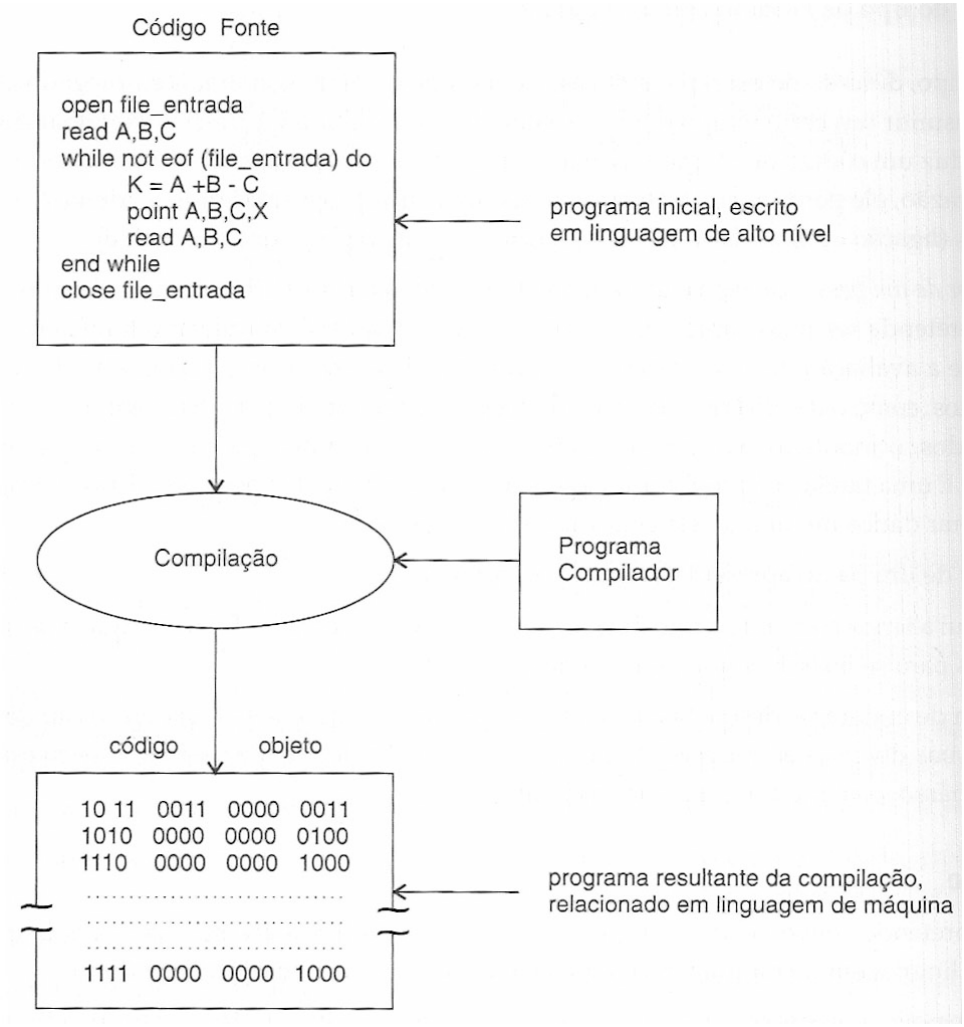

Figura 1.10: Processo de compilação

1.3.5 Ligação

Existe uma série de tarefas que o programador não precisa incluir no código fonte. Exemplos dessas tarefas são como apresentar informação na tela, como obter dados do teclado, como ler ou gravar em um disco rígido. O código para essas tarefas é em geral organizado em arquivos formando uma biblioteca (*library*) de rotinas fornecida em conjunto com o compilador da linguagem.

O que importa ao programador é ligar (*linkeditar*) o código objeto gerado pelo processo de compilação com as rotinas da biblioteca necessárias para a execução do programa. A Figura 1.11 apresenta o processo de linkedição.

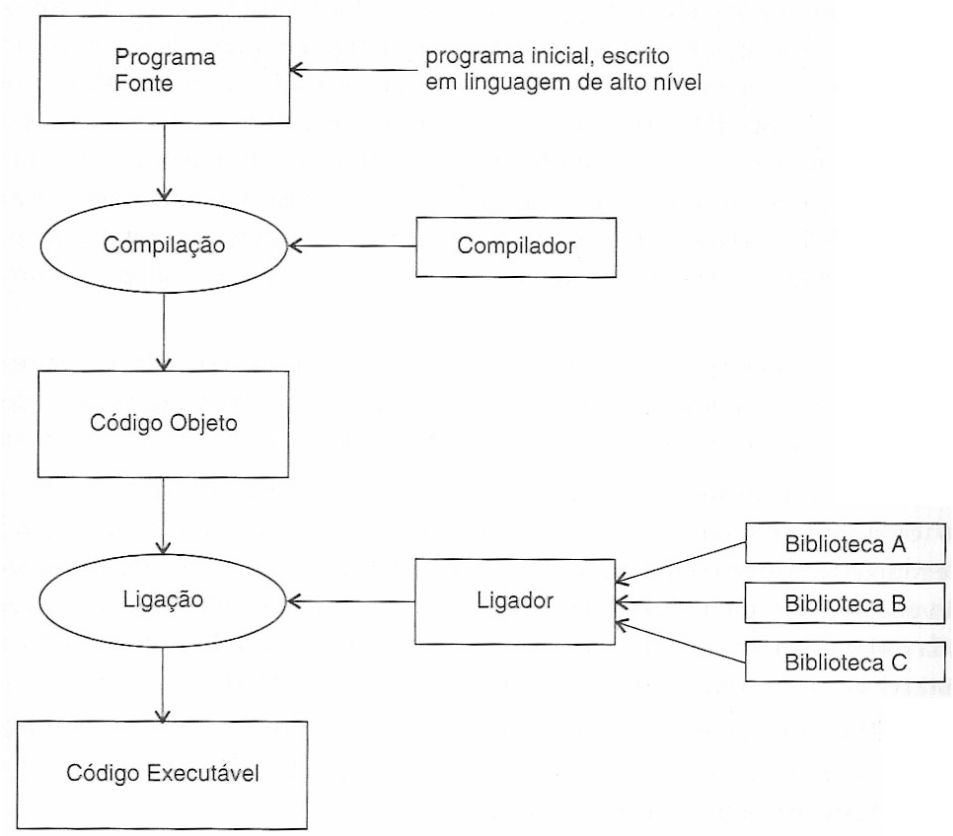

Figura 1.11: Processo de linkedição

Todo esse processo seria muito simples se não ocorressem erros. Na prática, o programador deve lidar com a possibilidade de acontecerem erros ao programar. Por exemplo, o compilador não irá gerar o código objeto se houver algum erro no código fonte (erro de compilação). Outro exemplo, um programa executável não será gerado a partir de um código objeto se uma ou mais rotinas disponíveis nas bibliotecas da linguagem não forem encontradas (erro de linkedição).

Mesmo que todo o processo de compilação e ligação funcione corretamente, gerando um programa executável, isso não é garante que o programador não tenha cometido erros. É necessário verificar se a ação do programa corresponde de fato ao que foi projetado (erro de lógica).

Portanto, o diagrama apresentado na Figura 1.12 ilustra de forma mais realística a realidade do trabalho do programador.

1.4 Algoritmos

Um *algoritmo* consiste em uma sequência de instruções sem ambiguidade que é executada até que determinada condição se verifique.

O conceito de algoritmo é frequentemente ilustrado pelo exemplo de uma receita, embora muitos algoritmos sejam mais complexos. Eles podem repetir passos (fazer iterações) ou necessitar de decisões (tais como comparações ou lógica) até que a tarefa seja completada. Um algoritmo corretamente executado não irá resolver um problema se estiver implementado incorretamente ou se não for apropriado ao problema.

Um algoritmo não representa, necessariamente, um programa de computador, e sim os passos necessários para realizar uma tarefa. Sua implementação pode ser feita por um computador, por outro tipo de máquina ou mesmo por um ser humano.

Diferentes algoritmos podem realizar a mesma tarefa usando um conjunto diferenciado de instruções em mais ou menos tempo, espaço ou esforço do que outros. Tal diferença pode ser reflexo da complexidade computacional aplicada. Por exemplo, um algoritmo para se vestir pode especificar que você vista primeiro as meias e os sapatos antes de vestir a calça enquanto outro algoritmo especifica que você deve primeiro vestir a calça e depois as meias e os sapatos. Fica claro que o primeiro algoritmo é mais difícil de executar que o segundo apesar de ambos levarem ao mesmo resultado.

Etimologia: a palavra algoritmo tem origem no sobrenome, *Al-Khwarizmi*, do matemático persa do século IX, Mohamed ben Musa, cujas obras foram traduzidas no ocidente cristão no século XII, tendo uma delas recebido o nome "Algorithmi de numero indorum", sobre os algoritmos usando o sistema de numeração decimal.

1.4.1 Pseudocódigo

Pseudocódigo é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples (nativa a quem o escreve, de forma a ser entendida por qualquer pessoa) sem necessidade de conhecer a sintaxe de nenhuma linguagem de programação.

Alguns livros introdutórios de programação utilizam algum tipo de pseudocódigo para ilustrar os seus exemplos, de forma que todos os estudantes possam entendê-los independentemente da linguagem que vierem a utilizem. O Exemplo 1.1 apresenta um pseudocódigo em língua portuguesa.

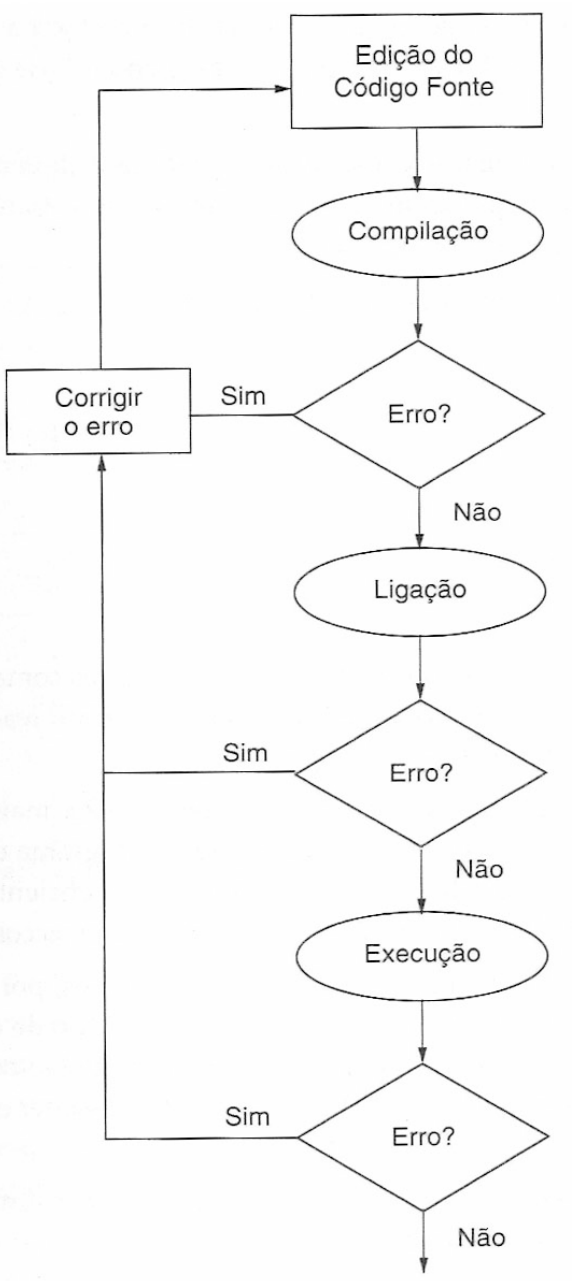

Figura 1.12: Processo de linkedição

Exemplo 1.1. *Exemplo de pseudocódigo.*

```
escreva: Qual é a nota do aluno?
leia nota
se nota maior ou igual a sete
```

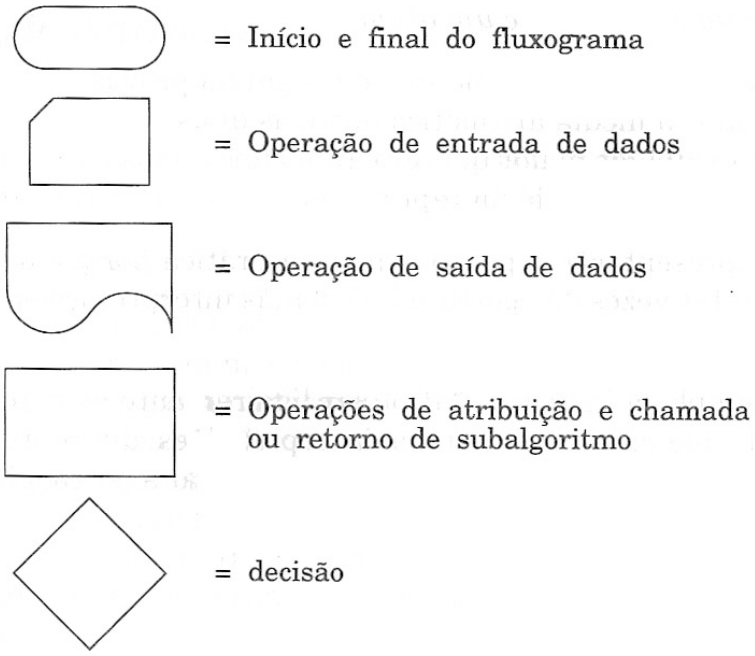

Figura 1.13: Símbolos usados em fluxogramas.

```
  então:
    escreva Ele passou
  senão:
    escreva: Ele foi reprovado
fim do se
fim do programa
```

Embora no caso da língua portuguesa existam alguns interpretadores de pseudocódigo, nenhum tem a projeção das linguagens Pascal ou C, que no caso da língua inglesa se assemelham bastante a um pseudocódigo.

1.4.2 Fluxograma

Fluxograma é um tipo de diagrama, e pode ser entendido como uma representação esquemática de um processo. Em programação, trata-se de uma ferramenta para a elaboração de algoritmos que, em uma etapa posterior, podem usados como base para que o programador escreva um programa fonte.

Não existe uma padronização para os símbolos utilizados em fluxogramas. A Figura 1.13 apresenta alguns símbolos comumente empregados em fluxogramas para representar algoritmos.

Capítulo 2

Introdução à Programação em C

Nada é complicado se nos prepararmos previamente – Confúcio

2.1 Linguagem de Programação C

C é uma linguagem de programação criada em 1972, por Dennis Ritchie, no AT&T Bell Labs, para desenvolver o sistema operacional UNIX (que foi originalmente escrito em Assembly). Desde então, tornou-se uma das linguagens de programação mais usadas, e influenciou muitas outras, especialmente C++, que foi desenvolvida como uma extensão de C.

Em 1978, Brian Kernighan e Dennis Ritchie publicaram a primeira edição do livro *The C Programming Language*. Esse livro, conhecido pelos programadores de C, como "K&R", serviu durante muitos anos como uma especificação informal da linguagem.

No fim da década de 1970, C começou a substituir BASIC como a linguagem de programação de microcomputadores mais usada. Durante a década de 1980, foi adaptada para uso no PC IBM, e a sua popularidade começou a aumentar significativamente. Ao mesmo tempo, Bjarne Stroustrup e sua equipe nos laboratórios Bell, começou a trabalhar num projeto onde se adicionavam construções de orientação à objetos à linguagem C. O produto desse trabalho, chamado C++, é nos dias de hoje a linguagem de programação mais usada no desenvolvimento do sistema operacional Windows e da maioria das aplicações de grande porte; C permanece mais popular no mundo UNIX.

Em 1983, o instituto norte-americano de padrões (ANSI) formou um comitê para estabelecer uma especificação do padrão da linguagem C. Após um processo longo e árduo, o padrão foi completo em 1989 e ratificado como ANSI X3.159-1989 *Programming Language C*. Esta versão da linguagem é frequentemente referida como ANSI C. Em 1990, o padrão ANSI C, após sofrer umas modificações menores, foi adotado pela Organização Internacional de Padrões (ISO) como ISO/IEC 9899:1990.

2.1.1 Ambiente de Desenvolvimento Integrado para C

IDE, do inglês *Integrated Development Environment* ou Ambiente Integrado de Desenvolvimento, é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo.

As características e ferramentas mais comuns encontradas nos IDEs são:

- **Editor:** edita o código fonte do programa escrito nas linguagens suportadas pelo IDE;
- **Compilador:** compila o código fonte do programa em código objeto;
- **Linkeditor:** liga o código objeto às bibliotecas necessárias e gera um código executável;
- **Depurador:** (debugger) auxilia no processo de encontrar e corrigir erros (*bugs*) no código fonte do programa, na tentativa de aprimorar a qualidade de software;

Os principais IDES disponíveis atualmente como software livre e compatíveis com o desenvolvimento em linguagem C na plataforma Windows são: Netbeans, Eclipse, Visual Studio Community Edition, Dev-C++ e Code::Block.

Na disciplina **Introdução à Computação e Algoritmos** do CES/JF será utilizado o IDE Dev-C++ versão 5.6.2 ou superior disponível em `http://sourceforge.net/projects/orwelldevcpp/`.

2.1.2 Exemplo Preliminar

Escreva seu primeiro programa fonte em linguagem C. Depois salve-o com o nome `Exemplo1.c`, compile e execute-o no IDE Dev-C. Para escrever um novo código fonte, a partir do menu principal, selecione *Arquivo* → *Novo* → *Arquivo Fonte*. Digite as linhas de código a seguir.

```
#include <stdio.h>
int main() {
    printf("Alo, mundo!");
}
```

Para compilar, *linkeditar* e executar o programa, selecione *Executar* → *Compilar & Executar* no menu principal do DEV-C ou utilize a *hot-key* F11. Se não houver erros de compilação ou ligação, a execução deverá apresentar em uma janela do DOS a mensagem: `Alo, mundo!` seguido de `Pressione qualquer tecla para continuar...`

Entendendo o Exemplo Preliminar

É preciso entender todos os componentes desse programa fonte C. O termo `#include` trata-se de uma diretiva ao pré-processador. O arquivo de cabeçalho `<stdio.h>` é necessário

sempre que o programa C utilizar a instrução `printf()`.

O programa C é composto de uma ou mais funções. Uma função é um bloco de código fonte que possui um nome e é delimitado por chaves de abertura e fechamento. `main()` é a função mais importante em C porque a partir dela é que tem início a execução do programa. No exemplo preliminar, a função `main()` possui 3 linhas de código.

A linha de código `printf("Alo, mundo!");` é uma instrução para exibir alguma informação na saída padrão do computador (vídeo). Nesse exemplo, a informação a ser apresentada é a cadeia de caracteres `"Alo, mundo!"`. Observe que uma cadeia de caracteres deve ser delimitada por aspas duplas.

A instrução `printf()` também é uma função, assim como `main()`, porém, não precisa ser codificada pelo programador. Ela faz parte da biblioteca de recursos da linguagem C.

Uma função pode receber argumentos para serem manipulados em seu código. Os argumentos devem ser inseridos entre parênteses, logo após o nome da função. Assim, `"Alo, mundo!"` é fornecido como argumento para a função `printf()`. Observe que `main` é seguido por `(void)` que significa, nesse contexto, sem argumentos, ou seja, nesse exemplo, nenhum argumento deve ser passado à função `main()`.

Todas as três linhas de código no interior de `main()` terminam em ponto-e-vírgula. Isso é uma exigência do compilador C: toda instrução tem que terminar com esse caractere delimitador.

Os termos `include`, `int` e `printf` são exemplos de palavras reservadas da linguagem C. Isso implica que possuem um significado único e definido para o compilador. Se você digitar erradamente alguma delas o compilador acusará erro.

2.2 Saída de Dados na Saída Padrão

A função `printf()` é a forma mais usual de se exibir informação na saída padrão, ou seja, no monitor de vídeo. A função `printf()` requer a biblioteca `stdio.h` que deve ser informada em uma diretiva ao pré-processador `#include`.

Uma cadeia de caracteres permite a inclusão de caracteres de controle, identificados pelo delimitador barra invertida (\). O caractere de controle `\n` significa avançar o cursor para o início da próxima linha. O Exemplo 2.1 ilustra o emprego do caractere de controle `\n` para o avanço do cursor.

Exemplo 2.1. *Emprego de* `printf()` *com caractere de controle para avanço do cursor.*

```
#include <stdio.h>
int main() {
    printf("Alo, mundo!\n");
}
```

```
Alo, mundo!
Pressione qualquer tecla para continuar...
```

A simples inclusão de uma nova chamada à função `printf()` não implica no avanço do cursor para o início da próxima linha. Essa ação deve ser explicitamente representada pelo uso do caractere de controle `\n`. O Exemplo 2.2 gera a mesma saída do Exemplo 2.1 porém utilizando duas chamadas à função printf().

Exemplo 2.2. *Função* `printf()` *e uso do caractere de controle para avanço do cursor.*

```
#include <stdio.h>
int main() {
    printf("Alo, ");
    printf("mundo!\n");
}
```

```
Alo, mundo!
Pressione qualquer tecla para continuar...
```

Observe que programas fonte apresentados até o momento utilizam um recurso visual denominado *indentação*, onde os comandos presentes na função `main()` – entre as chaves de abertura e fechamento – possuem um recuo na linha.

O uso correto e consistente da indentação do código fonte consiste em uma boa prática de programação pois é muito importante para facilitar a leitura e o entendimento do algoritmo e deve ser utilizado em todos os programas desenvolvidos.

A Tabela 2.1 apresenta os principais caracteres de controle que podem ser utilizados para a formatação de saída com a função `printf()`.

Tabela 2.1: Caracteres de controle para uso na função `printf()`

Caractere	**Formatação**
\n	Avanço para o início da próxima linha
\t	Avanço para a próxima tabulação
\r	Retorno para o início da linha atual
\b	Retrocesso
\''	Inserção de uma aspas duplas
\'	Inserção de um apóstrofo
\\	Inserção de uma barra invertida

Exercício 2.1. *Escreva um programa C para exibir na tela a saída:*

```
CENTRO DE ENSINO SUPERIOR DE JUIZ DE FORA
ENGENHARIA DE SOFTWARE/ELETRICA

DISCIPLINA: INTRODUCAO A COMPUTACAO E ALGORITMOS
```

Exercício 2.2. *Escreva um programa C para exibir na tela a saída:*

```
65    "A"    66    "B"    67    "C"    68    "D"
```

2.3 Outras Funções para Saída de Dados

A função mais simples para saída de dados em C é `putchar()`, capaz de exibir um único caractere ASCII na saída padrão.

Exemplo 2.3. *Função* `putchar()` *para saída um caractere na saída padrão.*

```
#include <stdio.h>
int main() {
    putchar(65);
    putchar('B');
}
```

Outra função semelhante é `puts()`, que permite exibir na saída padrão uma cadeira de caracteres delimitada por aspas duplas.

Exemplo 2.4. *Função* `puts()` *para saída uma cadeia de caracteres na saída padrão.*

```
#include <stdio.h>
int main() {
    puts("ENGENHARIA");
}
```

2.4 Comentários em C

Uma boa prática de programação consiste em documentar o código fonte, com o objetivo de facilitar o seu entendimento e posterior manutenção. A documentação pode ser realizada através de comentários, conforme apresentado no Exemplo 2.5.

Exemplo 2.5. *Comentários em código fonte C.*

```
/* Programa para exibir uma mensagem na tela */
#include <stdio.h>
```

```
int main() {
    printf("Alo, mundo!\n"); /* saída na tela */
}
```

Qualquer trecho de código fonte delimitado por `/*` e `*/` não é considerado pelo compilador. Você pode inserir comentários em qualquer parte de um programa C.

Comentários de Linha

Como alternativa, C oferece comentários de linha, delimitados por `//`. Comentários de linha podem ser inseridos em qualquer posição de uma linha no código fonte e, nesse caso, o restante da linha será desconsiderado durante o processo de compilação (Exemplo 2.6).

Exemplo 2.6. *Comentários de linha em C.*

```
// Programa para exibir uma mensagem na tela
#include <stdio.h>
int main() {
    printf("Alo, mundo!\n"); // saída na tela
}
```

2.5 Tipos de Dados Primitivos

Os tipos de dados primitivos com os quais C trabalha são os números inteiros e os números reais (também chamados números de ponto flutuante no contexto da linguagem).

A inserção do ponto decimal define para o compilador C um valor numérico como sendo de ponto flutuante e a delimitação por apóstrofos define um valor inteiro como sendo um caractere da tabela ASCII. Caracteres ASCII podem ser armazenados em qualquer tipo de dado inteiro, embora usualmente se usa o tipo `char`.

2.5.1 Tipos Inteiros

A linguagem C oferece mais de um tipo de dado inteiro conforme apresentado na Tabela 2.2, que inclui a faixa de valores válida para o compilador do IDE Dev-C++. Observe que a faixa de valores pode variar entre os diversos compiladores C existentes.

Observe que caracteres da tabela ASCII são normalmente referenciados através do tipo inteiro `char`. Isso está correto. Lembre-se que cada código ASCII corresponde a um número inteiro na tabela.

Os tipos `short int` e `long int` podem ser abreviados por `short` e `long` respectivamente.

Tabela 2.2: Tipos de dados inteiros

Tipo	Bytes	Faixa de Valores
`char`	1	-128 a +127
`short int`	2	-32.768 a +32.767
`int`	4	-2.147.438.648 a +2.147.438.647
`long int`	4	-2.147.438.648 a +2.147.438.647

Modificador `unsigned`

O modificador `unsigned` altera a faixa de valores de cada um dos tipos primitivos apresentados ao descartar valores negativos. A Tabela 2.3 apresenta os valores máximos e mínimos para os tipos inteiros com o modificador `unsigned`.

Tabela 2.3: Tipos de dados inteiros com o modificador unsigned

Tipo	Bytes	Faixa de Valores
`unsigned char`	1	0 a +255
`unsigned short int`	2	0 a +65.535
`unsigned int`	4	0 a +4.294.967.295
`unsigned long int`	4	0 a +4.294.967.295

2.5.2 Tipos de Ponto Flutuante

A linguagem C também oferece mais de um tipo de dado de ponto flutuante para armazenamento de valores numéricos reais, conforme apresentado na Tabela 2.4. Observe que a faixa de valores pode variar entre os diversos compiladores C existentes.

Tabela 2.4: Tipos de dados de ponto flutuante

Tipo	Bytes	Faixa de Valores	Precisão
`float`	4	$\pm 3.4 \times 10^{38}$	6 dígitos decimais
`double`	8	$\pm 1.7 \times 10^{308}$	15 dígitos decimais

2.5.3 Constantes

Uma *constante* representa um valor fixo, que pode ser de qualquer um dos tipos primitivos apresentados. Observe que uma constante do tipo ponto flutuante se diferencia de uma constante do tipo inteiro pela inserção do ponto decimal. Constantes do tipo caractere devem ser delimitadas por apóstrofos. O Exemplo 2.7 ilustra a aplicação dos tipos primitivos e de constantes em C.

Exemplo 2.7. *Tipos primitivos e constantes em C.*

```
#include <stdio.h>
int main() {
    printf("Inteiro..: %d\n",10);
    printf("Real.....: %f\n",10.);
    printf("Caractere: %c\n",'A');
}

Inteiro..: 10
Real.....: 10.000000
Caractere: A
```

2.6 Especificadores de Formato

A função `printf()` exige um *especificador de formato* para tipos primitivo incluído como parte do argumento. Os principais especificadores de formato são `%d` para `int`, `%f` para `float` e `%c` para `char`.

A constante cadeia de caracteres incluída na função `printf()`, delimitada por apóstrofos, deve conter um especificador de formato para cada constante de um tipo primitivo a ser exibida. Após a vírgula, é informado o valor propriamente dito, compatível com o especificador de formato definido.

Por padrão, valores do tipo ponto flutuante `float` e `double` são apresentados com 6 dígitos decimais.

Exercício 2.3. *Escreva um programa C para exibir na tela a saída a seguir usando tipos primitivos:*

```
10    10.000000    A
```

2.7 Variáveis de Memória

Uma variável consiste em um segmento de memória compreendendo um ou mais bytes contíguos. Cada variável deve possuir um nome, que é usado para referenciar o local na

memória onde encontram-se armazenadas as informações guardadas na variável.

Não é possível nomear duas variáveis distintas com o mesmo nome. Recomenda-se empregar nomes mneumônicos ao se declarar variáveis de memória. Entretanto, a definição de variáveis em C deve seguir um conjunto de regras para nomeação.

- Pode conter letras maiúsculas e minúsculas, dígitos numéricos e o caractere de sublinhar.
- Deve começar por letra ou caractere de sublinhar.
- Pode conter até 31 caracteres.
- Palavras reservadas da linguagem não podem ser usadas como nomes de variáveis.

Além disso, é importante observar que C diferencia caracteres alfabéticos maiúsculos e minúsculos. Portanto, é possível, embora não seja recomendável, definir variáveis de memória distintas com nomes `var1`, `VAR1` e `Var1`.

2.7.1 Declaração de Variáveis na Memória

Para que uma variável de memória possa ser utilizada em um programa C deve inicialmente ser *declarada*. A declaração requer, nessa ordem, a especificação do tipo de dado e do nome da variável, conforme as regras para nomeação de variáveis.

É uma boa prática de programação declarar todas as variáveis a serem usadas logo no início do programa, antes de quaisquer outros comandos. Essa prática facilita o entendimento do algoritmo e evita erros.

A *inicialização* de uma variável de memória é feita a partir do operador de atribuição representado pelo símbolo de igualdade (`=`).

O Exemplo 2.8 apresenta um programa fonte onde é declarada e inicializada uma variável de memória do tipo inteiro.

Exemplo 2.8. *Declaração e inicialização de uma variável de memória do tipo inteiro.*

```
#include <stdio.h>
int main() {
    int num;
    num = 10;
    printf("Conteudo da variavel num: %d\n", num);
}
```

```
Conteudo da variavel num: 10
```

A especificação de formato deve ser compatível com o tipo de dado. Caso ocorra inconsistência entre o formato e o tipo de dado, o compilador não apresentará erro mas o valor será exibido na saída padrão de forma incorreta, conforme ilustrado no Exemplo 2.9.

Exemplo 2.9. *Compatibilidade entre o tipo de dado e o especificador de formato.*

```
#include <stdio.h>
int main() {
    int num1 = 10;
    float num2 = 10;
    printf("%f %d\n",num1,num2);
}
```

```
0.000000 1076101120
```

2.8 Comando de Atribuição

Como é possível observar a partir dos Exemplos 2.8 e 2.9, o comando de atribuição (=) pode ser usado para alocar um determinado valor a uma variável de memória, desde que compatível com o tipo primitivo definido.

A variável que recebe o valor do comando de atribuição deve corresponder ao lado esquerdo do sinal de igualdade e o lado direito pode ser uma variável, uma constante ou uma expressão aritmética que contenha variáveis, constantes e operadores aritméticos.

Atenção: o comando de atribuição não pode ser entendido como "igualdade aritmética". A variável de memória que recebe a atribuição deve obrigatoriamente estar do lado esquerdo da expressão. O Exemplo 2.10 apresenta um programa C com uso incorreto do comando de atribuição. Nesse caso, o compilador acusará erro de compilação.

Exemplo 2.10. *Uso incorreto do comando de atribuição em C.*

```
#include <stdio.h>
int main() {
    int num;
    10 = num; // Uso incorreto do comando de atribuição.
    printf("Conteudo da variavel num: %d\n", num);
}
```

A expressão no lado direito de um comando de atribuição pode conter constantes e variáveis, conforme apresentado no Exemplo 2.11

Exemplo 2.11. *Exemplos de comandos de atribuição com constantes e variáveis.*

```
#include <stdio.h>
int main() {
    int num1;
    num1 = 10;
```

```
    int num2;
    num2 = num1;
    printf("num1: %d\n", num1);
    printf("num2: %d\n", num2);
}
```

```
num1: 10
num2: 10
```

Exercício 2.4. *Escreva um programa C para definir 3 variáveis, uma de cada um dos tipos primitivos apresentados (*`int`*,* `float` *e* `char`*), atribuir valores e exibir os valores atribuídos na tela.*

Exercício 2.5. *Ainda no programa do Exercício 2.4, inclua linhas de instrução para alterar os valores das variáveis definidas e reapresente os valores na tela.*

Observe que a declaração e a inicialização de uma variável de memória podem ser definidos em uma única linha de código, conforme apresentado no Exemplo 2.12.

Exemplo 2.12. *Declaração e atribuição de valor a variável em uma única instrução.*

```
#include <stdio.h>
int main() {
    int num = 10;
    printf("Conteudo da variavel num: %d\n", num);
}
```

```
Conteudo da variavel num: 10
```

2.9 Operadores Aritméticos

As operações aritméticas fundamentais empregam os seguintes símbolos em C: `+` para a adição, `-` para a subtração, `*` para a multiplicação e `/` para a divisão. Na linguagem C, um operador aritmético deve ser utilizado em conjunto com dois operandos, que podem ser constantes ou variáveis numéricas.

Por padrão, uma operação aritmética na qual os dois operandos forem do tipo inteiro é processada como uma operação inteira e o resultado é também um valor do tipo inteiro (Exemplo 2.13). Se pelo menos um dos operandos for do tipo ponto flutuante, a operação será processada em ponto flutuante e o resultado será um valor em ponto flutuante (Exemplo 2.14).

Exemplo 2.13. *Operadores aritméticos onde os operandos são valores numéricos do tipo inteiro.*

```
#include <stdio.h>
int main() {
    int num1 = 10;
    int num2 = 5;
    printf("Adicao.......: %d\n", 10 + 5);
    printf("Subtracao....: %d\n", num1 - 5);
    printf("Multiplicacao: %d\n", 5 * num2);
    printf("Divisao......: %d\n", num1 / num2);
}
```

```
Adicao.......: 15
Subtracao....: 5
Multiplicacao: 25
Divisao......: 2
```

Exemplo 2.14. *Operadores aritméticos onde os operandos são valores numéricos de ponto flutuante.*

```
#include <stdio.h>
int main() {
    float num1 = 10;
    float num2 = 5;
    printf("Adicao.......: %f\n", 10.0 + 5.);
    printf("Subtracao....: %f\n", num1 - 5.);
    printf("Multiplicacao: %f\n", 5.0 * num2);
    printf("Divisao......: %f\n", num1 / num2);
}
```

```
Adicao.......: 15.000000
Subtracao....: 5.000000
Multiplicacao: 25.000000
Divisao......: 2.000000
```

Observe que uma constante numérica com ponto decimal é interpretada pelo compilador C como sendo de ponto flutuante.

2.9.1 Operador Menos Unário

Como exceção, o operador para a subtração pode ser empregado com um único operando, conforme apresentado no Exemplo 2.15.

Exemplo 2.15. *Operador Menos Unário.*

```
#include <stdio.h>
int main() {
    int num = 10;
    num = -num;
    printf("%d\n", num);
}

-10
```

2.9.2 Operador Módulo

Além das quatro operações aritméticas básicas, C oferece o operador *módulo* ou resto da divisão inteira, cujo símbolo é `%`. Observe que a operação de resto da divisão inteira é válida apenas quando usada com variáveis ou constantes do tipo primitivo inteiro. Os Exemplos 2.16 e 2.17 ilustram a aplicação do operador módulo.

Exemplo 2.16. *Resto da divisão inteira com constantes do tipo inteiro.*

```
#include <stdio.h>
int main() {
    printf("Modulo %d\n", 10 % 3);
}

Modulo 1
```

Exemplo 2.17. *Resto da divisão inteira com variável e constante do tipo inteiro.*

```
#include <stdio.h>
int main() {
    int n = 10;
    printf("Modulo %d\n", n % 3);
}

Modulo 1
```

Observe a partir do Exemplo 2.18 que não há limite para o número de constantes ou variáveis em um único `printf()`.

Exemplo 2.18. *Função* `printf()` *com mais de um caractere de controle.*

```
#include <stdio.h>
int main() {
    int num1 = 10;
    int num2 = 3;
    int num3 = num1 + num2;
    printf("%d + %d = %d\n", num1, num2, num3);
}
```

```
10 + 3 = 13
```

2.9.3 Divisão Inteira

A operação de divisão é processada de acordo com o tipo dos operandos associados. Caso os dois operadores sejam do tipo inteiro, a divisão também será inteira e o resultado, consequentemente será inteiro (Exemplo 2.19).

Exemplo 2.19. *Divisão com operandos inteiros.*

```
#include <stdio.h>
int main() {
    int num1 = 10;
    int num2 = 3;
    int num3 = num1 / num2;
    printf("%d / %d = %d\n", num1, num2, num3);
}
```

```
10 / 3 = 3
```

Em uma operação de divisão com operandos do tipo inteiro, mesmo que o valor calculado seja atribuído a uma variável de ponto flutuante, a operação de divisão permanece inteira (Exemplo 2.20).

Exemplo 2.20. *Divisão com operandos inteiros e atribuição a variável de ponto flutuante.*

```
#include <stdio.h>
int main() {
    int num1 = 10;
    int num2 = 3;
    float num3 = num1 / num2;
    printf("%d / %d = %f\n", num1, num2, num3);
}
```

```
10 / 3 = 3.000000
```

Se pelo menos um dos operandos envolvidos em uma operação de divisão for do tipo ponto flutuante então a operação será real e o resultado será um número de ponto flutuante (Exemplo 2.21).

Exemplo 2.21. *Divisão com operandos de tipos distintos.*

```
#include <stdio.h>
int main() {
    int num1 = 10;
    float num2 = 3;
    float num3 = num1 / num2;
    printf("%d / %f = %f\n", num1, num2, num3);
}
```

```
10 / 3.000000 = 3.333333
```

2.9.4 Precedência de Operadores Aritméticos

A precedência dos operadores aritméticos é apresentada na Tabela 2.5.

Tabela 2.5: Precedência de operadores aritméticos

Operador	Descrição
( )	Parênteses
-	Menos unário
* / %	Multiplicação, divisão e módulo
+ -	Adição e subtração

O uso de parênteses permite a mudança na ordem de precedência dos operadores. O Exemplo 2.22 ilustra alguns casos de precedência entre operadores.

Exemplo 2.22. *Precedência em operadores.*

```
#include <stdio.h>
int main() {
    int num;
    num = 2 + 3 * 5;
    printf("%d \n", num);
    num = (2 + 3) * 5;
    printf("%d \n", num);
    num = -num * 2 + 50;
    printf("%d \n", num);;
```

```
}

17
25
0
```

2.9.5 Expressões Aritméticas no Corpo da Função `printf()`

Embora não seja uma boa prática de programação, é possível inserir expressões aritméticas no corpo da função `printf()`, conforme apresentado no Exemplo 2.23.

Exemplo 2.23. *Expressão matemática no corpo da função* `printf()`*.*

```
#include <stdio.h>
int main() {
    int num1 = 10;
    int num2 = 3;
    printf("%d + %d = %d\n", num1, num2, num1 + num2);
}

10 + 3 = 13
```

Exercício 2.6. *Escreva um programa C para definir e atribuir valor a 2 variáveis de memória do tipo* `int`*. Exibir na tela o resultado das cinco operações aritméticas com as variáveis definidas.*

Exercício 2.7. *Escreva um programa C para definir e atribuir valor a 2 variáveis de memória do tipo* `float`*. Exibir na tela o resultado das quatro operações aritméticas básicas com as variáveis definidas.*

Exercício 2.8. *Escreva um programa C para definir e atribuir valor a 2 variáveis de memória, uma do tipo* `int` *e uma do* `float`*. Exibir na tela o resultado das quatro operações aritméticas básicas com as variáveis definidas.*

2.10 Controle e Formatação de Saída

2.10.1 Controle do Número de Dígitos Decimais

Por padrão, a saída de valores de ponto flutuante na saída padrão (vídeo) a partir da função `printf()` é exibida com 6 dígitos decimais.

O modificador de precisão permite controlar o número de dígitos decimais, conforme apresentado no Exemplo 2.24.

Exemplo 2.24. *Controle do número de dígitos decimais com o modificador de precisão.*

```
#include <stdio.h>
int main() {
    float num1 = 10;
    float num2 = 3;
    float num3 = num1 / num2;
    printf("%.1f / %.1f = %.1f\n", num1, num2, num3);
}
```

```
10.0 / 3.0 = 3.3
```

2.10.2 Controle do Comprimento do Campo de Saída

Além do número de dígitos decimais, o especificador de formato usado na função `printf()` permite controlar o comprimento total do campo de saída para todos os tipos de dados numéricos, conforme apresentado no Exemplo 2.25.

Exemplo 2.25. *Controle do comprimento do campo de saída na função* `printf()`.

```
#include <stdio.h>
int main() {
    float num1 = 10;
    int num2 = 3;
    float num3 = num1 / num2;
    printf("%5.1f / %4d = %5.1f\n", num1, num2, num3);
}
```

```
 10.0 /    3 =   3.3
```

Observe que se o especificador de formato definir um comprimento de campo menor que o mínimo necessário para a correta exibição do valor numérico a ser exibido, o compilador C irá desconsiderar o tamanho do campo especificado (Exemplo 2.26).

Exemplo 2.26. *Controle de comprimento do campo menor que o necessário para exibição do valor numérico.*

```
#include <stdio.h>
int main() {
    float num1 = 100;
    int num2 = 3;
    float num3 = num1 / num2;
    printf("%3.1f / %1d = %2.1f\n", num1, num2, num3);
}
```

```
10.0 / 3 = 33.3
```

2.10.3 Exibição de Zeros Não Significativos

O controle do comprimento de campo permite também a exibição dos zeros não significativos na saída de valores numéricos a partir da função `printf()`. Esse recurso é obtido ao antepor-se um dígito 0 entre o especificador de formato % e o controle do comprimento de campo, conforme apresentado no Exemplo 2.27.

Exemplo 2.27. *Exibição de zeros não significativos a partir do controle do comprimento do campo.*

```
#include <stdio.h>
int main() {
    float num1 = 3.1416;
    int num2 = 323;
    printf("%08.4f\n", num1);
    printf("%06d\n",num2);
}

003.1416
000323
```

2.10.4 Alinhamento à Esquerda

Por padrão, valores numéricos são alinhados à direita. É possível, entretanto, modificar a exibição dos valores numéricos alterando-se o alinhamento para a esquerda, conforme apresentado no Exemplo 2.28.

Exemplo 2.28. *Alinhamento de valores numéricos à esquerda.*

```
#include <stdio.h>
int main() {
    float num1 = 100.0;
    int num2 = 15;
    printf("%7.1f\n",num1);
    printf("%-7.1f\n",num1);
    printf("%5d\n",num2);
    printf("%-5d\n",num2);
}

  100.0
100.0
   15
15
```

2.11 Definição de Constantes na Memória

Para se definir uma constante na memória e alocar um valor que não pode ser alterado posteriormente durante a execução do programa, utiliza-se a palavra reservada `const` (Exemplo 2.29). Observe que se o programador tentar alterar o valor de uma constante previamente definida, o compilador acusará um erro.

Exemplo 2.29. *Declaração de constantes na memória.*

```
#include <stdio.h>
int main() {
    int raio = 2;
    const float pi = 3.1415926;
    float area;
    area = pi * raio * raio;
    printf("Area do circulo de raio %d: %.2f\n",raio, area);
}
```

```
Area do circulo de raio 2: 12.57
```

2.12 Entrada de Dados na Entrada Padrão

A função `scanf()` é a forma mais usual de fazer a entrada de dados a partir da entrada padrão (teclado). Para seu correto funcionamento é necessário especificar a biblioteca `<stdio.h>`.

A função `scanf()` recebe como argumento uma constante cadeia de caracteres com um ou mais especificadores de formato seguido por um ou mais endereços de memória de variáveis previamente definidas no código fonte do programa C. O Exemplo 2.30 apresenta um caso de uso simples da função `scanf()`.

Exemplo 2.30. *Função para entrada de dados `scanf()`.*

```
#include <stdio.h>
int main() {
    int num;
    printf("Digite um numero inteiro: ");
    scanf("%d", &num);
    printf("Voce digitou o numero %d.\n", num);
}
```

```
Digite um numero inteiro: 6
Voce digitou o numero 6.
```

A variável de memória que recebe o valor digitado pelo usuário deve ser precedida pelo operador *endereço de memória*, representado pelo símbolo `&`. A omissão do operador `&` não causa erro de compilação mas pode determinar a *abortagem* da execução do programa.

2.12.1 Especificadores de Formato para `scanf()`

A função `scanf()` exige a especificação do tipo de dado, com os mesmos especificadores de formato usados na função `printf()`: `%d` para tipos inteiros, `%f` para tipos de ponto flutuante e `%c` para caracteres da tabela ASCII.

A função `scanf()` permite a entrada de dados para mais de uma variável de memória. Nesse caso, para cada entrada de dados, deve corresponder um especificador de formato e uma variável de memória precedida pelo operador de endereço (Exemplo 2.31).

Exemplo 2.31. *Entrada de múltiplos valores com* `scanf()`.

```
#include <stdio.h>
int main() {
    int num1;
    int num2;
    printf("Digite dois numeros inteiros: ");
    scanf("%d%d", &num1, &num2);
    printf("Voce digitou os numeros %d e %d.\n", num1, num2);
}
```

```
Digite dois numeros inteiros: 4 7
Voce digitou os numeros 4 e 7.
```

A função `scanf()` também permite o uso do especificador de tamanho para delimitar o número de caracteres a serem alocados para uma variável. O Exemplo 2.32 apresenta um caso de uso desse recurso.

Exemplo 2.32. *Controle do número de caracteres aceitos para uma variável a partir do especificador de tamanho na função* `scanf()`.

```
#include <stdio.h>
int main() {
    int dia;
    int mes;
    int ano;
    printf("Digite a data atual no formato DDMMAAAA: ");
    scanf("%2d%2d%4d", &dia, &mes, &ano);
    printf("Data: %02d/%02d/%d\n", dia, mes, ano);
}
```

```
Digite a data atual no formato DDMMAAAA: 09082010
Data: 09/08/2010
```

Também é possível inserir outros caracteres no especificador de formato de uma chamada à função `scanf()`. Nesse caso, o respectivo caractere deve ser informado na entrada de dados do usuário na posição solicitada. O Exemplo 2.33 ilustra o recurso.

Exemplo 2.33. *Outros caracteres na entrada de dados através da função* `scanf()`.

```
#include <stdio.h>
int main() {
    int hora;
    int min;
    printf("Digite a hora atual no formato HH:MM ");
    scanf("%2d:%2d", &hora, &min);
    printf("%2d horas e %2d minutos.\n", hora, min);
}
```

```
Digite a hora atual no formato HH:MM 14:47
14 horas e 47 minutos.
```

Exercício 2.9. *Escreva um programa C que solicite ao usuário a entrada de* 3 *números inteiros quaisquer. O programa deve calcular e apresentar na tela a soma dos números informados pelo usuário.*

Exercício 2.10. *Escreva um programa C que solicite ao usuário a entrada de* 3 *números reais quaisquer. O programa deve calcular e apresentar na tela a média aritmética dos números.*

Exercício 2.11. *Escreva um programa C que solicite ao usuário o salário de um funcionário e que calcule e mostre na tela o novo salário, sabendo-se que houve um reajuste de* 12.5%.

Exercício 2.12. *Escreva um programa C que calcule e apresente na tela a área de um triângulo retângulo. Os valores da base e altura devem ser fornecidos pelo usuário em tempo de execução.*

Exercício 2.13. *Escreva um programa C que solicite ao usuário a digitação do seu ano de nascimento. O programa deve calcular e apresentar na tela: (a) a idade do usuário e (b) em qual ano terá* 65 *anos.*

Exercício 2.14. *Escreva um programa C que calcule e apresente na tela a média ponderada de duas notas fornecidas pelo usuário em tempo de execução. Considere peso* 2 *para a primeira prova e* 3 *para a segunda prova.*

Exercício 2.15. *Escreva um programa C que solicite ao usuário o valor da hora atual no formato HHMMSS. O programa deve calcular e mostrar na tela o total de segundos transcorridos desde o início do dia.*

Exercício 2.16. *Escreva um programa C que solicite ao usuário a digitação de um número real. O programa deve exibir na tela o valor inteiro mais próximo do número real informado. Por exemplo, se o número digitado for* 3.8*, o valor inteiro mais próximo é* 4*.*

Exercício 2.17. *Escrever um programa C que solicite ao usuário a digitação de um número inteiro entre* 100 *e* 999*. Calcular a soma dos algarismos do número digitado. Por exemplo, se o número for* 234*, a soma deve ser* 9*.*

2.13 Outras Funções para Entrada de Dados

A função `getchar()` é a forma mais simples de entrada de dados em C. Esta função permite a entrada de um único caractere da tabela ASCII.

Exemplo 2.34. *Função* `getchar()` *para entrada de um único caractere na entrada padrão.*

```
#include <stdio.h>
int main() {
    char ch;
    ch = getchar();
    putchar(ch);
}
```

As funções `getche()` e `getch()` pertencem à biblioteca `conio.h` e possuem objetivo semelhante à função `getchar()`. A diferença é que o usuário não precisa teclar `Enter` para que a execução continue. Além disso, `getch()` não ecoa o caractere digitado na saída padrão.

```
#include <stdio.h>
#include <conio.h>
int main() {
    char ch;
    ch = getche();
    putchar(ch);
    ch = getch();
    putchar(ch);
}
```

```
AAB
```

2.14 Constantes e Funções Matemáticas

O arquivo de cabeçalho `math.h` inclui as declarações necessárias para diversas funções matemáticas disponíveis nos compiladores C, incluindo-se funções para calcular potência e raiz quadrada, funções trigonométricas para cálculos que envolvem seno, cosseno e tangente, além de constantes para números irracionais como, por exemplo, π e $\sqrt{2}$.

Uma chamada a uma função matemática pode ser ser associada a um comando de atribuição ou utilizada diretamente em um `printf()` para exibição na saída padrão do sistema. O Exemplo 2.35 apresenta a chamada a uma função matemática em um comando de atribuição.

Exemplo 2.35. *Cálculo do valor absoluto de um número inteiro fornecido pelo usuário.*

```
#include <stdio.h>
#include <math.h>
int main() {
    int num;
    int valorabs;
    printf("Digite um numero inteiro qualquer: ");
    scanf("%d", &num);
    valorabs = abs(num);
    printf("Valor absoluto de %d: %d.\n", num, valorabs);
}

Digite um numero inteiro qualquer: -5
Valor absoluto de -5: 5.
```

O Exemplo 2.36 apresenta a chamada a uma função matemática em um `printf()`.

Exemplo 2.36. *Cálculo do valor absoluto de um número inteiro fornecido pelo usuário.*

```
#include <stdio.h>
#include <math.h>
int main() {
    int num;
    printf("Digite um numero inteiro qualquer: ");
    scanf("%d", &num);
    printf("Valor absoluto de %d: %d.\n", num, abs(num));
}

Digite um numero inteiro qualquer: -8
Valor absoluto de -8: 8.
```

2.14.1 Constantes Matemáticas

A Tabela 2.6 apresenta diversas constantes matemáticas definidas na biblioteca `math.h`. Todas as constantes são definidas como números de ponto flutuante do tipo `double`.

Tabela 2.6: Constantes matemáticas definidas em `math.h`

Símbolo	**Descrição**	**Constante**	**Valor**
e	Número de Euler	M_E	2.7182818284590452354
$\log_2 e$	Logaritmo de e na base 2	M_LOG2E	1.4426950408889634074
$\log_{10} e$	Logaritmo de e na base 10	M_LOG10E	0.43429448190325182765
π	Pi	M_PI	3.14159265358979323846
$\pi/2$	Meio Pi	M_PI_2	1.57079632679489661923
$\pi/4$	Quarto de Pi	M_PI_4	0.78539816339744830962
$\sqrt{2}$	Raiz quadrada de 2	M_SQRT2	1.41421356237309504880

2.14.2 Funções Matemáticas

As funções matemáticas da biblioteca `math.h` retornam valores numéricos do tipo `double`.

Funções Trigonométricas

As funções trigonométricas recebem como argumento o ângulo em radiano.

- `sin(double)` retorna o valor do seno.
- `cos(double)` retorna o valor do cosseno.
- `tan(double)` retorna o valor da tangente.

Funções Logaritmicas

- `log2(double)` retorna o valor do logaritmo na base 2.
- `log10(double)` retorna o valor do logaritmo na base 10.
- `log(double)` retorna o valor do logaritmo neperiano (ou natural), na base e.

Potências

- `pow(double, double)` retorna o valor da base elevada ao expoente. Recebe dois argumentos do tipo `double`, o primeiro é a base e o segundo o expoente.
- `sqrt(double)` retorna o valor da raiz quadrada.

Arredondamento

- `ceil(double)` retorna o primeiro `double` sem casas decimais acima do valor fornecido como argumento. Exemplo: `ceil(45.98561)` retorna 46.
- `floor(double)` retorna o primeiro `double` sem casas decimais abaixo do valor fornecido como argumento. Exemplo: `floor(45.98561)` retorna 45.

2.14.3 Valor Absoluto

- `abs(int)` retorna o valor absoluto de um número inteiro fornecido como argumento. O retorno é um número inteiro.
- `fabs(double)` retorna o valor absoluto de um `double` fornecido como argumento. O retorno é um `double`.

Exemplo 2.37. *Emprego de constante e de função matemática da biblioteca* `math.h` *para o cálculo do seno de um ângulo cujo ângulo é fornecido pelo usuário em graus.*

```
#include <stdio.h>
#include <math.h>
int main() {
    int angulo;
    float radiano;
    printf("Digite o angulo em graus: ");
    scanf("%d", &angulo);
    radiano = angulo * M_PI / 180;
    printf("Seno de %d graus: %f\n", angulo,sin(radiano));
}
```

```
Digite o angulo em graus: 30
Seno de 30 graus: 0.50000
```

Exercício 2.18. *Escreva o programa C para o calculo da area de um circulo utilizando a funçao matemática* `pow()`.

Exercício 2.19. *Escreva um programa C que solicite ao usuario em tempo de execução o raio R e que calcule e mostre na tela a área A e o volume V de uma esfera dados por*

$$A = 4\pi R^2 \quad e \quad V = \frac{4\pi R^3}{3}$$

Exercício 2.20. *Escreva um programa C que que solicite ao usuario em tempo de execuçao as coordenadas x e y de dois pontos quaisquer do plano cartesiano e que calcule e mostre na tela a distancia entre os pontos.*

Exercício 2.21. *Escreva um programa C que que solicite ao usuario em tempo de execuçao o valor de um angulo em graus e que calcule e mostre na tela os valores da cosecante e da cotangente. Lembre-se que 180° equivale a π radianos.*

Exercício 2.22. *A função exponencial é usada para calcular a desintegração do carbono* 14 *através da equação* $y(t) = y_0 e^{-0.00012t}$*, onde* y_0 *é a massa inicial da substância medida em gramas e* t *é o tempo transcorrido, medido em anos. Escreva um programa C que solicite ao usuário a massa inicial de uma amostra de carbono* 14 *e que apresente na saída padrão a massa após* 10, 100 *e* 1000 *anos.*

Capítulo 3

Estruturas de Seleção

Quem pouco pensa, muito erra. – Leonardo da Vinci

Todos os programas apresentados até o momento são executados exatamente na ordem em que as instruções são escritas, do início até o fim. A maioria dos algoritmos a serem implementados em uma linguagem de programação não seguem essa sequência. A razão para isso é que muitas vezes e necessário tomar uma decisão e, a partir daí, continuar a execução a partir de caminhos alternativos.

3.1 Operadores Relacionais

Para tomar uma decisão é necessário um mecanismo para se comparar valores. A linguagem C oferece um conjunto de *operadores relacionais* com esse objetivo, conforme apresentado na Tabela 3.1.

Tabela 3.1: Operadores Relacionais

Símbolo	**Descrição**
$<$	Menor que
$<=$	Menor ou igual a
$>$	Maior que
$>=$	Maior ou igual a
$==$	Igual a
$!=$	Diferente de

3.1.1 Expressões Envolvendo Operadores Relacionais

A partir dos operadores relacionais apresentados na Tabela 3.1 é possível elaborar *expressões lógicas* ou booleanas. Tais expressões podem resultar em apenas um de dois valores possíveis: verdadeiro ou falso.

Na linguagem C, um valor logicamente verdadeiro é representado pelo número 1 e um valor logicamente falso é representado pelo número 0.

O Exemplo 3.1 ilustra a aplicação dos operadores relacionais em expressões lógicas.

Exemplo 3.1. *Operadores relacionais em expressões lógicas.*

```
#include <stdio.h>
int main() {
    printf("%d\n",5>4);
    printf("%d\n",1==2);
    printf("%d",'A'!='a');
}

1
0
1
```

3.2 Estruturas de Seleção da Linguagem C

As *estruturas de seleção* permitem que o programa possa decidir o caminho a seguir durante a execução. Por exemplo, dependendo do valor de uma variável, uma estrutura de decisão pode executar um conjunto de comandos ou não.

As estruturas de seleção utilizam os operadores relacionais para compor expressões lógicas e decidir qual o caminho que a execução de um programa deve seguir dependendo do resultado da expressão, ou seja, verdadeiro ou falso.

3.2.1 Estrutura de Seleção `if-else`

O comando `if-else` é a estrutura de seleção mais utilizada na maioria das linguagens de programação. Basicamente, é utilizada quando torna-se necessário escolher entre dois caminhos possíveis para a execução do programa.

A forma geral do comando `if-else` é dada por

```
if (expressão lógica)
   bloco de comandos 1;
else
```

```
   bloco de comandos 2;
```

Atente para o uso de parênteses para delimitar a expressão lógica a ser avaliada na estrutura de seleção `if-else`.

O comando `if-else` avalia o valor da expressão lógica e, caso o resultado seja verdadeiro, o *bloco de comandos 1* é executado. Caso o resultado seja falso, o *bloco de comandos 2*, associado à cláusula `else`, é executado. Entende-se por *bloco de comandos* um conjunto formado por um ou mais comandos da linguagem C.

Observe que em uma dada execução do programa é executado apenas o *bloco de comandos 1*, associado ao resultado verdadeiro para a expressão lógica avaliada, ou o *bloco de comandos 2*, associado ao resultado falso para a expressão lógica avaliada.

O Exemplo3.2 ilustra o emprego da estrutura de seleção `if-else` usando expressões lógicas para definir o caminho a ser seguido pela execução do programa.

Exemplo 3.2. *Programa C que solicita ao usuário a digitação de um número inteiro qualquer e verifica se é igual ou diferente de* 10*.*

```
#include <stdio.h>
int main() {
   int num;
   printf("Digite um numero inteiro: ");
   scanf("%d",&num);
   if(num==10)
      printf("%d igual a 10",num);
   else
      printf("%d diferente de 10",num);
}
```

```
Digite um numero inteiro: 11
O numero 11 eh diferente de 10.
```

Exercício 3.1. *Escreva um programa C que solicite ao usuário a digitação de um número inteiro N em tempo de execução. O programa deve calcular e apresentar na tela $\sqrt{N}$ caso $N \geq 0$ ou N^2 caso $N < 0$.*

Exercício 3.2. *Escreva um programa C que solicite ao usuário a digitação de dois números inteiros em tempo de execução. O programa deve verificar se os números digitados são iguais ou diferentes.*

Exercício 3.3. *Escreva um programa C que solicite ao usuário a digitação de dois números quaisquer em tempo de execução. O programa deve apresentar na tela o maior dos dois números informados.*

Exercício 3.4. *Escreva um programa C que solicite ao usuário a digitação das quatro notas de um aluno e que calcule e apresente na tela a média aritmética das notas e uma*

mensagem de aprovado ou reprovado, considerando para aprovação a média maior ou igual a 70. *Considere as notas como sendo números inteiros no intervalo* $[0, 100]$.

Uma vez que para a linguagem C todo número diferente de 0 é considerado logicamente verdadeiro e o 0 é considerado logicamente falso, o Exemplo 3.3 apresenta um comando de seleção `if-else` que baseia-se nessa característica para definir se um número informado pelo usuário é nulo ou não.

Exemplo 3.3. *Programa C que solicita ao usuário a digitação de um número inteiro qualquer e verifica se é igual ou diferente de* 0.

```
#include <stdio.h>
int main() {
   int num;
   printf("Digite um numero inteiro: ");
   scanf("%d",&num);
   if(num) // num é verdadeiro, isto é, num é diferente de 0
      printf("Numero %d -> diferente de 0",num);
   else
      printf("Numero %d -> igual a 0",num);
}
```

```
Digite um numero inteiro: 3
Numero 3 -> diferente de 0.
```

Exercício 3.5. *Escreva um programa C que solicite ao usuário a digitação de um número inteiro e positivo qualquer e que verifique se o valor digitado é par ou ímpar.*

Exercício 3.6. *Escreva um programa C que solicite ao usuário a digitação de um número inteiro e positivo em tempo de execução. O programa deve informar na tela se o número e ou não divisível por* 5.

Estrutura de Seleção `if` Sem Cláusula `else`

A cláusula `else` não é obrigatória. O Exemplo 3.4 apresenta um programa C que emprega a estrutura de seleção `if` sem a cláusula `else`.

Exemplo 3.4. *Programa C que solicita ao usuário a digitação de um número inteiro qualquer e que calcula o valor absoluto.*

```
#include <stdio.h>
int main() {
   int num;
   printf("Inteiro qualquer: ");
```

```
   scanf("%d",&num);
   if(num < 0)
      num = -num;
   printf("Valor absoluto: %d\n",num);
}

Digite um numero inteiro: -1
Valor absoluto: 1
```

Exercício 3.7. *Escreva um programa C que solicite ao usuario a digitaçao de um número real N em tempo de execuçao. O programa deve calcular e exibir o logaritmo neperiano* $(\log N)$ *caso* $N \geq 0$ *ou o próprio número N, caso* $N < 0$.

Uso de Blocos de Código em Estruturas `if-else`

Retornando à forma geral do comando `if-else`, dada por

```
if (expressão lógica)
   bloco de comandos 1;
else
   bloco de comandos 2;
```

é importante observar que quando houver mais de um comando associado ao *bloco de comandos 1* ou ao *bloco de comandos 2*, então torna-se necessário delimitar por chaves de abertura e de fechamento o conjunto de comandos.

A esse conjunto de instruções C delimitadas por chaves de abertura e fechamento dá-se o nome de *bloco de código*. Como poderá ser visto ao longo do curso, em muitas situações um bloco de código é exigido por diversos comandos da linguagem C quando há mais de uma instrução a ser executada.

Embora não seja necessário, é permitido delimitar-se por chaves de abertura de fechamento um bloco de código que contenha apenas um único comando.

O Exemplo 3.5 ilustra a aplicação de blocos de código em um comando `if-else`. Observe que o bloco de código associado à cláusula `else` possui apenas um único comando. Nesse caso, a delimitação do bloco por chaves é opcional.

Atenção: o uso incorreto de chaves de abertura e de fechamento é fonte de erros ao compilar um programa C. Procure sempre verificar se para cada chave de abertura corresponde uma chave de fechamento.

Exemplo 3.5. *Programa C que solicita ao usuário a digitação de um número inteiro e positivo e que calcula e exibe a raiz quadrada e a raiz cúbica. Uma mensagem de erro é apresentada caso o número digitado não seja positivo.*

```
#include <stdio.h>
#include <math.h>
int main() {
   int num;
   printf("Digite um numero inteiro e positivo: ");
   scanf("%d", &num);
   if(num>0) {
      printf("Raiz quadrada: %f\n",sqrt(num));
      printf("Raiz cubica..: %f",pow(num,0.333));
   } else {
      printf("Erro. Numero nao positivo.");
   }
}
```

```
Digite um numero inteiro e positivo: 9
Raiz quadrada: 3
Raiz cubica..: 2.078561
```

Exercício 3.8. *Segundo uma tabela médica, o peso ideal está relacionado com a altura e o sexo. Fazer um programa C que receba, em tempo de execução, a altura* H *e o sexo de uma pessoa e que calcule e imprima o seu peso ideal, utilizando as seguintes fórmulas: (a) para homens:* $72.7 \times H - 58$ *e (b) para mulheres* $62.1 \times H - 44.7$.

Estruturas `if-else` Aninhadas

Para escrever muitos algoritmos torna-se necessário empregar estruturas de seleção aninhadas, ou seja, uma dentro da outra. Uma vez que cada estrutura `if-else` permite apenas selecionar entre dois valores lógicos (verdadeiro ou falso), a solução para algoritmos que envolvam mais de duas condições possíveis recai no emprego de estruturas aninhadas conforme mostrado no Exemplo 3.6.

Exemplo 3.6. *Programa C que solicita ao usuário a digitação da idade I de uma pessoa e que verifica se é menor* ($I < 18$)*, adulto ou idoso* ($I \geq 65$).

```
#include <stdio.h>
int main() {
   int idade;
   printf("Digite a idade: ");
   scanf("%d", &idade);
   if(idade<18)
      printf("Menor");
   else if(idade<65)
      printf("Adulto");
   else
```

```
      printf("Idoso");
}

Digite a idade: 30
Adulto
```

Exercício 3.9. *Escreva um programa C que receba dois números e execute as operações listadas a seguir, de acordo com a escolha do usuário. Se a opção digitada for inválida, mostrar uma mensagem de erro. Observe que não é possível calcular a divisão quando o divisor for nulo.*

1. *média entre os números digitados;*
2. *diferença do maior pelo menor;*
3. *produto dos números digitados;*
4. *divisão do maior pelo menor.*

Exercício 3.10. *Escreva um programa C que solicite ao usuário a digitação de duas notas de um aluno (no intervalo* $[0, 100]$*) e que calcule e apresente na tela a média aritmética das notas e a mensagem segundo a regra:*

- *média menor que 40 – Reprovado*
- *média igual ou maior que 40 e menor que 70 – Exame final*
- *média igual oo maior que 70 – Aprovado*

3.3 Operadores Lógicos

Muitas vezes as expressões relacionais a serem avaliadas necessitam combinar mais de uma condição utilizando-se os operadores relacionais apresentados anteriormente na Tabela 3.1.

Por exemplo, pode ser necessário executar uma ação se um determinado número está um intervalo específico $[a, b]$, ou seja, deve ser não menor que a **e** não maior que b. Em outra situação, uma ação deve ser executada somente se um caractere é igual a `'A'` **ou** igual a `'a'`.

Para compor expressões relacionais que sejam capazes de lidar adequadamente com essas situações podem ser utilizados os operadores lógicos, apresentados na Tabela 3.2.

3.3.1 Operador Lógico &&

Uma expressão relacional composta que empregue um operador E lógico (`&&`) será considerada verdadeira se, e somente se, todos os componentes forem logicamente verdadeiros. Se apenas um dos componentes for logicamente falsa então a expressão relacional será avaliada como falsa. O Exemplo 3.7 ilustra a aplicação do operador lógico `&&`.

Exemplo 3.7. *Programa C para exemplificar o emprego do operador lógico* E.

Tabela 3.2: Operadores Lógicos

Símbolo	Descrição
&&	E lógico
||	OU lógico
!	NÃO lógico

```
#include <stdio.h>
int main() {
   int num1 = 2;
   float num2 = 3.5;
   char ch = 'R';
   if(num1<=2 && num2>3.4 && ch=='T')
      printf("Dados validos");
   else
      printf("Dados invalidos");
}

Dados invalidos.
```

3.3.2 Operador Lógico ||

Uma expressão relacional composta que empregue um operador *OU* lógico (||) será considerada verdadeira se, e somente se, pelo menos um dos componentes for logicamente verdadeiro. Somente se todos os componentes forem logicamente falsos é que a expressão relacional será avaliada como falsa. O Exemplo 3.8 ilustra a aplicação do operador lógico ||.

Exemplo 3.8. *Programa C para exemplificar o emprego do operador lógico* OU.

```
#include <stdio.h>
int main() {
   int num1 = 2;
   float num2 = 3.5;
   char ch = 'R';
   if(num1<=2 || num2>3.4 || ch=='T')
      printf("Dados validos");
   else
      printf("Dados invalidos");
}

Dados validos.
```

3.3.3 Operador Lógico !

O operador lógico *NÃO*, representado pelo símbolo `!`, é um operador unário. Se objetivo é reverter o valor lógico de uma expressão relacional. Assim, se uma dada expressão for logicamente verdadeira, o operador `!` torna a expressão logicamente falsa e vice-versa. O Exemplo 3.9 ilustra o uso do operador lógico `!`.

Exemplo 3.9. *Programa C para exemplificar o emprego do operador lógico* NÃO.

```
#include <stdio.h>
int main() {
   int num;
   printf("Digite um numero inteiro: ");
   scanf("%d",&num);
   if(!num)
      printf("Nulo");
   else
      printf("Nao nulo");
}
```

```
Digite um numero inteiro: 3
Nao Nulo
```

3.3.4 Expressões Relacionais e Operadores Lógicos

O emprego de operadores lógicos permite a composição de expressões relacionais complexos, que podem empregar diversos operadores relacionais, conforme ilustrado no Exemplo 3.10.

Exemplo 3.10. *Escrever um programa C que solicite ao usuário a digitação dos lados de um triângulo e que exiba na tela a classificação do triângulo quanto ao tamanho dos lados (equilátero, isósceles ou escaleno). Um triângulo é equilátero se possui os três lados iguais, isósceles se possui* 2 *lados iguais e escaleno se possui os* 3 *lados de tamanhos diferentes.*

```
#include <stdio.h>
int main() {
   int a;
   int b;
   int c;
   printf("Digite os lados de um triangulo: ");
   scanf("%d%d%d",&a,&b,&c);
   if(a == b && a == c)
      printf("Triangulo equilatero");
   else if(a==b || a==c || b==c)
```

```
        printf("Triangulo isosceles");
    else
        printf("Triangulo escaleno");
}

Digite os lados de um triangulo: 2 5 3
Triangulo escaleno.
```

Atenção: Considerando-se o Exemplo 3.10 é importante observar que a linguagem C não permite a composição de expressões relacionais do tipo: `if(a==b && c)` ou `if(a==b || ==c || b==c)`. Em outros termos, uma expressão relacional deve ser formada por dois operandos e um operador relacional e a composição de duas ou mais expressões relacionais deve ser formada com operadores lógicos.

Exercício 3.11. *Escreva um programa C que solicite ao usuário o código de origem de um produto e que mostre o a procedência, de acordo com a Tabela 3.3.*

Cód. Origem	**Procedência**
1	Sul
2	Norte
3	Leste
4	Oeste
5 ou 6	Nordeste
7 ou 8 ou 9	Sudeste
10 a 20	Centro-Oeste
21 a 30	Noroeste

Tabela 3.3: Procedência de Produtos

Categoria	**Idade**
Infantil	5 a 7
Juvenil	8 a 10
Adolescente	11 a 15
Adulto	16 a 30
Senior	Acima de 30

Tabela 3.4: Categorias de Nadadores

3.3.5 Problemas de Múltiplas Opções

Nos casos de problemas de múltipla opções, onde é necessário o emprego de duas ou mais estruturas `if else`, a sintaxe pode ser ligeiramente alterada conforme apresentado no Exemplo 3.11. Este estilo de codificação `else if` é mais recomendado nessas situações porque facilita a legibilidade.

Exemplo 3.11. *Escrever um programa C que solicite ao usuário a digitação dos lados de um triângulo e que exiba na tela a classificação do triângulo quanto ao tamanho dos lados (equilátero, isósceles ou escaleno). Um triângulo é equilátero se possui os três lados iguais, isósceles se possui* 2 *lados iguais e escaleno se possui os* 3 *lados de tamanhos diferentes.*

```
#include <stdio.h>
int main() {
   int a;
   int b;
   int c;
   printf("Digite os lados de um triangulo: ");
   scanf("%d%d%d",&a,&b,&c);
   if(a==b && a==c)
       printf("Triangulo equilatero");
   else if(a==b || a==c || b==c)
       printf("Triangulo isosceles");
   else
       printf("Triangulo escaleno");
}
```

```
Digite os lados de um triangulo: 2 5 3
Triangulo escaleno
```

Exercício 3.12. *Escreva um programa C que solicite ao usuário a idade de um nadador e mostre na tela a sua categoria, usando as regras apresentadas na Tabela 3.4.*

Exercício 3.13. *Três valores numéricos podem ser os lados de um triângulo se cada valor for menor que a soma dos outros valores. Escreva um programa C que solicite ao usuário a digitação de três valores inteiros e verifique se podem formar um triângulo.*

Exercício 3.14. *Escrever um programa C que solicite ao usuário a digitação dos coeficientes de uma equação do segundo grau ($ax^2 + bx + c = 0$) e que calcule e retorne as raízes reais da equação. Observe que uma equação do 2^o grau pode ter duas, uma ou nenhuma raiz real, dependendo do valor de Δ, onde $\Delta = b^2 - 4ac$.*

Exercício 3.15. *Capicua ou palíndromo é definido como um número que lido da direita para a esquerda ou da esquerda para a direita é idêntico. Escreva um programa C que solicite ao usuário a digitação de um número inteiro e positivo no intervalo* $[1000, 9999]$ *e que verifique se o número é palíndromo.*

Exercício 3.16. *O Teorema de Pitágoras é provavelmente o mais célebre dos teoremas da Matemática. Enunciado pela primeira vez por filósofos gregos chamados de pitagóricos, estabelece uma relação simples entre o comprimento dos lados de um triângulo retângulo: "o quadrado da hipotenusa é igual à soma dos quadrados dos catetos". Escreva um programa C para verificar se os valores fornecidos pelo usuário em tempo de execução para as medidas dos lados formam ou não um triângulo retângulo.*

3.4 Operador Condicional Ternário

A linguagem C fornece um operador condicional ternário, muito semelhante a um bloco `if-else`. A sintaxe do comando é `condição ? expressão1 : expressão2`.

O primeiro operando é uma condição a ser logicamente avaliada, o segundo operando é a expressão a ser executada caso a condição avaliada seja logicamente verdadeira. O terceiro operando é a expressão a ser executada caso a condição avaliada seja logicamente falsa.

Os Exemplos 3.12 e 3.13 apresentam alguns casos de uso do operador condicional ternário.

Exemplo 3.12. *Operador condicional ternário associado a um comando de atribuição.*

```
#include <stdio.h>
int main() {
   int num;
   int res;
   printf("Digite um numero inteiro: ");
   scanf("%d",&num);
   res = (num>=0) ? 1 : 0;
   printf("%d",res);
}

Digite um numero inteiro: 3
1
```

Exemplo 3.13. *Operador condicional ternário associado à saída padrão.*

```
#include <stdio.h>
int main() {
   int num;
   printf("Quantos animais de estimacao voce possui? ");
   scanf("%d", &num);
   printf("Voce possui %d anima%s de estimacao",
      num, (num == 1) ? "l" : "is");
}

Quantos animais de estimacao voce possui? 1
Voce possui 1 animal de estimacao
```

3.5 Precedência de Operadores

Agora que já foram apresentados diversos operadores aritméticos, relacionais e lógicos, torna-se importante observar a ordem de precedência de execução entre eles.

Suponha que seja necessário escrever um programa para selecionar candidatos a uma vaga de emprego com os seguintes requisitos: idade acima de 25 anos e formação em Engenharia ou Informática. O Exemplo 3.14 tenta resolver o problema.

Exemplo 3.14. *Precedência entre operadores da linguagem C.*

```
#include <stdio.h>
int main() {
   int idade;
   int graduacao;
   printf("Idade: ");
   scanf("%d",&idade);
   printf("Graduacao: (1) Engenharia  (2) Informatica  (3) Outro ");
   scanf("%d",&graduacao);
   if(idade>25 && graduacao==1 || graduacao==2)
        printf("Candidato aprovado");
   else
        printf("Candidato reprovado");
}
```

```
Idade: 20
Graduacao: (1) Engenharia  (2) Informatica  (3) Outro 2
Candidato aprovado
```

A resposta está incorreta porque a precedência do operador relacional `&&` é maior que do operador `||`. A expressão relacional que consta do Exemplo 3.14 é avaliada da seguinte forma: `(idade > 25 && graduacao == 1) || graduacao == 2`

Para corrigir a precedência dos operadores relacionais é necessário usar parênteses (Exemplo 3.15).

Exemplo 3.15. *Alteração de precedência entre operadores com o uso de parênteses.*

```
#include <stdio.h>
int main() {
   int idade;
   int graduacao;
   printf("Idade: ");
   scanf("%d",&idade);
   printf("Graduacao - (1) Engenharia  (2) Informatica  (3) Outro: ");
```

```
   scanf("%d",&graduacao);
   if(idade>25 && (graduacao==1 || graduacao==2))
        printf("Candidato aprovado");
   else
        printf("Candidato reprovado");
}

Idade: 20
Graduacao - (1) Engenharia   (2) Informatica    (3) Outro: 2
Candidato reprovado.
```

3.5.1 Tabela de Precedência Entre Operadores

A Tabela 3.5 apresenta a precedência entre os operadores aritméticos, relacionais e lógicos utilizados até esse ponto do curso.

Tabela 3.5: Precedência entre Operadores

Símbolo	Descrição
`( )`	Parênteses
`-`	Menos unário
`!`	Não lógico
`* / %`	Mutiplicação, divisão e módulo
`+ -`	Adição e subtração
`< <= > >=`	Menor que, menor ou igual a, maior que, maior ou igual a
`== !=`	Igual, diferente
`&&`	E lógico
`||`	OU lógico
`?:`	Operador condicional ternário

Exercício 3.17. *Escrever um programa C que solicite ao usuário a digitação do dia e do mês corrente no formato DD/MM e que verifique se é uma data válida ou inválida. Por exemplo,* 30/02 *e* 31/04 *são datas inválidas.*

3.6 Comando de Desvio Condicional `switch`

O comando de desvio condicional `switch` é uma alternativa ao uso de diversas estruturas `if-else` em problemas de múltiplas opções.

A forma geral do comando `switch` é apresentada a seguir:

```
switch(expressão) {
   case constante1:
      sequência_de_comandos;
      break;
   case constante2:
      sequência_de_comandos;
      break;
   case constante3:
      sequência_de_comandos;
      break;
   case ...

   default:
      sequência_de_comandos;
}
```

Uma *sequência de comandos* diferencia-se de um bloco de comandos por não ser delimitado por chaves de abertura e de fechamento.

A execução do comando `switch` segue os seguintes passos:

- A expressão é avaliada.
- O resultado da expressão é comparado com as constantes associadas às cláusulas `case`.
- Quando o resultado da expressão for igual a uma das constantes, a execução é transferida para o início da sequência de comandos associada a esta constante.
- A execução continua sequencialmente até o fim do comando `switch` a menos que um comando `break` seja encontrado. Nesse caso, a execução do comando `switch` é terminada.
- Caso o valor da expressão não corresponda a nenhuma das constantes associadas às cláusulas `case`, a sequência de comandos associada à cláusula `default` é executada.
- A cláusula `default` é opcional. Nenhuma sequência de comandos será executada caso o valor da expressão não corresponda a nenhuma cláusula `case` e não houver uma cláusula `default` definida.

O comando `break` é um dos comandos de *desvio incondicional* da linguagem C. O comando `break` é usado no corpo do comando `switch` para interromper a execução de uma sequência de comandos e pular para a instrução seguinte ao comando `switch`.

Há ainda alguns pontos importantes que devem ser mencionados sobre o comando `switch`:

- O resultado da *expressão* avaliada deve ser um valor inteiro.
- Caso não exista um comando de desvio incondicional `break`, todas as instruções seguintes à cláusula `case` selecionada serão executadas, mesmo que pertençam as sequências de comandos seguintes.

- O comando `switch` só pode testar a igualdade.
- Cada constante associada a uma cláusula `case` deve possuir um valor diferente.
- Não há uma ordem estabelecida para as diversas cláusulas `case` de um comando `switch`.

O Exemplo 3.16 ilustra a aplicação do comando de desvio condicional `switch`.

Exemplo 3.16. *Escrever um programa C para solicitar ao usuário a digitação de um número inteiro correspondente a classificação em uma competição esportiva e que apresente na tela a medalha conquistada.*

```
#include <stdio.h>
int main() {
   int clas;
   printf("Digite a classificacao: ");
   scanf("%d",&clas);
   switch(clas) {
      case 1:
         printf("Medalha de ouro");
         break;
      case 2:
         printf("Medalha de prata");
         break;
      case 3:
         printf("Medalha de bronze");
         break;
      default:
         printf("Sem medalha");
   }
}
```

```
Digite a classificacao: 2
Medalha de prata.
```

Observe que as cláusulas `case` de um comando `switch` não podem ser associadas a expressões relacionais. O Exemplo 3.17 ilustra essa restrição da estrutura.

Exemplo 3.17. *Escrever um programa C para solicitar ao usuário a digitação de uma resposta do tipo* sim *ou* não *a uma pergunta. O usuário pode usar letras maiúsculas ou minúsculas ao responder.*

```
#include <stdio.h>
int main() {
   char resp;
```

```
   printf("Digite (S)im ou (N)ao: ");
   scanf("%c", &resp);
   switch(resp) {
      case 'S':
      case 's':
         printf("Voce respondeu SIM");
         break;
      case 'N':
      case 'n':
         printf("Voce respondeu NAO");
         break;
      default:
         printf("Voce respondeu incorretamente");
   }
}
```

Exercício 3.18. *Escrever um programa C que solicite ao usuario o peso de uma pessoa na Terra e o numero correspondente a um dos demais planetas do sistema solar e que apresente na tela o peso no planeta especificado. Considere a Tabela 3.6 para os valores da gravidade relativa dos planetas tomando como base* 1 *o valor na Terra.*

Tabela 3.6: Gravidade relativa nos planetas do Sistema Solar

Número	**Gravidade**	**Planeta**
1	0,37	Mercúrio
2	0,88	Vênus
3	0,38	Marte
4	2,64	Júpiter
5	1,15	Saturno
6	1,17	Urano

Exercício 3.19. *Escrever um programa C que solicite ao usuário qual o prato principal, a sobremesa e a bebida escolhidos a partir do cardápio de um restaurante e que informe o total de calorias que será consumido pelo cliente. Considere a Tabela 3.7 de calorias.*

3.7 Funções para Teste de Caracteres

A biblioteca padrão `<ctype.h>` disponibiliza diversas funções para teste de caracteres, conforme apresentado na Tabela 3.8. Em todos os casos a função retorna verdadeiro ou falso.

Tabela 3.7: Calorias por item do cardápio de um restaurante

Prato	**Sobremesa**	**Bebida**
Vegetariano (180 cal)	Abacaxi (75 cal)	Cha (20 cal)
Peixe (230 cal)	Sorvete diet (110 cal)	Suco de Laranja (70 cal)
Frango (250 cal)	Mousse diet (170 cal)	Suco de Melão (100 cal)
Carne de Porco (350 cal)	Mousse chocolate (200 cal)	Refrigerante (120 cal)

Tabela 3.8: Funções para teste de caracteres

Função	**Teste**
islower	Letra minúscula
isupper	Letra maiúscula
isalpha	Letra maiúscula ou minúscula
isalnum	Letra ou dígito numérico
isdigit	Dígito numérico
isblank	Espaço em branco ou marca de tabulação

Além das funções para teste de caracteres, a biblioteca padrão `<ctype.h>` oferece as funções `tolower()` e `toupper()` que permitem a conversão de uma letra maiúscula para minúscula e vice-versa.

Exemplo 3.18. *Escrever um programa C para solicitar ao usuário a digitação de um caractere em tempo de execução. O programa deve verificar se o usuário digitou uma vogal ou uma consoante.*

```
#include <stdio.h>
#include <ctype.h>
int main() {
    char ch;
    printf("Digite um caractere: ");
    scanf("%c",&ch);
    if(isalpha(ch)) {
       if(islower(ch))
          ch = toupper(ch);
       if(ch=='A' || ch=='E' || ch=='I' || ch=='O' || ch=='U')
          printf("Voce digitou uma vogal");
       else
          printf("Voce digitou uma consoante");
    }
```

```
}
```

```
Digite um caractere: a
Voce digitou uma vogal
```

Exercício 3.20. *Escrever um programa C para solicitar ao usuário a digitação de um caractere em tempo de execução. O programa deve verificar se o usuário digitou letra, número, espaço ou nenhum desses.*

Exercício 3.21. *Escrever um programa C para solicitar ao usuário a digitação de um caractere em tempo de execução. O programa deve verificar se o usuário digitou letra ou número. Se digitou letra, informar se a letra digitada é maiúscula ou minúscula.*

Exercício 3.22. *Escrever um programa C que solicite ao usuário a digitação da sigla da unidade da federação (UF) e que exiba o nome do Estado correspondente. Para simplificar o exercício, considere apenas os Estados da região Sudeste.*

Exercício 3.23. *Um número hexadecimal pode conter dígitos numéricos (*0 *a* 9*) e letras (A a E). Escreva um programa C que solicite ao usuário a digitação de um número hexadecimal composto por um único dígito e que exiba na tela o correspondente valor em decimal.*

3.8 Operador Aritmético de Atribuição

A linguagem C permite a codificação de programas-fonte concisos. Para tanto, fornece atalhos para alguns operadores.

O operador aritmético de atribuição permite reunir um comando de atribuição a um operador aritmético. Por exemplo, a linha código:

```
num = num + 10;
```

pode ser reescrita na forma

```
num += 10;
```

O operador aritmético de atribuição pode ser empregado em conjunto com os operadores aritméticos de adição (+), subtração (-), multiplicação (*), divisão (/) e módulo (%).

Capítulo 4

Estruturas de Repetição

A mente que se abre a uma nova ideia
jamais voltara ao seu tamanho original – Albert Einstein

Em diversos casos, um algoritmo necessita repetir parte do codigo diversas vezes. O mecanismo de repetição e um dos mais importantes recursos empregados em programas de computador. Atraves da iteração de lacos, um trecho definido pode ser executado quantas vezes forem necessarias para se implementar corretamente o algoritmo desejado.

A linguagem C oferece tres comandos de repetição: `for`, `while` e `do-while`. Dependendo das caracteristicas do algoritmo a ser implementado, pode ser mais interessante empregar um ou outro dos comandos mencionados.

4.1 Operadores de Incremento e Decremento

Os operadores de incremento (`++`) e de decremento (`--`) permitem, respectivamente, adicionar ou subtrair uma unidade de uma variável do tipo inteiro.

Uma expressão aritmética que contenha o operador de adição para somar uma unidade pode ser reescrita utilizando-se o operador de incremento `++` (Exemplo 4.1).

Exemplo 4.1. *Emprego do operador de incremento.*

```
#include <stdio.h>
int main() {
   int a;
   a = 7;
   a++; /* equivale a a = a + 1; */
   printf("%d", a);
}
```

De forma análoga, uma expressão aritmética que contenha o operador de subtração para subtrair uma unidade pode ser reescrita utilizando-se o operador de decremento `--` (Exemplo 4.2).

Exemplo 4.2. *Emprego do operador de decremento.*

```
#include <stdio.h>
int main() {
   int a;
   a = 7;
   a--; /* equivale a a = a - 1; */
   printf("%d", a);
}
```

4.1.1 Formas Pré-fixada e Pós-fixada para os Operadores de Incremento e Decremento

É importante observar alguns particularidades quanto ao emprego dos operadores de incremento e decremento. Quando usados no contexto de expressões, o uso incorreto pode ocasionar resultados inesperados.

O operador de incremento é dito *pré-fixado* quando adiciona uma unidade à variável para depois calcular o valor da expressão (Exemplo 4.3). Ao contrário, o operador de incremento é dito *pós-fixado* quando adiciona uma unidade à variável somente depois que o cálculo do valor da expressão já foi realizado (Exemplo 4.4). A mesma consideração é válida para o operador de decremento.

Exemplo 4.3. *Emprego do operador de incremento pré-fixado.*

```
#include <stdio.h>
#include <math.h>
int main() {
    int num1, num2;
    num1 = 10;
    num2 = num1++;
    printf("%d",num2);
}
```

Exemplo 4.4. *Emprego do operador de incremento pós-fixado.*

```
#include <stdio.h>
#include <math.h>
int main() {
    int num1, num2;
```

```
    num1 = 10;
    num2 = num1++;
    printf("%d",num2);
}
```

4.2 Estrutura de Repetição `for`

A estrutura de repetição `for` aparece em várias linguagens de programação, mas em C apresenta uma grau maior de flexibilidade.

A ideia básica da estrutura `for` consiste em empregar uma variável, geralmente um *contador*, para controlar as repetições a serem executadas. O bloco de comandos associada à estrutura `for` é executado e ao final a variável de controle é incrementada ou decrementada e comparada com o valor final que deve alcançar. Caso a condição para o término da repetição tenha sido atingida a execução da estrutura `for` é interrompida.

4.2.1 Forma Geral da Estrutura `for`

A forma geral da estrutura de repetição `for` é:

```
for(valor_inicial;teste;passo)
bloco_de_comandos
```

onde:

- o `valor_inicial` corresponde a uma expressão utilizada para inicializar a variável de controle da estrutura `for`;
- o `teste` corresponde a uma expressão cujo objetivo é verificar se a estrutura `for` deve ser terminada;
- o `passo` permite alterar o valor da variável de controle da estrutura `for`.

A execução da estrutura `for` segue os seguintes passos:

1. o `valor_inicial` é atribuído à variável de controle;
2. o `teste` é avaliado para determinar se a estrutura de repetição `for` deve ser executada;
3. se o resultado do `teste` for verdadeiro o `bloco_de_comandos` é executado, caso contrário a instrução `for` é terminada;
4. o `passo` é processado;
5. a execução retorna ao passo 2.

O Exemplo 4.5 apresenta um código-fonte C que emprega a estrutura de repetição `for`.

Exemplo 4.5. *Programa para exibir na tela todos os números inteiros no intervalo* $[1, 10]$.

```
#include <stdio.h>
int main() {
   int i; /* variável de controle */
   for(i=1; i<=10; i++)
      printf("%d\n", i); /* bloco de comandos */
}
```

Os Exemplos 4.6 e 4.7 apresentam outros casos de uso para a estrutura de repetição `for`.

Exemplo 4.6. *Programa para exibir na tela todos os números pares no intervalo* $[1, 20]$.

```
#include <stdio.h>
int main() {
   int i; /* variável de controle*/
   for(i=2; i<=20; i+=2)
      printf("%d\n", i); /* bloco de comandos */
}
```

O Exemplo 4.7 apresenta um algoritmo para calcular uma soma de forma iterativa, onde cada novo valor a ser adicionado é gerado em uma estrutura de repetição. É importante observar que em algoritmos para cálculo de um somatório é necessário a atribuição de um valor inicial para a variável que recebe o valor da soma. Normalmente esse valor inicial corresponde ao número 0 por se tratar do valor neutro da adição.

Exemplo 4.7. *Programa para calcular e exibir na tela a soma dos números inteiros no intervalo* $[1, 100]$.

```
#include <stdio.h>
int main() {
   int i; /* variável de controle*/
   int soma = 0;
   for(i=1; i<100; i++)
      soma += i;
   printf("Soma: %d",soma);
}
```

4.2.2 Laço Infinito

É importante observar que o programador é inteiramente responsável pela construção de uma estrutura de repetição `for` que contenha uma *condição de teste* que efetivamente termine a execução do laço. Caso contrário, a execução do laço continuará indefinidamente. Esse erro de lógica é chamado *laço infinito* (Exemplo 4.8).

Exemplo 4.8. *Laço infinito em estrutura de repetição* `for`.

```
#include <stdio.h>
int main() {
    int i;
    for(i=1; i>=0 ; i++) /* Laço infinito! */
        printf("%d\n",i);
}
```

Exercício 4.1. *Escreva um programa C para exibir na tela em ordem decrescente todos os numeros naturais no intervalo* $[1, 25]$.

Exercício 4.2. *Escreva um programa C para exibir na tela em ordem decrescente todos os numeros ímpares no intervalo* $[1, 50]$.

Exercício 4.3. *Escreva um programa C para exibir na tela em ordem crescente* 10 *números reais no intervalo* $[0, 1]$.

Exercício 4.4. *Escrever uma programa C para exibir na tela uma tabela com a temperatura em graus Celsius e a temperatura equivalente em Fahrenheit, para valores entre* 15 *e* 35 *graus Celsius. Fórmula de conversão:* $T_F = 9T_C/5 + 32$

Exercício 4.5. *Escrever uma programa C que exiba na tela todos os números inteiros compreendidos entre um limite inferior e um limite superior fornecidos pelo usuário em tempo de execução.*

Exercício 4.6. *Escrever uma programa C que exiba na tela em ordem crescente todos os números inteiros compreendidos entre um limite inferior e um limite superior fornecidos pelo usuário em tempo de execução. Observe que o usuário pode digitar primeiro o limite superior.*

Exercício 4.7. *Escreva um programa C que solicite ao usuario a digitaçao de* 10 *numeros inteiros e positivos e que calcule e exiba na tela a media aritmetica dos numeros fornecidos.*

Exercício 4.8. *Escreva um programa C que solicite ao usuario a digitaçao de* 10 *caracteres e que calcule e exiba na tela a quantidade de letras do alfabeto que foram digitadas.*

Exercício 4.9. *Escreva um programa C que solicite ao usuario a digitaçao de* 10 *caracteres e que calcule e exiba na tela a quantidade de letras maiúsculas que foram digitadas.*

Exercício 4.10. *Escreva um programa C que solicite ao usuario a digitaçao de* 10 *numeros inteiros. Exibir na tela o maior valor informado pelo usuário.*

Exercício 4.11. *Escreva um programa C para calcular e exibir na tela o fatorial de um numero inteiro e positivo, fornecido pelo usuario em tempo de execuçao.* $N! = 1 \times 2 \times 3 \times \cdots \times N$, *para* $N > 0$ *e* $N! = 1$ *para* $N = 0$.

Exercício 4.12. *Escreva um programa C que solicite ao usuario a digitaçao de um numero inteiro e positivo N e que calcule e mostre na tela a seguinte soma:* $S = 1 + 1/2 + 1/3 + 1/4 + 1/5 + \cdots + 1/N$.

Exercício 4.13. *Escreva um programa C que solicite ao usuario a digitaçao de um numero inteiro e positivo N e que calcule e mostre na tela a seguinte soma:* $S = 1 - 1/2 + 1/3 - 1/4 + 1/5 - \cdots \pm 1/N$.

Exercício 4.14. *Escreva um programa C que solicite ao usuario a digitaçao de um numero inteiro e positivo qualquer e que verifique se o numero e primo. Um número inteiro e positivo é dito primo se divisível apenas por* 1 *e por ele próprio.*

Exercício 4.15. *Escrever um programa C para calcular e exibir na tela aproximações para o numero irracional π a partir da serie:* $\pi = 4\frac{4}{3} + \frac{4}{5}\frac{4}{7} + \frac{4}{9}\ldots$ *Exiba o resultado para* 10, 100, 1000 *e* 10000 *termos da serie.*

Exercício 4.16. *Escrever um programa C que calcule e exiba na tela os N primeiros termos da serie de Fibonacci:* $1, 1, 2, 3, 5, 8, 13, 21, \ldots,$ *onde N e fornecido pelo usuario em tempo de execucao.*

4.2.3 Estruturas `for` Avançadas

A estrutura de repetição `for` da linguagem C é bastante flexível e permite construções complexas. Na maioria dos casos, porém, essas construções devem ser evitadas porque tornam o código-fonte confuso.

Estrutura `for` Com Mais de Uma Variável de Controle

O Exemplo 4.9 apresenta um programa-fonte C onde a estrutura `for` utiliza duas variáveis de controle.

Exemplo 4.9. *Estrutura de repetição que utiliza duas variáveis de controle.*

```
#include <stdio.h>
int main() {
    int i;
    int j;
    for (i=0, j=10; i+j>0; i++, j-=2)
        printf("i = %d e j = %2d\n", i, j);
}
```

```
i = 0 e j = 10
i = 1 e j =  8
i = 2 e j =  6
i = 3 e j =  4
```

```
i = 4 e j =  2
i = 5 e j =  0
i = 6 e j = -2
i = 7 e j = -4
i = 8 e j = -6
i = 9 e j = -8
```

Estrutura `for` Sem Valor Inicial

Além disso, vale observar que não é necessário definir o valor inicial para a variável de controle de uma estrutura `for` diretamente no comando (Exemplo 4.10).

Exemplo 4.10. *Estrutura de repetição* `for` *sem a inicialização da variável de controle.*

```
#include <stdio.h>
int main() {
    int i = 0;
    for ( ; i<10; i++)
        printf("%d\n", i);
}
```

Estrutura `for` Sem Incremento

De forma análoga, o incremento não necessariamente deve ser explicitado na estrutura de repetição `for` (Exemplo 4.11).

Exemplo 4.11. *Estrutura de repetição* `for` *sem a definição do incremento da variável de controle.*

```
#include <stdio.h>
int main() {
    int i = 0;
    for ( ; i<10; )
        printf("%d\n", i++);
}
```

Estrutura `for` Sem Condição de Teste

Mesmo a condição de teste pode ser suprimida em uma estrutura de repetição `for`. Nesse caso, o bloco de comandos associado à estrutura deve permitir que a execução do laço `for` seja interrompido caso contrário ocorrerá um laço infinito.

O Exemplo 4.12 utiliza o comando incondicional `break` para terminar a execução da estrutura de repetição `for` quando uma determinada condição é atingida.

Exemplo 4.12. *Estrutura de repetição* `for` *sem a condição de teste.*

```
#include <stdio.h>
int main() {
    int i = 0;
    for ( ; ; ) {
        printf("%d\n", i++);
        if(i==10)
            break;
    }
}
```

Estrutura `for` Sem Bloco de Comandos

Por fim, é possível implementar uma estrutura de repetição for sem um bloco de comandos associado. Nessa situação, todo o conteúdo da estrutura limita-se aos parâmetros informados nas condições iniciais, teste e incremento (Exemplo 4.13). Observe que nesse caso é necessário inserir as chaves de abertura e fechamento para especificar um bloco de comandos vazio.

Exemplo 4.13. *Estrutura de repetição* `for` *sem bloco de comandos.*

```
#include <stdio.h>
int main() {
    long soma = 0;
    int i;
    for(i=1; i<=100; soma+=i++) {
    }
    printf("Soma: %d", soma);
}
```

```
Soma: 5050
```

4.2.4 Laços Aninhados

Em alguns algoritmos é necessário inserir uma estrutura de repetição `for` dentro do bloco de comandos de uma outra estrutura de repetição `for`. Quando se tem uma estrutura de repetição no interior de outra estrutura de repetição dá-se o nome de *laços aninhados.*

O Exemplo 4.14 apresenta o funcionamento de estruturas `for` aninhadas.

Exemplo 4.14. *Programa C para exibir o processo de execução de laços* `for` *aninhados.*

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

```
int main() {
    int i, j;
    int num;
    printf("Numero de inteiro: ");
    scanf("%d",&num);
    for (i=1; i<=num; i++)
        for (j=1; j<=i; j++)
            printf("i=%d  j=%d\n",i,j);
}
```

```
Numero de inteiro: 3
i=1  j=1
i=1  j=2
i=1  j=3
i=2  j=1
i=2  j=2
i=2  j=3
i=3  j=1
i=3  j=2
i=3  j=3
```

Exercício 4.17. *Escrever um programa C exiba na tela um "quadrado" formado pelo caractere * repetido diversas vezes. O tamanho do quadrado, ou seja, o número de asteriscos na linha e na coluna deve ser informado pelo usuário.*

Exercício 4.18. *Escrever um programa C exiba na tela um "retângulo" formado pelo caractere * repetido diversas vezes. O tamanho do retângulo, ou seja, o número de asteriscos na linha e na coluna devem ser informados pelo usuário.*

Exercício 4.19. *Escrever um programa C exiba na tela um "triângulo" formado pelo caractere * repetido diversas vezes. O tamanho do triângulo, ou seja, o número de asteriscos na maior largura e maior altura devem ser informados pelo usuário.*

Exercício 4.20. *Escrever um programa C que solicite ao usuário a digitação de diversos valores inteiros e positivos. A entrada de dados deve terminar quando o usuário informar um valor negativo. Exibir a soma dos valores digitados pelo usuário. O valor negativo não deve ser somado.*

4.3 Estrutura de Repetição `while`

A estrutura de repetição `while` permite repetir um trecho de codigo enquanto uma determinada condição for verdadeira. Em geral, uma estrutura `while` fornece um código mais simples e fácil de ser entendido do que a estrutura `for` quando nao se conhece a priori o numero de repetições que serao executadas no corpo do laço.

4.3.1 Forma Geral da Estrutura `while`

A forma geral da estrutura de repetição `while` é:

```
while(condição)
   bloco_de_comandos
```

onde a `condição` corresponde a uma expressão relacional que deve ser logicamente avaliada e `bloco-de-comandos` corresponde a uma ou mais instruções C que serão executadas somente enquanto a condição avaliada for verdadeira.

O Exemplo 4.15 ilustra a aplicação da estrutura de repetição `while`.

Exemplo 4.15. *Programa C para exibir na tela todos os numeros naturais menores que* 100*.*

```
#include <stdio.h>
int main() {
   int i = 1;
   while(i<100) {
      printf("%4d",i);
      i++;
   }
   printf("\n");
}
```

Uma solução alternativa poderia empregar o operador unario de incremento `++`. Nesse caso, como o bloco de comandos do comando `while` é constituído por apenas uma única instrução C, não é necessário a delimitação através de chaves de abertura e fechamento (Exemplo 4.16).

Exemplo 4.16. *Programa C para exibir na tela todos os numeros naturais menores que* 100 *onde o bloco de códigos contém apenas uma única instrução.*

```
#include <stdio.h>
int main() {
   int i = 1;
   while(i<100)
      printf("%4d",i++);
   printf("\n");
}
```

Exemplo 4.17. *Programa C para calcular e exibir na tela a soma dos numeros naturais até* 100*.*

```
#include <stdio.h>
int main() {
   int num = 1;
   int soma = 0;
   while(num<=100) {
      soma+=num;
      num++;
   }
   printf("Soma: %d\n",soma);
}
```

4.3.2 *Flag* ou Sentinela

Em muitos algoritmos, nao se conhece previamente o numero de iterações que uma estrutura de repetição deverá executar. Na maioria dos casos, um laço deve ser executado ate que o usuario digite um valor que determina o termino das iterações. Esse valor informado e chamado *flag* ou *sentinela*. No Exemplo 4.18, o *flag* é o numero 0.

Exemplo 4.18. *Programa C para calcular a soma dos numeros inteiros fornecidos pelo usuario em tempo de execução. A entrada de dados termina quando o usuario digita o valor* 0*.*

```
#include <stdio.h>
int main() {
    int soma = 0;
    int num;
    printf("Digite um numero inteiro: ");
    scanf("%d",&num);
    while(num!=0) {
        soma+=num;
        printf("Digite um numero inteiro: ");
        scanf("%d",&num);
    }
    printf("Soma: %d", soma);
}
```

```
3
5
0
Soma: 8
```

4.3.3 Estrutura `while` Sempre Verdadeira

Uma estrutura de repetição `while` pode ser escrita de forma que a condição a ser avaliada seja sempre verdadeira. Nesse caso, o término da execução da estrutura de repetição deverá ser explicitamente implementada no corpo do laço através de um comando `break`.

O Exemplo 4.19 ilustra a aplicação de uma estrutura `while` sempre verdadeira.

Exemplo 4.19. *Programa C para calcular a soma dos numeros inteiros fornecidos pelo usuario em tempo de execução utilizando uma estrutura* `while` *sempre verdadeira. A entrada de dados termina quando o usuario digita o valor* 0.

```
#include <stdio.h>
int main() {
    int soma = 0;
    int num;
    while(1) {
        printf("Digite um numero inteiro: ");
        scanf("%d",&num);
        if(num==0)
           break;
        soma+=num;
    }
    printf("Soma: %d", soma);
}

3
5
0
Soma: 8
```

Para resolver os Exercícios 4.21 a 4.27 utilize a estrutura de repetição `while`.

Exercício 4.21. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de* 10 *números inteiros quaisquer. Calcule e exiba na tela a soma somente dos números positivos.*

Exercício 4.22. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de diversos caracteres. A entrada de dados deve terminar quando o usuário digitar um ponto. Exibir na tela o total de caracteres digitados.*

Exercício 4.23. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de diversos caracteres. A entrada de dados deve terminar quando o usuário digitar um ponto. Exibir na tela o total de letras e o total de dígitos numéricos digitados.*

Exercício 4.24. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de diversos caracteres. A entrada de dados deve terminar quando o usuário*

digitar qualquer caractere que não for uma letra. Exibir na tela o total de letras maiúsculas e o total de letras minúsculas digitadas.

Exercício 4.25. *Dado um pais A, com* 5 *milhoes de habitantes e uma taxa de natalidade de* 3% *ao ano, e um pais B com* 7 *milhoes de habitantes e uma taxa de natalidade de* 2% *ao ano, fazer um programa C para calcular e exibir na tela o tempo necessario para que a população do pais A ultrapasse a população do pais B.*

Exercício 4.26. *Escrever um programa C que solicite ao usuario a digitação em tempo de execução de diversos numeros inteiros no intervalo* $[1, 1000]$. *O* flag *e qualquer valor fora desse intervalo. Apresentar na tela o maior e o menor valor válido – ou seja, dentro do intervalo – digitado pelo usuario.*

Exercício 4.27. *Escrever um programa C que solicite ao usuário a digitação de diversos numeros inteiros e positivos. O programa deve verificar cada um dos numeros fornecidos e um quadrado perfeito. O* flag *e um número não-positivo. Um numero e dito quadrado perfeito quando a raiz quadrada e um numero inteiro.*

4.4 Estrutura de Repetição `do-while`

De forma semelhante à estrutura `while`, a estrutura de repetição `do-while` permite repetir um trecho de codigo enquanto uma determinada condição for verdadeira.

Entretanto, guarda uma diferenca importante em relação à estrutura `while`. Em um laço `do-while` o bloco de comandos associado à estrutura de repetição e executado obrigatoriamente pelo menos uma vez, independente da expressão relacional associada ao comando. Isso acontece porque a avaliação da condição e feita no final da estrutura de repetição `do-while`.

4.4.1 Forma Geral da Estrutura `do-while`

A forma geral da estrutura de repetição `do-while` é:

```
do
   bloco_de_comandos
while(condição)
```

onde a `condição` corresponde a uma expressão relacional que deve ser logicamente avaliada e `bloco-de-comandos` corresponde a uma ou mais instruções C que serão executadas somente enquanto a condição avaliada for verdadeira.

O Exemplo 4.20 ilustra a aplicação da estrutura de repetição `while`.

Exemplo 4.20. *Programa C para exibir na tela todos os numeros naturais menores que* 100 *utilizando a estrutura de repetição* `do-while`.

```
#include <stdio.h>
int main() {
   int num = 1;
   do {
      printf("%4d",num);
      num++;
   } while(num<100);
   printf("\n");
}
```

Em alguns casos, a estrutura de repetição `do-while` permite a construção de algoritmos relativamente mais simples e de melhor implementação em comparação à estrutura `while` (Exemplo 4.21).

Exemplo 4.21. *Programa C para calcular a soma dos numeros inteiros fornecidos pelo usuario em tempo de execucao utilizando o comando* `do-while`*. A entrada de dados termina quando o usuario informa o valor* 0*.*

```
#include <stdio.h>
int main() {
    int num, soma = 0;
    do {
        printf("Digite um numero inteiro: ");
        scanf("%d",&num);
        soma+=num;
    } while (num!=0);
    printf("Soma: %d",soma);
}

3
5
0
Soma: 8
```

4.5 Comando continue

O comando `continue` permite ignorar o restante das instruções inseridas em uma estrutura de repetição e avançar a execução para a avaliação da condição de teste. O Exemplo 4.22 ilustra a aplicação do comando `continue`.

Exemplo 4.22. *Programa C para calcular a soma dos numeros inteiros fornecidos pelo usuario em tempo de execucao utilizando o comando* `continue`*. A entrada de dados termina quando o usuario informa o valor* 0*.*

```
#include <stdio.h>
int main() {
    int num, soma = 0;
    do {
        printf("Digite um numero inteiro: ");
        scanf("%d",&num);
        if(num==0)
           continue;
        soma+=num;
    } while(num!=0);
    printf("Soma: %d",soma);
}

3
5
0
Soma: 8
```

Para resolver os Exercícios 4.28 a 4.34 utilize a estrutura de repetição `do-while`.

Exercício 4.28. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de* 10 *números inteiros quaisquer. Calcule e exiba na tela a soma somente dos números positivos.*

Exercício 4.29. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de diversos caracteres. A entrada de dados deve terminar quando o usuário digitar um ponto. Exibir na tela o total de caracteres digitados.*

Exercício 4.30. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de diversos caracteres. A entrada de dados deve terminar quando o usuário digitar um ponto. Exibir na tela o total de letras e o total de dígitos numéricos digitados.*

Exercício 4.31. *Escrever um programa C que solicite ao usuário a digitação em tempo de execução de diversos caracteres. A entrada de dados deve terminar quando o usuário digitar qualquer caractere que não for uma letra. Exibir na tela o total de letras maiúsculas e o total de letras minúsculas digitadas.*

Exercício 4.32. *Dado um pais A, com* 5 *milhoes de habitantes e uma taxa de natalidade de* 3% *ao ano, e um pais B com* 7 *milhoes de habitantes e uma taxa de natalidade de* 2% *ao ano, fazer um programa C para calcular e exibir na tela o tempo necessario para que a população do pais A ultrapasse a população do pais B.*

Exercício 4.33. *Escrever um programa C que solicite ao usuario a digitação em tempo de execução de diversos numeros inteiros no intervalo* $[1, 1000]$. *O flag e qualquer valor fora desse intervalo. Apresentar na tela o maior e o menor valor válido – ou seja, dentro do intervalo – digitado pelo usuario.*

Exercício 4.34. *Escrever um programa C que solicite ao usuário a digitação de diversos numeros inteiros e positivos. O programa deve verificar cada um dos numeros fornecidos e um quadrado perfeito. O flag e um número não-positivo. Um numero e dito quadrado perfeito quando a raiz quadrada e um numero inteiro.*

4.6 Consistência na Entrada de Dados

Em geral, os exemplos de código-fonte C apresentados solicitam ao usuário a digitação de dados a partir da entrada padrão (teclado). O leitor mais atento deve ter observado que o usuário é instruído a informar um determinado tipo de informação — por exemplo, um valor inteiro ou uma letra do alfabeto — mas os programas-fonte não verificam se o tipo de dado solicitado foi de fato digitado.

A introdução das estruturas de repetição permitem que os algoritmos construídos em linguagem C possam de alguma forma processar o tipo de dado digitado pelo usuário e verificar se estão de acordo com o esperado. Esse processo de detecção de erros na entrada de dados recebe o nome de *consistência*.

Especificamente, a *consistência na entrada de dados* visa impedir que valores fornecidos pelo usuário em tempo de execução sejam aceitos caso estejam em desacordo com o tipo, quantidade ou limites esperados.

O Exemplo 4.23 ilustra a aplicação da técnica de consistência na entrada de dados utilizando uma estrutura de repetição `while`. O programa impede que números inteiros negativos sejam aceitos.

Exemplo 4.23. *Programa C que solicita ao usuário a digitação de* 5 *números naturais e que calcula e exibe a soma dos números informados.*

```
#include <stdio.h>
int main() {
    int i, num, soma = 0;
    for (i=1;i<=5;i++) {
        printf("Numero natural (%d/5): ",i);
        scanf("%d",&num);
        while (num<1) { // Consistência na entrada de dados
            printf("Erro.\nNumero natural (%d/5): ",i);
            scanf("%d", &num);
        }
        soma+=num;
    }
    printf("Soma = %d",soma);
}
```

Exemplo 4.24. *Programa C que solicita ao usuário a digitação de diversas letras do alfabeto e que calcula e exibe o número de letras informadas. O flag é o caractere "ponto". A consistência na entrada de dados é feita a partir de uma estrutura de repetição* `do-while`.

```
#include <stdio.h>
#include <ctype.h>
int main() {
    int letras = 0;
    char ch;
    do {
        printf("Digite uma letra: ");
        scanf(" %c",&ch);
        if(isalpha(ch))
            letras++;
        else if(ch=='.')
            break;
        else
            printf("Erro! ");
    } while(ch!='.');
    printf("Numero de letras digitadas: %d",letras);
}
```

```
Digite uma letra: e
Digite uma letra: G
Digite uma letra: 5
Erro! Digite uma letra: k
Digite uma letra: @
Erro! Digite uma letra: m
Digite uma letra: .
Numero de letras digitadas: 4
```

Exercício 4.35. *Escrever um programa C que solicite ao usuário a digitação de* 10 *numeros inteiros no intervalo* $[1, 10]$. *O programa deve calcular e exibir a soma dos números fornecidos pelo usuário. Faça a consistência na entrada de dados de forma que todos os números inteiros fora do intervalo sejam desprezados e o usuário seja instruído a digitar um novo valor.*

4.7 Números Pseudoaleatórios

A linguagem C permite a geração automática de *numeros pseudoaleatorios*, uteis em diversos problemas computacionais, principalmente em modelagem e simulaçao probabilistica. Os números gerados são ditos pseudoaleatórios porque na realidade são obtidos a partir do emprego de funções matemáticas pré-definidas.

A biblioteca padrão `<stdlib.h>` contém a função `rand()`, que retorna um numero inteiro pseudoaleatório no intervalo $[0, 32767]$. O Exemplo 4.25 ilustra a aplicação da função `rand()`.

Exemplo 4.25. *Programa C que gera números pseudoaleatórios a partir da função* `rand()`.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i;
    printf("Numeros pseudoaleatorios:\n");
    for(i=0;i<10;i++)
        printf("%d\n",rand());
}
```

4.7.1 Semente de Randomização

O problema do algoritmo implementado no Exemplo 4.25 é que a cada nova execução do programa executável gerado, os mesmos números serão gerados e exibidos na mesma ordem. A explicação para esse comportamento refere-se à forma como o compilador processa a geração de números pseudoaleatórios, ou seja, a partir de uma função matemática padrão.

Para contornar o problema é necessário alterar a *semente de randomização*. Em outros termos, é preciso alterar a fórmula matemática usada pelo compilador para a geração dos números pseudoaleatórios. Para tanto, deve-se utilizar a função `srand()` disponível também na biblioteca `<stdlib.h>`.

A função `srand()` deve receber um número inteiro como argumento. Para que que seja possível informar sempre um argumento diferente e, dessa forma, produzir uma função geradora de números pseudoaleatórios distinta, normalmente se utiliza a função `time(0)`, da biblioteca `time.h`, que retorna o número de milisegundos transcorridos desde o dia 01/01/1970. Como esse valor sempre varia no tempo, trata-se de uma boa solução para a construção de funções gerados adequadas.

O Exemplo 4.26 ilustra o emprego das funções `srand()` e `time(0)` para permitir a alteração da semente de randomização usada pela função `rand()` para a geração de números pseudoaleatórios.

Exemplo 4.26. *Programa C que permite a alteração da semente de randomização para geração de números pseudoaleatórios.*

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

```
int main() {
    srand(time(0));
    int i;
    printf("Numeros pseudoaleatorios:\n");
    for(i=0;i<10;i++)
        printf("%d\n",rand());
}
```

Conforme apresentado no Exemplo 4.26, os algoritmos implementados em C que utilizem números pseudoaleatórios devem incluir a instrução para a alteração da semente de randomização no início do bloco de comandos da função `main()`. Dessa forma, a cada nova execução do programa executável, valores diferentes serão gerados.

4.7.2 Escopo dos Números Pseudo-Aleatórios Gerados

A função `rand()` retorna um número pseudo-aleatório no intervalo $[0, 32767]$. Para alterar o intervalo pode-se empregar alguma manipulação algébrica, conforme apresentado no Exemplo 4.27.

Exemplo 4.27. *Programa C que permite a geração de números pseudoaleatórios no intervalo* $[0, 9]$.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    srand(time(0));
    int i, num;
    for(i=0;i<10;i++) {
        num = rand()%10;
        printf("%d\n",num);
    }
}
```

Mesmo para a geração de caracteres pseudo-aleatórios é possível empregar-se a função `rand()`. Nesse caso, utiliza-se a codificação numérica ASCII correspondente, conforme apresentado no Exemplo 4.28.

Exemplo 4.28. *Programa C que permite a geração de letras maiúsculas pseudoaleatórios no intervalo.*

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

```
#include <time.h>
int main() {
    srand(time(0));
    int i;
    char letra;
    for(i=0;i<10;i++) {
        letra = rand()%26+65;
        printf("%d (%c)\n",letra,letra);
    }
}
```

```
67 (C)
84 (T)
72 (H)
65 (A)
68 (D)
73 (I)
87 (W)
78 (N)
82 (R)
85 (U)
```

Exercício 4.36. *Escrever um programa C que solicite ao usuário a digitação de dois números inteiros quaisquer A e B em tempo de execução. O programa deve gerar e exibir na tela* 10 *números inteiros aleatórios no intervalo* $[A, B]$.

Exercício 4.37. *Escrever um programa C para gerar e exibir na tela* 20 *números de ponto flutuante no intervalo* $0.00, 0.99]$. *Os números gerados devem ser exibidos com dois dígitos decimais.*

Exercício 4.38. *Escrever um programa C que simule o lançamento de uma moeda* 1000 *vezes e que exiba o número de ocorrências de* cara *e de* coroa.

Exercício 4.39. *Escrever um programa C solicite ao usuário a digitação de um número inteiro no intervalo* $[0, 32767]$. *Calcular e exibir na tela quantos números pseudoaleatórios necessitam ser gerados até coincidir com o valor informado pelo usuário.*

Exercício 4.40. *Escrever um pequeno jogo de azar em C. O usuário deve tentar acertar um número inteiro pseudoaleatório escolhido pelo programa no intervalo* $[1, 100]$. *O usuário deve digitar um novo palpite até acertar. Ao final, o programa deve informar quantos palpites o usuário digitou. Nota: nesse caso, o número máximo de palpites até o acerto não deve ultrapassar* 7.

Capítulo 5

Estruturas Homogêneas

A mente que se abre a uma nova ideia
jamais voltara ao seu tamanho original – Albert Einstein

Até o momento, todas as variáveis utilizadas nos programas-fonte escritos em linguagem C receberam apenas um conteúdo por vez. Assim, em uma variável do tipo `int`, não foi possível alocar ao mesmo tempo dois valores inteiros diferentes.

Entretanto, em muitos algoritmos computacionais é necessário empregar estruturas capazes de armazenar mais de um valor do mesmo tipo de dado em uma única variável de memória. Na linguagem C, uma estrutura homogênea — também chamada *array* — permite definir uma variável que atende a esse propósito.

Teoricamente, é possível definir e manipular estruturas homogêneas de qualquer dimensão. Na prática, porém, a maioria das aplicações requer *arrays* de uma ou duas dimensões apenas.

5.1 Estruturas Homogêneas Unidimensionais

Uma estrutura homogênea unidimensional — também chamada *vetor* — possui uma única dimensão. A Figura 5.1 ilustra a representação de um vetor na memória do computador.

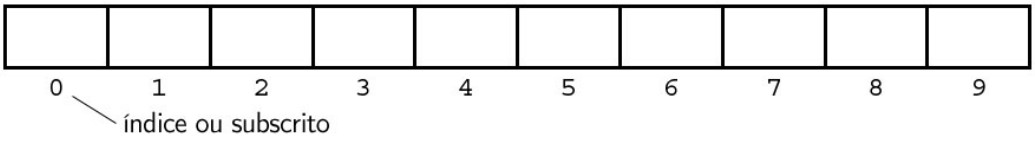

Figura 5.1: Representação de um vetor na memória

Na Figura 5.1, a estrutura homogênea compõe-se de 10 elementos que ocupam

posições contíguas de memória. Cada elemento do vetor é acessado através de um índice ou subscrito. Na linguagem C, o índice do primeiro elemento de um *array* é sempre igual a 0.

Se um vetor é declarado como sendo de um tipo inteiro, por exemplo `int`, isso implica dizer que todos os elementos deverão armazenar valores `int`. Por isso a estrutura é chamada *homogênea*. Se a Figura 5.1 for declarada como `int`, todos os elementos deverão receber valores inteiros.

5.1.1 Declaração de um *Array*

Para declarar um *array* em C deve-se informar o tipo primitivo a ser armazenado nos elementos do vetor, seguido do nome da variável e do tamanho da estrutura homogênea informado entre colchetes. Por exemplo,`int var[10];` define uma variável do tipo *array* de inteiros de tamanho 10 cujo nome é `var`.

O Exemplo 5.1 apresenta um programa C que declara um *array* unidimensional, atribui valor a dois elementos e exibe o conteúdo desses elementos do vetor na saída padrão.

Exemplo 5.1. *Programa C que declara, inicializa e exibe o conteúdo de dois elementos de um array de tamanho* 10*.*

```
#include <stdio.h>
int main() {
   int vet[10]; // Vetor de inteiros de tamanho 10
   vet[0] = 45; // Atribui o valor 45 ao elemento de índice 0 do vetor
   vet[4] = 28; // Atribui o valor 28 ao elemento de índice 0 do vetor
   printf("%d\n",vet[0]);
   printf("%d",vet[4]);
}
```

A Figura 5.2 apresenta o conteúdo do vetor `vet` declarado no Exemplo 5.1 após a execução dos comandos de atribuição presentes no código-fonte.

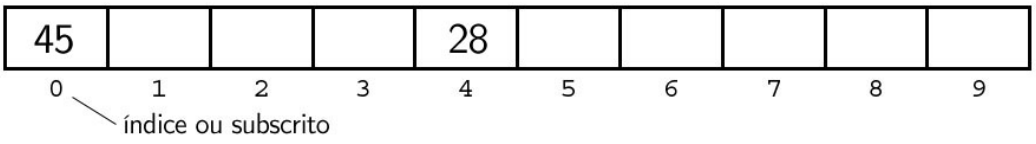

Figura 5.2: Conteúdo do vetor `vet` na memória

Pode-se observar que o acesso a um determinado elemento do vetor se dá através do respectivo índice. Essa regra vale tanto para a atribuição de valor ao elemento quanto para retornar o conteúdo do elemento, conforme apresentado no Exemplo 5.1.

5.1.2 *Arrays* e Comandos de Repetição

O emprego de comandos de repetição é muito eficiente para acessar os elementos de um vetor, seja para atribuição de valor ou para obter o conteúdo armazenado em cada elemento. Em geral, a variável de controle do laço é usada como índice para acessar os elementos do vetor. O Exemplo 5.2 ilustra a manipulação de uma estrutura homogênea unidimensional a partir de um laço `for`.

Exemplo 5.2. *programa C que cria um vetor de inteiros de tamanho* 10 *e que inicializa cada elemento com um valor inteiro fornecido pelo usuário em tempo de execução.*

```
#include <stdio.h>
int main() {
    int i, vet[10];
    for(i=0;i<10;i++) {
        printf("[%2d] ",i);
        scanf("%d",&vet[i]);
    }
    printf("\nConteudo do vetor:\n");
    for(i=0;i<10;i++)
        printf("%d ",vet[i]);
    printf("\n");
}
```

Exercício 5.1. *Escrever um programa C para criar um vetor de* `float` *de tamanho* 8 *e inicializar cada elemento com um valor fornecido pelo usuário em tempo de execução.*

Exercício 5.2. *Escrever um programa C para criar um vetor de caracteres de tamanho* 12 *e inicializar cada elemento com uma letra fornecida pelo usuário em tempo de execução. Caso o usuário digite um caractere inválido, o elemento deve ser preenchido com um asterisco.*

Exercício 5.3. *Escrever um programa C para criar um vetor de inteiros de tamanho* 10 *e inicializar cada elemento com um valor no intervalo* $[1, 10]$ *fornecido pelo usuário em tempo de execução. Caso o usuário digite um número inteiro fora do intervalo, deve ser solicitado a digitar novamente. Esse processo deve se repetir até que seja fornecido um valor válido.*

Não necessariamente uma estrutura homogênea deve ser inicializada a partir da ação do usuário. O Exemplo 5.3 apresenta um algoritmo que atribui valores a um vetor a partir do próprio índice.

Exemplo 5.3. *Programa C que cria um vetor de inteiros de tamanho* 10 *e que inicializa cada elemento com o valor do respectivo índice.*

```
#include <stdio.h>
int main() {
    int i, vet[10];
    for(i=0;i<10;i++)
        vet[i] = i;
    printf("Conteudo do vetor:");
    for(i=0;i<10;i++)
        printf("\n[ %2d] %2d",i,vet[i]);
}
```

Exercício 5.4. *Escrever um programa C para criar um vetor de inteiros de tamanho* 15*, inicializar cada elemento com números crescentes e sequenciais a partir de* 10 *e exibir o conteúdo do vetor na saída padrão.*

Exercício 5.5. *Escrever um programa C para criar um vetor de inteiros de tamanho* 10*, inicializar cada elemento com números crescentes ímpares a partir de* 1 *e exibir o conteúdo do vetor na saída padrão.*

Exercício 5.6. *Escrever um programa C para criar um vetor de inteiros de tamanho* 18*, inicializar cada elemento de forma que o conteúdo do elemento seguinte seja igual ao conteúdo do elemento anterior adicionado de* 2*. O conteúdo do primeiro elemento do vetor deve ser fornecido pelo usuário em tempo de execução. Exibir o conteúdo do vetor na saída padrão.*

Exercício 5.7. *Escrever um programa C para criar um vetor de inteiros de tamanho* 12*, inicializar cada elemento com números decrescentes múltiplos de* 3 *a partir de* 90 *e exibir o conteúdo do vetor na saída padrão.*

5.1.3 Diretiva `#define`

A diretiva para o pré-processador C denominada `#define` permite definir constantes simbólicas e que serão substituídas no código-fonte durante a compilação. Com o uso desta diretiva torna-se mais simples manter o código envolvendo constantes correto. Ela também simplifica a compreensão do código ao usar nomes simbólicos que indicam o significado das constantes definidas.

No caso de estruturas homogêneas, é recomendado o uso de diretivas `#define` para se especificar o tamanho, ou seja, o número de elementos do *array*.

A vantagem em se usar constantes simbólicas é que qualquer modificação nessas definições — por exemplo, mudar o tamanho do *array* de 100 para 120 elementos — pode ser realizada em um único local. O restante do código permanece inalterado (Exemplo 5.4).

Exemplo 5.4. *Programa C que cria um vetor de inteiros de tamanho* 10 *e que inicializa cada elemento com o valor do respectivo índice.*

```
#include <stdio.h>
#define SIZE 10
int main() {
    int i, vet[SIZE];
    for(i=0;i<SIZE;i++)
        vet[i] = i;
    printf("Conteudo do vetor:");
    for(i=0;i<SIZE;i++)
        printf("\n[ %2d] %2d",i,vet[i]);
```

5.1.4 Atribuição de Valor na Declaração do *Array*

O conteúdo de um ou mais elementos de um *array* pode ser atribuído no momento da declaração da estrutura homogênea. Nesse caso, os valores devem ser fornecidos entre parênteses e separados por vírgula (Exemplo 5.5).

Exemplo 5.5. *Programa C que cria um vetor de inteiros de tamanho* 10 *e que inicializa cada elemento na declaração do array.*

```
#include <stdio.h>
#define SIZE 10
int main() {
    int i, vet[SIZE] = {2, 5, 7, 9, 0, -1, 6, 8, -2, 4};
    printf("Conteudo do vetor:");
    for(i=0;i<SIZE;i++)
        printf("\n[%2d] %2d",i,vet[i]);
}
```

Embora não seja uma prática recomendável, é possível incumbir o próprio compilador C em determinar o tamanho do *array* a partir do número de elementos atribuídos na declaração, conforme apresentado no Exemplo 5.6.

Exemplo 5.6. *Programa C que cria um vetor de inteiros de tamanho não declarado e que inicializa cada elemento na declaração do array.*

```
#include <stdio.h>
#define SIZE 10
int main() {
    int i, vet[] = {2, 5, 7, 9, 0, -1, 6, 8, -2, 4};
    printf("Conteudo do vetor:");
    for(i=0;i<SIZE;i++)
        printf("\n[%2d] %2d",i,vet[i]);
}
```

Não é necessário atribuir valor a todos os elementos de um *array* na declaração, conforme pode ser observado no Exemplo 5.7. Os elementos que não forem explicitamente inicializados recebem o valor 0.

Exemplo 5.7. *Programa C que cria um vetor de inteiros de tamanho* 10 *e que inicializa os* 5 *primeiros elementos na declaração do array.*

```
#include <stdio.h>
#define SIZE 10
int main() {
    int i, vet[SIZE] = {-2, 3, 4, -1, 8};
    printf("Conteudo do vetor:");
    for(i=0;i<SIZE;i++)
        printf("\n[%2d] %2d",i,vet[i]);
}
```

Exercício 5.8. *Escrever um programa C para criar um vetor de inteiros de tamanho* 10*, inicializar cada elemento com* 0 *(na declaração) e exibir o conteúdo do vetor na saída padrão.*

Exercício 5.9. *Escrever um programa C para criar um vetor de inteiros de tamanho* 10 *e inicializar cada elemento com valores inteiros no intervalo* $[0, 9]$ *(na declaração) e exibir o conteúdo do array na saída padrão. Em seguida, multiplicar por* 2 *o conteúdo de cada elemento e exibir novamente o vetor na saída padrão.*

5.1.5 *Arrays* e Números Pseudoaleatórios

Números pseudoaleatórios representam uma forma conveniente de atribuir valor aos elementos de uma estrutura homogênea. O Exemplo 5.8 apresenta um programa C que utiliza números randômicos para atribuir valor a um *array*.

Exemplo 5.8. *Programa C que atribui valor a um array de inteiros de tamanho* 10 *a partir de números pseudoaleatórios no intervalo* $[0, 32767]$.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 10
int main() {
    srand(time(0));
    int i, vet[SIZE];
    printf("Conteudo do vetor:");
    for(i=0;i<SIZE;i++) {
        vet[i] = rand();
```

```
        printf("\n[%2d] %6d",i,vet[i]);
    }
}
```

Exercício 5.10. *Escrever um programa C para criar um vetor de inteiros de tamanho* 20 *cujo conteúdo seja números pseudoaleatórios no intervalo* $[0, 9]$.

Exercício 5.11. *Escrever um programa C para criar e exibir na saída padrão um vetor de inteiros de tamanho* 20 *cujo conteúdo seja números pseudoaleatórios no intervalo* $[0, 99]$. *Exibir na saída padrão o índice correspondente ao elemento de menor valor.*

Exercício 5.12. *Escrever um programa C para criar e exibir na saída padrão um vetor de inteiros de tamanho* 50 *cujo conteúdo seja números pseudoaleatórios no intervalo* $[0, 9]$. *Exibir na saída padrão a soma dos elementos do vetor.*

Exercício 5.13. *Escrever um programa C para criar e exibir na saída padrão dois vetores de inteiros A e B de tamanho* 20 *cujo conteúdo sejam números pseudoaleatórios no intervalo* $[0, 99]$. *Calcular e exibir na saída padrão o vetor soma C dado por* $C = A + B$.

Exercício 5.14. *Escrever um programa C para criar e exibir na saída padrão um vetor de caracteres* A *de tamanho* 15 *cujo conteúdo sejam letras maiúsculas geradas aleatoriamente. Gerar e exibir na saída padrão um vetor de caracteres* B *cujo conteúdo seja idêntico ao vetor* A *porém com letras minúsculas.*

Exercício 5.15. *Escrever um programa C para criar e exibir na saída padrão um vetor de inteiros de tamanho* 20 *cujo conteúdo seja números pseudoaleatórios no intervalo* $[0, 99]$. *Calcular e exibir na saída padrão o total de elementos cujo conteúdo seja números pares e o total de elementos cujo conteúdo seja números ímpares.*

Exercício 5.16. *Escrever um programa C para criar e exibir na saída padrão dois vetores* A *e* B *de* `float` *de tamanho* 10 *cujo conteúdo seja números randômicos no intervalo* $[0.00, 1.00]$. *O programa deve concatenar os dois vetores formando um novo vetor* C *de* 20 *elementos, na forma* $A[0], A[1], \ldots, A[9], B[0], B[1], \ldots, B[9]$. *Exibir na saída padrão o vetor resultante.*

Exercício 5.17. *Escrever um programa C para criar e exibir na saída padrão dois vetores* A *e* B *de* float *de tamanho* 10 *cujo conteúdo seja números randômicos no intervalo* $[0.00, 1.00]$. *O programa deve intercalar os dois vetores formando um novo vetor* C *de* 20 *elementos, na forma* $A[0], B[0], A[1], B[1], \ldots, A[9], B[9]$. *Exibir na saída padrão o vetor resultante.*

Exercício 5.18. *Escrever um programa C para criar e exibir na saída padrão dois vetores* A *e* B *de inteiros de tamanho* 12 *cujo conteúdo corresponda a números randômicos no intervalo* $[0, 999]$. *O programa deve localizar e exibir na saída padrão os números gerados que são comuns aos dois vetores.*

Exercício 5.19. *Escrever um programa C para criar e exibir na saída padrão um vetor* A *de inteiros de tamanho* 20 *cujo conteúdo corresponda a números randômicos no intervalo*

$[-100, 100]$. *Em seguida, o programa deve criar dois novos vetores* B *e* C. B *deve conter os números* < 0 *e* C *deve conter os números* ≥ 0. *Por fim, o programa deve exibir na saída padrão os vetores* B *e* C *gerados.*

Exercício 5.20. *Escrever um programa C para criar e exibir na saída padrão um vetor* A *de inteiros de tamanho* 40 *cujo conteúdo corresponda a números randômicos no intervalo* $[0, 99]$. *O programa deve gerar e exibir na saída padrão um novo vetor* A *excluindo todas as ocorrências de números repetidos do vetor* A. *Por exemplo, se o vetor* A *começar por* $45, 56, 45, 3, \ldots$ *o vetor* B *começará por* $45, 56, 3, \ldots$

Exercício 5.21. *Escrever um programa C que simule o lançamento de um dado* 1000 *vezes e que exiba na saída padrão o número de ocorrências de cada face do dado.*

5.2 Estruturas Homogêneas Bidimensionais

Uma estrutura homogênea bidimensional — também chamada *matriz* — possui duas dimensões, representadas por linhas e colunas. A Figura 5.3 ilustra a representação de uma matriz na memória do computador.

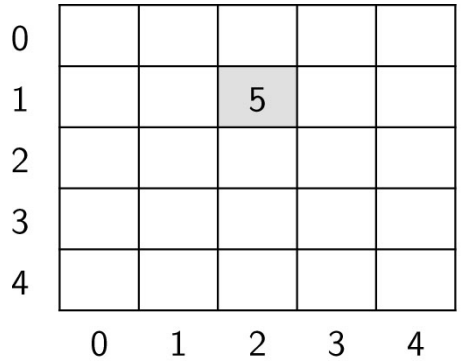

Figura 5.3: Representação de uma matriz na memória

A estrutura homogênea bidimensional representada na Figura 5.3 possui dimensão 5×5, ou seja, 5 linhas por 5 colunas. Observe que na linguagem C os índices das linhas e das colunas inicia-se obrigatoriamente pelo valor 0. Assim, o número 5 em destaque na Figura 5.3 corresponde ao conteúdo do elemento da matriz de índices 1 para linha e 2 para coluna.

Na sintaxe da linguagem C a representação em memória da matriz representada na Figura 5.3 equivalente a `int vet[5][5];` onde os valores entre colchetes correspondem ao número de linhas e de colunas da estrutura homogênea bidimensional.

Para acessar o elemento em destaque na Figura 5.3 é necessário informar o nome da matriz seguido dos índices da linha e da coluna entre colchetes. Por exemplo, o comando `mat[1][2] = 5;` atribui o valor 5 para o elemento da matriz de índices 1 (linha) e 2

(coluna). Observe que os índices correspondentes à linha e à coluna são sempre números inteiros.

Uma estrutura homogênea bidimensional que possui igual número de linhas e de colunas é chamada de *matriz quadrada*. A estrutura homogênea bidimensional representada na Figura 5.3 é uma matriz quadrada de ordem 5.

5.2.1 Percorrimento de Matriz em C

Para percorrer uma matriz em C são necessárias duas estruturas de laço — normalmente utiliza-se o comando `for` — uma para o controle de linhas e outra para o controle de colunas. O Exemplo 5.9 apresenta um programa C que utiliza números randômicos para atribuir valor aos elementos de uma matriz quadrada.

Exemplo 5.9. *Programa C que atribui valor aos elementos de uma matriz quadrada de ordem* 5 *com números inteiros aleatórios no intervalo* $[0, 9]$.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 5
int main() {
    srand(time(0));
    int i; // índice para linhas
    int j; // índice para colunas
    int mat[SIZE][SIZE];
    printf("Conteudo da matriz:\n");
    for(i=0;i<SIZE;i++) {
        for(j=0;j<SIZE;j++) {
            mat[i][j] = rand()%10;
            printf("%4d",mat[i][j]);
        }
        printf("\n");
    }
}
```

```
Conteudo da matriz:
   5   6   7   0   0
   5   0   4   0   5
   8   5   2   6   1
   6   1   1   5   9
   9   3   5   8   3
```

5.2.2 Inicialização de Matrizes

De forma semelhante aos vetores em C, uma matriz pode ser inicializada ao ser declarada. O Exemplo 5.10 apresenta um programa C que declara e inicializa os elementos de uma matriz quadrada de ordem 3. Observe que os elementos de cada linha devem ser delimitados por chaves.

Exemplo 5.10. *Programa C que declara e atribui valor aos elementos de uma matriz quadrada de ordem* 3.

```
#include <stdio.h>
#define SIZE 3
int main() {
    int i, j, mat[SIZE][SIZE] = {{1, 4, 4},{5, 7, 9},{8, 2, 3}};
    printf("Conteudo da matriz:\n");
    for(i=0;i<SIZE;i++) {
        for(j=0;j<SIZE;j++)
            printf("%4d",mat[i][j]);
        printf("\n");
    }
}
```

```
Conteudo da matriz:
   1   4   4
   5   7   9
   8   2   3
```

5.2.3 Atribuição de Valor aos Elementos de uma Matriz

Os elementos de uma matriz também podem ser fornecidos pelo usuário em tempo de execução. O Exemplo 5.11 apresenta um programa C que solicita ao usuário a digitação do conteúdo para os elementos de uma estrutura homogênea bidimensional.

Exemplo 5.11. *Programa C que solicita ao usuário a digitação em tempo de execução do conteúdo para os elementos de uma matriz de números reais de ordem* 2×3.

```
#include <stdio.h>
int main() {
    int i, j;
    float mat[2][3];
    printf("Forneca os valores para a matriz:\n");
    for(i=0;i<2;i++) {
        for(j=0;j<3;j++) {
            printf("[%d][%d] ",i,j);
```

```
            scanf("%f",&mat[i][j]);
        }
    }
    printf("\nConteudo da matriz:\n");
    for(i=0;i<2;i++) {
        for(j=0;j<3;j++)
            printf("%8.2f",mat[i][j]);
        printf("\n");
    }
}

Forneca os valores para a matriz:
[0][0] 4
[0][1] 5
[0][2] 1
[1][0] 8
[1][1] 5
[1][2] 8

Conteudo da matriz:
    4.00    5.00    1.00
    8.00    5.00    8.00
```

Exercício 5.22. *Escrever um programa C que inicialize randomicamente os elementos de uma matriz quadrada de ordem* 10 *de caracteres no intervalo* $[A, Z]$. *Exibir o conteúdo da matriz na saída padrão.*

Exercício 5.23. *Escrever um programa C que inicialize randomicamente os elementos de uma matriz quadrada de ordem* 8 *de números reais no intervalo* $[0.0, 9.9]$. *Exibir o conteúdo da matriz na saída padrão.*

Exercício 5.24. *Escrever um programa C que inicialize randomicamente os elementos de uma matriz de ordem* 5×10 *de inteiros no intervalo* $[0, 9]$. *Exibir na saída padrão a soma dos elementos da matriz gerada.*

Exercício 5.25. *Escrever um programa C que inicialize randomicamente os elementos de uma matriz quadrada de ordem* 10 *de inteiros no intervalo* $[0, 999]$. *Exibir na saída padrão o índice e o valor correspondente ao maior elemento da matriz.*

Exercício 5.26. *Escrever um programa C que inicialize randomicamente os elementos de uma matriz quadrada de ordem* 10 *de inteiros no intervalo* $[0, 99]$. *Solicitar ao usuário, em tempo de execução, a digitação de um número inteiro também no intervalo* $[0, 99]$. *Verificar se o número digitado está presente na matriz.*

Exercício 5.27. *Escrever um programa C que inicialize randomicamente os elementos de duas matriz quadradas* A *e* B *de ordem* 5 *de inteiros no intervalo* $[0, 9]$. *Calcular e apresentar na saída padrão a matriz soma* C *das matrizes geradas* $C = A + B$.

Exercício 5.28. *Uma matriz identidade é uma matriz quadrada que possui todos os membros da diagonal principal iguais a* 1*. Todos os demais elementos têm conteúdo igual a* 0*. Escrever um programa C que gere e apresente na saída padrão uma matriz identidade de ordem* 8*.*

Exercício 5.29. *Escrever um programa C que gere e apresente na saída padrão uma matriz quadrada de ordem* 8 *cujos elementos da diagonal secundária possuem números inteiros aleatórios no intervalo* $[1, 9]$ *e os demais elementos da matriz possuem valor zero.*

Exercício 5.30. *Uma matriz é dita triangular superior quando somente os elementos acima e na diagonal principal são diferentes de* 0*. Escrever um programa C que gere e apresente na saída padrão uma matriz triangular superior de ordem* 8 *cujo conteúdo são números aleatórios no intervalo* $[1, 99]$*.*

Exercício 5.31. *Escrever um programa C que inicialize randomicamente os elementos de uma matriz de ordem* 5×10 *de inteiros no intervalo* $[0, 9]$*. Gerar e apresentar na saída padrão o vetor soma de cada linha da matriz gerada.*

Exercício 5.32. *Escrever um programa C que inicialize randomicamente os elementos de uma matriz quadrada de ordem* 8 *de inteiros no intervalo* $[0, 99]$*. Apresentar na saída padrão a matriz transposta.*

Exercício 5.33. *Escrever um programa C para inicializar uma matriz quadrada de ordem* 12 *com todos os valores iguais a* $i + j$*, onde* i *é o índice correspondente à linha e* j *é o índice correspondente à coluna. Em seguida, exibir a matriz na saída padrão.*

Exercício 5.34. *Considere uma matriz quadrada de ordem* 10 *cujos elementos são inteiros no intervalo* $[0, 99]$ *gerados randomicamente. Escrever um programa C para trocar todos os números pares por* 0 *e todos os números ímpares por* 1*. Exibir a matriz resultante na saída padrão.*

Exercício 5.35. *Matriz esparsas são aquelas que possuem a maioria dos elementos com valor* 0*. Escreva um programa C para gerar e apresentar na saída padrão uma matriz quadrada de ordem* 12 *cujo conteúdo corresponda a* 0 *e* 1*. Para cada índice, considere a possibilidade de* $1/6$ *para o preenchimento com o valor* 1 *e* $5/6$ *para o preenchimento com o valor* 0*. Exiba a matriz gerada na saída padrão.*

Exercício 5.36. *Uma matriz* A *é dita simétrica se* $A^t = A$*, ou seja, é igual a sua própria matriz transposta. Escreva um programa C para gerar e apresentar na saída padrão uma matriz quadrada simétrica de ordem* 8 *cujos elementos são números inteiros no intervalo* $[1, 25]$ *gerados randomicamente.*

Capítulo 6

Cadeias de Caracteres

A mente que se abre a uma nova ideia
jamais voltara ao seu tamanho original – Albert Einstein

Cadeias de caracteres ou *strings* são estruturas de dados utilizadas para a manipulação de sequências de caracteres. Existem muitas aplicações em Algoritmos que empregam cadeias de caracteres.

Conforme visto a partir do capítulo 2, uma constante do tipo cadeira de caracteres é delimitada por aspas duplas. Assim, `"ALGORITMO"` representa uma cadeia de caracteres válida em C. Entretanto, ainda não foi apresentado como definir variáveis de memória do tipo *string* ou como ler cadeias de caracteres a partir da entrada padrão. Esse assunto será discutido nesse capítulo.

6.1 Cadeias de Caracteres em C

Na linguagem C, cadeias de caracteres são representadas por vetores do tipo `char` terminadas obrigatoriamente pelo caractere nulo (`'\0'` na tabela ASCII). Portanto, para armazenar uma cadeia de caracteres na memória, deve-se reservar uma posição adicional no vetor para o caractere de fim da cadeia. A Figura 6.1 ilustra a representação de uma cadeia de caracteres na memória do computador.

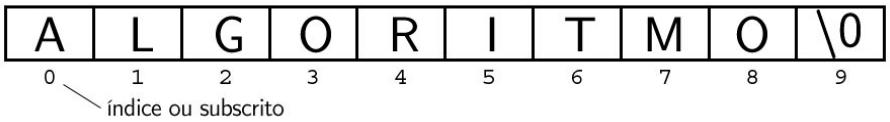

Figura 6.1: Representação de uma cadeia de caracteres na memória em C

O Exemplo 6.1 ilustra a representação de uma cadeia de caracteres. Como queremos representar a palavra ALGORITMO, composta por 9 caracteres, declaramos um vetor

com dimensão 10 (um elemento adicional para armazenarmos o caractere nulo no final da cadeia). O programa preenche cada um dos elementos do vetor, incluindo o caractere `'\0'` e exibe a *string* na saída padrão.

Exemplo 6.1. *Programa C que declara uma cadeia de caracteres e atribui valor a cada elemento separadamente.*

```
#include <stdio.h>
int main() {
   char string[10];
   string[0] = 'A';
   string[1] = 'L';
   string[2] = 'G';
   string[3] = 'O';
   string[4] = 'R';
   string[5] = 'I';
   string[6] = 'T';
   string[7] = 'M';
   string[8] = 'O';
   string[9] = '\0';
   printf("%s\n",string);
}
```

```
ALGORITMO
```

O especificador de formato `%s` da função `printf` permite exibir na saída padrão uma cadeia de caracteres. A função `printf` recebe um vetor de `char` e exibe elemento por elemento até encontrar o caractere nulo.

Atenção: se o caractere `'\0'` não for colocado ao final da cadeia de caracteres, a saída na tela pode ser executada de forma incorreta, pois não é possível localizar o identificador de fim de *string*.

Uma vez que as cadeias de caracteres são representadas através de vetores, é possível reescrever o Exemplo 6.1 utilizando-se a inicialização dos elementos conforme apresentado na declaração de um vetor de `char` (Exemplo 6.2).

Exemplo 6.2. *Programa C que declara e inicializa o conteúdo de uma cadeia de caracteres.*

```
#include <stdio.h>
int main() {
   char string[10] = {'A','L','G','O','R','I','T','M','O','\0'};
   printf("%s\n",string);
}
```

```
ALGORITMO
```

A inicialização de cadeias de caracteres nas formas apresentadas nos Exemplos 6.1 e 6.2 não é muito prática. Entretanto, a linguagem C permite que o conteúdo de *strings* possa ser atribuído escrevendo-se os caracteres entre aspas duplas. Nesse caso, o caractere nulo é representado implicitamente. O Exemplo 6.3 ilustra a declaração e inicialização de cadeias de caracteres utilizando-se essa abordagem.

Exemplo 6.3. *Programa C que atribui valor a uma cadeia de caracteres na memória.*

```
#include <stdio.h>
int main() {
   char string[10] = "ALGORITMO";
   printf("%s\n",string);
}
```

```
ALGORITMO
```

Nesse caso, não é necessário também declarar o tamanho do vetor (Exemplo 6.4). Ao atribuir valor ao vetor de `char`, o tamanho é determinado implicitamente pelo compilador da linguagem C.

Exemplo 6.4. *Programa C que atribui valor a uma cadeia de caracteres cujo tamanho é definido implicitamente.*

```
#include <stdio.h>
int main() {
   char string[] = "ALGORITMO";
   printf("%s\n",string);
}
```

```
ALGORITMO
```

No Exemplo 6.4 a variável `string` é automaticamente dimensionada e inicializada com 10 elementos. Para ilustrar a inicialização de cadeias de caracteres, considere as seguintes declarações:

```
char str1[] = "";
char str2[] = "ALGORITMO";
char str3[81];
char str4[81] = "ALGORITMO";
```

Nessas declarações, a variável `str1` armazena uma cadeia de caracteres vazia, representada por um vetor com um único elemento, o caractere `'\0'`. A variável `str2` representa um vetor com 10 elementos. A variável `str3` representa uma cadeia de caracteres capaz de representar cadeias com até 80 caracteres, já que foi dimensionada com 81 elementos. Essa variável, no entanto, não foi inicializada e seu conteúdo é desconhecido (lixo de memória). A variável `str4` também foi dimensionada para armazenar cadeias com até 80 caracteres, mas apenas os dez primeiros elementos foram utilizados na declaração.

6.2 Entrada de cadeias de caracteres

Para capturar um caractere simples fornecido pelo usuário através da entrada padrão é utilizada a função `scanf` com o especificador de formato `%c` (Exemplo 6.5).

Exemplo 6.5. *Programa C para entrada de dados do tipo caractere.*

```
#include <stdio.h>
int main() {
   char ch;
   printf("Digite um caractere: ");
   scanf("%c",&ch);
}
```

```
Digite um caractere: A
```

Dessa forma, se o usuário digitar a letra A, por exemplo, o código ASCII associado à letra A será armazenado na variável `ch`. Vale ressaltar que, diferentemente dos especificadores `%d` e `%f`, o especificador de formato `%c` não "pula" os caracteres em branco. Portanto, se o usuário teclar um espaço antes da letra A, o código ASCII correspondente ao espaço será armazenado em `ch` e o caractere A será capturado apenas em um próximo `scanf`. Se for necessário ignorar todas as ocorrências dos caracteres `' '`, `'\n'` e `'\t'` que porventura antecederem o caractere que se deseja capturar, basta incluir um espaço em branco no especificador de formato (Exemplo 6.6).

Exemplo 6.6. *Programa C para entrada de dados do tipo caractere que ignora* `' '`, `'\n'` *e* `'\t'`.

```
#include <stdio.h>
int main() {
   char ch;
   printf("Digite um caractere: ");
   scanf(" %c",&ch);
   printf("Voce digitou %c\n",ch);
}
```

```
Digite um caractere: <enter>
A
Voce digitou A
```

O especificador de formato `%s` pode ser utilizado para a entrada de dados do tipo cadeia de caracteres em um `scanf`. No entanto, seu uso possui algumas restrições. O especificador de formato `%s` termina a entrada de dados na ocorrência de um `' '`, `'\n'` ou `'\t'`.

Considere o Exemplo 6.7 que solicita ao usuário a digitação do nome completo.

Exemplo 6.7. *Programa C para entrada de dados do tipo cadeia de caracteres com especificador de formato* `%s`.

```
#include <stdio.h>
int main() {
   char nome[81];
   printf("Digite seu nome completo: ");
   scanf("%s",&nome);
   printf("Seu nome: %s\n",nome);
}
```

```
Digite seu nome completo: Wander Gaspar
Seu nome: Wander
```

Recomenda-se a entrada de dados do tipo *string* utilizando-se o especificador de formato `%[]`, no qual é listado entre os colchetes todos os caracteres que serão aceitos na leitura. Assim, por exemplo, o especificador de formato `%[aeiou]` lê sequências de vogais minúsculas, isto é, a leitura prossegue até se encontrar um caractere que não seja uma vogal em letras minúsculas (Exemplo 6.8).

Exemplo 6.8. *Programa C para entrada de dados do tipo cadeia de caracteres com especificador de formato* `%[]`.

```
#include <stdio.h>
int main() {
   char vogal[21];
   printf("Digite uma sequencia de vogais minusculas: ");
   scanf("%[aeiou]",&vogal);
   printf("Vogais digitadas: %s\n",vogal);
}
```

```
Digite uma sequencia de vogais minusculas: aeouiae auie
Vogais digitadas: aeouiae
```

Se o primeiro caractere entre colchetes no especificador de formato `%[]` for o acento circunflexo, tem-se o efeito inverso (negação). Por exemplo, o especificador de formato `%[^aeiou]` produz uma entrada de dados enquanto uma vogal *não* for encontrada. Essa construção permite capturar nomes compostos. Considere o Exemplo 6.9.

Exemplo 6.9. *Programa C para entrada de dados do tipo cadeia de caracteres com especificador de formato* `%[^]`.

```
#include <stdio.h>
int main() {
   char nome[81];
   printf("Digite seu nome completo: ");
   scanf(" %[^\n]",&nome);
   printf("Seu nome: %s\n",nome);
}
```

```
Digite seu nome completo: Wander Gaspar
Seu nome: Wander Gaspar
```

No Exemplo 6.9, o `scanf` lê uma sequência de caracteres enquanto não for encontrado um `'\n'`. Em termos práticos, captura-se a linha fornecida pelo usuário até que se tecle *Enter*. A inclusão do espaço no especificador de formato (antes do sinal `%`) garante que eventuais caracteres em branco que precedam o nome sejam descartados.

Para concluir esse tópico, deve-se salientar que o especificador de formato `%[^\n]` do Exemplo 6.9 pode ser ainda aperfeiçoado. Se o usuário digitar uma sequência com mais de 80 caracteres, estará invadindo um espaço de memória não reservado (o vetor foi dimensionado com 81 elementos). Para evitar essa possível invasão, é possível limitar o número máximo de caracteres que serão capturados incluindo-se essa informação no especificador de formato, conforme apresentado no Exemplo 6.10.

Exemplo 6.10. *Programa C para entrada de dados do tipo string com especificador de formato* `%[^]` *com limitação de caracteres.*

```
#include <stdio.h>
int main() {
   char palavra[10];
   printf("Digite uma palavra (máximo de 9 caracteres): ");
   scanf(" %9[^\n]",&palavra);
   printf("Palavra digitada: %s\n",palavra);
}
```

```
Digite uma palavra (máximo de 9 caracteres): ALGORITMOS
Palavra digitada: ALGORITMO
```

Exemplo 6.11. *Programa C que retorna o número de caracteres em uma* string *fornecida pelo usuário em tempo de execução.*

```
#include <stdio.h>
int main() {
   char nome[81];
   int tam = 0;
   printf("Digite seu nome: ");
   scanf(" %80[^\n]",&nome);
   while(nome[tam] != '\0')
       tam++;
   printf("Numero de caracteres em %s: %d\n",nome,tam);
}
```

```
Digite seu nome: WANDER GASPAR
Numero de caracteres em WANDER GASPAR: 13
```

Exercício 6.1. *Escrever um programa C para substituir todos os espaços em branco de uma frase digitada pelo usuário em tempo de execução por* _. *Exibir na saída padrão a frase antes e depois de substituídos os espaços.*

Exercício 6.2. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que exiba na saída padrão uma palavra por linha.*

Exercício 6.3. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que calcule e apresente na saída padrão a quantidade de vogais.*

Exercício 6.4. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário com todas as letras em minúsculas e que substitua e apresente na saída padrão a frase com a primeira letra de cada palavra substituída por sua equivalente maiúscula.*

Exercício 6.5. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que apresente na saída padrão a frase retirando-se todos os espaços em branco.*

Exercício 6.6. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que apresente na saída padrão a frase no sentido contrário, ou seja, começando pelo último caractere.*

Exercício 6.7. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que apresente na saída padrão o total de palavras digitadas.*

Exercício 6.8. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que apresente na saída padrão a frase "criptografada" conforme a regra:* $ch' = ch + 10$.

Exercício 6.9. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que apresente na saída padrão a frase "descriptografada" de acordo com a regra apresentada no exercício anterior.*

Exercício 6.10. ***Decifre o Código Secreto*** *(junho/2010) Escreva um programa C que solicite ao usuário a digitação de uma frase que deve conter até* 80 *caracteres. O programa deve decifrar o código secreto formado pela primeira letra de cada palavra digitada. O código deve ser exibido na tela em letras maiúsculas. Exemplo de execução:*

```
Frase: Pode remar ou velejar amanha.
Codigo: PROVA
```

6.3 Conversão de Cadeias de Caracteres em Tipos Numéricos

Há situações, como na entrada de dados, em que é importante converter cadeias de caracteres em tipos numéricos. A vantagem da entrada de dados sempre a partir de cadeias de caracteres consiste em evitar que o programa seja terminado caso o usuário digite um valor inválido para o tipo de dados esperado.

A biblioteca padrão `stdlib.h` dispõe de funções para promover a conversão de cadeias de caracteres para tipos numéricos inteiros e de ponto flutuante. Por exemplo, `"123"` pode ser convertido para um tipo inteiro com o valor `123`.

A função `atoi()` retorna um valor inteiro convertido a partir de uma *string* fornecida como argumento de entrada. A conversão é interrompida se um caractere inválido for encontrado. Se nenhum caractere for válido ou se o primeiro caractere for inválido a função retorna o valor `0` (zero). A Tabela 6.1 a seguir apresenta alguns exemplos de conversão de cadeias de caracteres para inteiro.

Tabela 6.1: Exemplos de conversão de *string* para inteiro

String	Inteiro
"157"	157
-1.6"	-1
"+50x"	50
"um"	0
"x505"	0

Exemplo 6.12. *Programa C que que solicita ao usuário a digitação de diversos valores inteiros e positivos e que retorna a soma dos valores válidos. O* flag *é qualquer valor negativo.*

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

```
#include<stdlib.h>
int main() {
    char num[11];
    int val;
    int soma = 0;
    do {
        printf("Inteiro: ");
        scanf(" %10[^\n]",&num);
        val = atoi(num);
        if(val<0)
            break;
        soma+=val;
    } while(1);
    printf("Soma: %d\n",soma);
}

Inteiro: 3
Inteiro: 3.6
Inteiro: -1
Soma: 6
```

A função `atof()` retorna um valor numérico de ponto flutuante convertido a partir de uma *string* fornecida como argumento de entrada. A conversão é interrompida se um caractere inválido for encontrado. Se nenhum caractere for válido ou se o primeiro caractere for inválido a função retorna o valor `0.000000`. A Tabela 6.2 apresenta alguns exemplos de conversão de cadeias de caracteres para `double`.

Tabela 6.2: Exemplos de conversão de *string* para ponto flutuante

String	Ponto Flutuante
"12"	12.000000
-0.123"	-0.123000
"123E+3"	123000.000000
"123.1e-5"	0.001231
"x506"	0.000000

Exemplo 6.13. *Programa C que solicita ao usuário a digitação de diversos valores reais e positivos e que retorna a soma dos valores válidos. O* flag *é qualquer valor negativo.*

```
#include<stdio.h>
#include<stdlib.h>
int main() {
```

```
    char num[16];
    double val;
    double soma = 0;
    do {
        printf("Numero: ");
        scanf(" %15[^\n]",&num);
        val = atof(num);
        if(val<0)
            break;
        soma+=val;
    } while(1);
    printf("Soma: %.2f\n",soma);
}
```

```
Numero: 3
Numero: 3.7
Numero: três
Numero: 1.5e0
Numero: -1
Soma: 8.20
```

Exercício 6.11. ***Soma de Valores*** *(junho/2010) Escreva um programa C que solicite ao usuário a digitação de diversos valores em tempo de execução. A entrada de dados termina quando o usuário digitar um valor que corresponda ao número zero. O objetivo do programa consiste em calcular e exibir a soma dos valores numéricos digitados. Mas cuidado, o problema não é tão simples assim! Observe que o usuário pode digitar caracteres que não correspondam a números, ao ponto decimal ou ao sinal de negativo e que, obviamente, não podem ser somados. Exemplo de execução:*

```
Valor: 10
Valor: -3A
Valor: 6o.
Valor: -1.5$$$
Valor: 0
Soma: 11.50
```

6.4 Funções de C para Cadeias de Caracteres

Uma vez que C manipula cadeias de caracteres a partir de vetores, a maioria dos procedimentos usuais como comparar *strings* e mover o conteúdo de uma *string* para outra requer o tratamento de índices e comandos de laço. Entretanto, como alternativa para a realização desses procedimentos, várias funções úteis para a manipulação de cadeias de caracteres estão disponíveis na biblioteca padrão `string.h`.

6.4.1 Cópia de Cadeias de Caracteres

A função `strcpy(string1,string2)` permite copiar o conteúdo da cadeia de caracteres `string2` para a cadeia de caracteres `string1`. É importante observar que a função não verifica se há espaço suficiente em `string1` para acomodar todo o conteúdo de `string2`. Portanto, esse tipo de verificação de compatibilidade deve ser executado pelo programador.

Exemplo 6.14. *Programa C que copia o conteúdo de uma cadeia de caracteres de uma variável para outra usando a função* `strcpy()`.

```
#include<stdio.h>
#include<string.h>
int main() {
    char string2[]="JUIZ DE FORA";
    char string1[13];
    if(sizeof(string1)>=sizeof(string2)) {
        strcpy(string1,string2);
        printf("%s\n",string1);
    }
    else
        printf("Erro.\n");
}
```

```
JUIZ DE FORA
```

Exemplo 6.15. *Programa C que solicita ao usuário a digitação de uma sequência de* 5 *caracteres e gera uma cadeia de caracteres onde a sequência informada é repetida* 10 *vezes.*

```
#include<stdio.h>
#include<string.h>
int main() {
    int i;
    char entrada[6];
    char saida[51];
    printf("Digite uma sequencia de 5 caracteres: ");
    scanf(" %5[^\n]",&entrada);
    for(i=0;i<10;i++)
        strcpy(saida+i*5,entrada); //Atenção!!
    saida[50] = '\0';
    printf("%s\n",saida);
}
```

```
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCD
```

A função `sizeof()` recebe como argumento de entrada uma variável qualquer e retorna o tamanho (número de *bytes*) alocados na memória para acomodar o conteúdo da variável.

A função `strncpy(string1,string2,n)` permite copiar os n primeiros caracteres da cadeia *string2* para a cadeia de caracteres `string1`. Também a função `strncpy()` não verifica se há espaço suficiente em *string1* para acomodar o conteúdo de `string2` a ser copiado. É importante que a cadeia de caracteres de destino tenha um espaço adicional para acomodar o marcador de fim de *string* (`'\0'`).

Exemplo 6.16. *Programa C que copia parte do conteúdo de uma cadeia de caracteres de uma variável para outra usando a função* ***strncpy()****.*

```
#include<stdio.h>
#include<string.h>
int main() {
    char string2[]="WANDER GASPAR";
    char string1[7];
    strncpy(string1,string2,6);
    printf("%s\n",string1);
}
```

```
WANDER
```

6.4.2 Tamanho de Cadeias de Caracteres

A função `strlen()` recebe como argumento de entrada uma cadeia de caracteres e retorna o tamanho (número de *bytes*) utilizados para acomodar a *string* na memória, sem considerar o espaço para o marcador de fim de *string*.

Observe que o espaço de fato utilizado por uma cadeia de caracteres pode ser menor que o espaço reservado na memória, isto é, eventualmente, o retorno da função `strlen()` pode ser igual ou menor que o retorno da função `sizeof()`. O Exemplo 6.17 ilustra a aplicação da função `strlen()`.

Exemplo 6.17. *Programa C que retorna o número de caracteres em uma* string *fornecida pelo usuário em tempo de execução.*

```
#include <stdio.h>
#include <string.h>
int main() {
   char nome[81];
   printf("Digite seu nome: ");
   scanf(" %80[^\n]",&nome);
   printf("Espaco alocado na memoria......: %d\n",sizeof(nome));
```

```
   printf("Tamanho da cadeia de caracteres: %d\n",strlen(nome));
}

Digite seu nome: WANDER GASPAR
Espaco alocado na memoria......: 81
Tamanho da cadeia de caracteres: 13
```

6.4.3 Concatenação de Cadeias de Caracteres

A função `strcat(string1,string2)` permite concatenar o conteúdo da cadeia de caracteres `string2` para o fim da cadeia de caracteres `string1`. Mais uma vez, é responsabilidade do programador garantir que haja espaço suficiente em `string1` para acomodar os caracteres concatenados. O Exemplo 6.18 ilustra a aplicação da função `strcat()`.

Exemplo 6.18. *Programa C que processa a concatenação de cadeias de caracteres a partir da função* `strcat()`.

```
#include <stdio.h>
#include <string.h>
int main() {
   char prenome[21];
   char sobrenome[21];
   char nome[41]=";
   printf("Digite seu prenome: ");
   scanf(" %20[^\n]",&prenome);
   printf("Digite seu sobrenome: ");
   scanf(" %20[^\n]",&sobrenome);
   strcat(nome,prenome);
   strcat(nome," ");
   strcat(nome,sobrenome);
   printf("Nome completo: %s\n",nome);
}

Digite seu prenome: WANDER
Digite seu sobrenome: GASPAR
Nome completo: WANDER GASPAR
```

A função `strncat(string1,string2,n)` permite concatenar parte do conteúdo da cadeia de caracteres `string2` para o fim da cadeia de caracteres `string1`. Lembre-se que é responsabilidade do programador garantir que haja espaço suficiente em `string1` para acomodar os caracteres concatenados. O Exemplo 6.19 ilustra a aplicação da função `strncat()`.

Exemplo 6.19. *Programa C que processa a concatenação de cadeias de caracteres a partir da função* `strncat()`.

```
#include <stdio.h>
#include <string.h>
int main() {
   char prenome[21];
   char sobrenome[21];
   char nome[41]="";
   printf("Digite seu prenome: ");
   scanf(" %20[^\n]",&prenome);
   printf("Digite seu sobrenome: ");
   scanf(" %20[^\n]",&sobrenome);
   strcat(nome,sobrenome);
   strcat(nome,", ");
   strncat(nome,prenome,1);
   strcat(nome,".");
   printf("Nome: %s\n",nome);
}
```

```
Digite seu prenome: WANDER
Digite seu sobrenome: GASPAR
Nome: GASPAR, W.
```

6.4.4 Comparação de Cadeias de Caracteres

A função `strcmp(string1,string2)` permite comparar o conteúdo de duas cadeias de caracteres `string1` e `string2` e verificar qual delas é maior ou se são iguais. O mecanismo de comparação é empregado caractere a caractere, da esquerda para a direita, conforme apresentado na Figura 6.2.

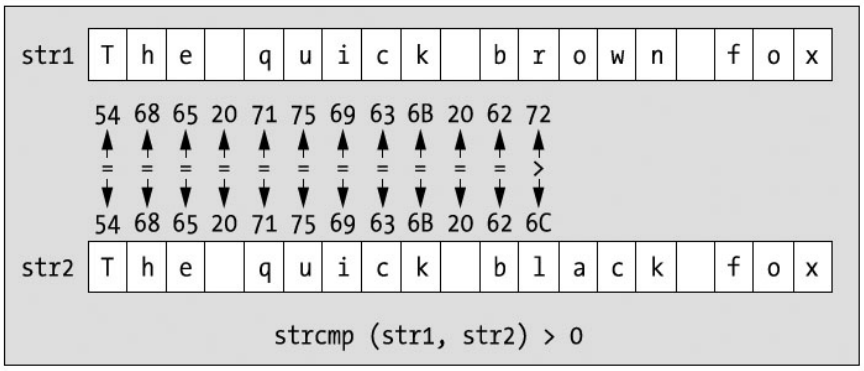

Figura 6.2: Mecanismo de comparação da função `strcmp()`

A função `strcmp(string1,string2)` retorna um número inteiro que pode ser:

- `0` (zero) caso as duas cadeias de caracteres sejam iguais;

- `<0` (menor que zero) caso `string1` seja menor que `string2`;
- `>0` (maior que zero) caso `string2` seja maior que `string2`.

Para ser mais exato, o número inteiro retornado pela função `strcmp(string1, string2)` corresponde à diferença entre os códigos ASCII dos caracteres onde foi detectado a primeira diferença mais à esquerda na comparação. O Exemplo 6.20 ilustra a aplicação da função `strcmp()`.

Exemplo 6.20. *Programa C que processa a comparação de cadeias de caracteres a partir da função* `strcmp()`.

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[6] = "Joao";
   char str2[6] = "Joana";
   printf("Diferenca: %d\n",strcmp(str1,str2));
}
```

```
Diferenca: 1
```

A função `strncmp(string1,string2)` permite comparar os n primeiros caracteres de duas cadeias `string1` e `string2` e verificar qual delas é maior ou se são iguais. O mecanismo de comparação é também empregado caractere a caractere, da esquerda para a direita, conforme apresentado anteriomente na Figura 6.2. O Exemplo 6.21 ilustra a aplicação da função `strcmp()`.

Exemplo 6.21. *Programa C que processa a comparação de cadeias de caracteres a partir da função* `strncmp()`.

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[6] = "Joao";
   char str2[6] = "Joana";
   printf("Diferenca: %d\n",strncmp(str1,str2,3));
}
```

```
Diferenca: 0
```

6.4.5 Pesquisa em uma Cadeia de Caracteres

A função `strchr(string,char)` verifica se um determinado caractere `char` está contido em uma cadeia de caracteres `string`. Caso o caractere esteja contido na cadeia, a função

retorna o endereço de memória do caractere. Em caso contrário é retornado a palavra reservada `NULL`. O Exemplo 6.22 ilustra a aplicação da função `strchr()`.

Exemplo 6.22. *Programa C que pesquisa por uma caractere em uma* string *a partir da função* `strchr()`.

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[6] = "AEIOU";
   char ch = 'I';
   if(strchr(str1,ch)==NULL)
       printf("NAO\n");
   else
       printf("SIM\n");
}
```

```
SIM
```

Caso seja necessário identificar o índice da cadeia onde o caractere foi encontrado, torna-se necessário empregar um recurso da linguagem C denominado ponteiro, que permite armazenar uma posição de memória. O Exemplo 6.23 apresenta um programa C que exibe na saída padrão o índice onde um determinado caractere encontra-se representado em uma cadeia.

Um ponteiro é declarado por um asterisco antecedido ao nome da variável. A palavra reservada `NULL` significa que nenhum endereço de memória está armazenado no ponteiro. Para maiores detalhes sobre ponteiros em C consulte um livro avançado sobre a linguagem.

Exemplo 6.23. *Programa C que exibe o índice de uma cadeia onde se encontra um caractere pesquisado a partir da função* `strchr()`.

```
#include <stdio.h>
#include <string.h>
int main() {
   char str[6] = "AEIOU";
   char ch = 'I';
   char *pos = NULL;
   pos = strchr(str,ch);
   printf("%d\n",pos-str);
}
```

```
2
```

A função `strstr(string1,str2)` verifica se uma determinada cadeia de caracteres `str2` está contida em uma outra cadeia `string1`. Em caso positivo a função retorna o

endereço de memória do primeiro caractere da cadeia procurada. Em caso contrário é retornado a palavra reservada `NULL`. O Exemplo 6.24 ilustra a aplicação da função `strstr()`.

Exemplo 6.24. *Programa C que exibe o índice do primeiro caractere de uma cadeia pesquisada a partir da função* `strstr()`.

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[6] = "AEIOU";
   char str2[3] = "OU";
   char *pos = NULL;
   pos = strstr(str1,str2);
   printf("%d\n",pos-str1);
}
```

```
3
```

Exercício 6.12. *Escrever um programa C que receba na entrada padrão duas palavras digitadas pelo usuário. O programa deve verificar se as palavras são iguais ou diferentes, independente dos caracteres estarem em letras minúsculas ou maiúsculas.*

Exercício 6.13. *Escrever um programa C que receba na entrada padrão uma frase digitada pelo usuário e que apresente na saída padrão a quantidade de vezes que uma determinada palavra — também fornecida pelo usuário — se encontra presente na frase.*

Exercício 6.14. *Escrever um programa C que receba na entrada padrão uma sequência de 5 letras maiúsculas. O programa deve retornar o número de vezes necessários para que seja gerado uma sequência aleatória idêntica.*

6.5 Vetor de Cadeias de Caracteres

Em muitas aplicações é desejável representar um vetor de cadeia de caracteres. Por exemplo, pode-se considerar uma aplicação que necessite armazenar os nomes de todos os alunos de uma turma em um vetor.

Uma cadeia de caracteres é representada por um vetor do tipo `char`. Para representar um vetor no qual cada elemento consiste em uma cadeia de caracteres é necessário ter um *array* bidimensional de `char`. Assumindo que nenhum nome de aluno terá mais de 80 caracteres e que o número máximo de alunos em uma turma é 50, é possível declarar uma matriz para armazenar os nomes dos alunos com a seguinte sintaxe:

```
char alunos[50][81];
```

Com essa variável declarada, `a1unos[i]` acessa a cadeia de caracteres com o nome do (i+1)-ésimo aluno da turma e, consequentemente, `alunos[i][j]` acessa a (j+1)-ésima letra do nome referenciado. Os Exemplos 6.25 e 6.26 ilustram a aplicação de vetores de cadeias de caracteres.

Exemplo 6.25. *Programa C que gera e exibe na saída padrão um vetor de* strings*, cujo conteúdo é fornecido pelo usuário em tempo de execução.*

```
#include <stdio.h>
int main() {
   char str[5][81];
   int i;
   for(i = 0; i < 5; i++) {
       printf("Digite seu nome: ");
       scanf(" %80[^\n]",str[i]);
   }
   for(i = 0; i < 5; i++)
       printf("%s\n",str[i]);
}
```

Exemplo 6.26. *Programa C que gera e exibe na saída padrão um vetor de* strings *e que verifica se uma determinada cadeia de caracteres informada pelo usuário na entrada padrão está contida no vetor gerado.*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
int main() {
   srand(time(0));
   char str[20][5];
   char s[5];
   int i, j;
   for(i = 0; i < 20; i++) {
       for(j = 0; j < 4; j++)
           str[i][j] = rand()%26+65;
       str[i][j] = '\0';
   }
   for(i = 0; i < 20; i++)
       printf("%2d %s\n",i,str[i]);
   printf("Sequencia de 4 letras maiusculas: ");
   scanf(" %4[^\n]",s);
   for(i = 0; i < 20; i++)
       if(strcmp(str[i],s) == 0)
```

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

Exercício 6.15. *Escrever um programa C que receba na entrada padrão o nome e as notas de uma turma de N alunos, onde N é fornecido pelo usuário. As notas devem estar no intervalo* $[0, 100]$*. Os nomes devem possuir até* 30 *caracteres. Exibir na saída padrão a relação de nomes e notas classificados em ordem decrescente de nota.*

Exercício 6.16. ***Gosto Não Se Discute*** *(junho/2010) Seu objetivo nessa questão consiste em gerar e exibir uma frase aleatória em uma única linha na tela. A frase deve conter três elementos: sujeito, verbo e predicado, nessa ordem. O sujeito deve ser escolhido randomicamente pelo programa entre* 4 *opções possíveis: Pedro, Maria, Carlos e Ana. Analogamente, o verbo deve ser escolhido aleatoriamente pelo programa entre gosta de, não gosta de, adora e detesta. Por fim, o predicado deve ser escolhido randomicamente pelo programa entre feijoada, futebol, samba e feriado. Exemplo de execução:*

```
Ana detesta feriado.
```

Exercício 6.17. ***Quem Ganhou A Rifa?*** *(junho/2010) Escreva um programa que simule o sorteio de uma rifa. Cada número no intervalo* $[1, 10]$ *deve ser associado a um nome. Os nomes e respectivos números devem ser exibidos na tela. Em seguida, o programa deve sortear aleatoriamente um número e apresentar o nome do respectivo vencedor na tela. Os* 10 *(dez) nomes podem ser digitados pelo usuário em tempo de execução ou inseridos diretamente no código fonte do programa. Considere cada nome com até* 10 *caracteres. Exemplo de execução:*

```
[1] JOAO [2] ROSA [3] PEDRO [4] MARIA [5] ANTONIO
[6] LUCAS [7] RITA [8] CARLOS [9] CLARA [10] MANUEL
Sorteio: 7
Vencedor: RITA
```

Capítulo 7

Funções

A mente que se abre a uma nova ideia
jamais voltara ao seu tamanho original – Albert Einstein

Para a construção de programas estruturados, é sempre preferível dividir as grandes tarefas de computação em tarefas menores e utilizar seus resultados parciais para compor o resultado final desejado (CELES *et al.*, 2004).

7.1 Modularização

Modularização é uma técnica de desenvolvimento de software que consiste em dividir um problema em diversas partes ou módulos. Em C, os módulos são implementados através do uso de **funções**.

Alguns benefícios dessa estratégia:

- Evita que uma sequência de comandos necessária em várias partes do software tenha que ser reescrita;
- Divide o software em partes logicamente coerentes;
- Aumenta a legibilidade do código-fonte;
- Facilita a manutenção do software.

Exemplo 7.1. *Função que calcula e retorna o volume de um cilindro. Os argumentos são o raio da base e a altura do cilindro.*

```
// cabeçalho da função:
float volumeCilindro(float r, float h) {
// corpo da função:
   const float PI = 3.1415926536;
   return (PI * r * r * h);
}
```

A definição de uma função consiste de um **cabeçalho** (ou declaração) e de um bloco de comandos. A declaração da função deve especificar, nesta ordem:

- o tipo do valor de retorno (ou argumento de saída);
- o nome da função;
- os tipos e nomes dos argumentos de entrada, especificados entre parênteses e separados por vírgulas.

Deve existir também pelo menos um comando `return` no corpo de uma função. O término da execução ocorre ao se alcançar o comando `return`. Quando isso acontece, o fluxo da execução retorna para o ponto do programa onde a função foi invocada.

Na realidade, a execução de um programa C sempre inicia a partir de uma função principal, denominada `main()`. Portanto, o código-fonte C, para ser executado após a compilação, deve conter uma, e somente uma função `main()`. Esta função principal, por sua vez, pode chamar (ou invocar), uma ou mais funções, conforme apresentado no Exemplo 7.2).

Exemplo 7.2. *Programa C que solicita ao usuário a digitação na entrada padrão do raio da base e da altura de um cilindro e que invoca a função que calcula e retorna o volume do cilindro.*

```
#include <stdio.h>
float volumeCilindro(float, float);
int main() {
   float raio;
   float altura;
   float volume;
   printf("Informe o raio da base do cilindro:");
   scanf("%f",&raio);
   printf("Informe a altura do cilindro:");
   scanf("%f",&altura);
   volume = volumeCilindro(raio,altura); // chama a função.
   printf("Volume do cilindro: %.2f",volume);
}
float volumeCilindro(float r, float h) {
   const float PI = 3.1415926536;
```

```
   return (PI * r * r * h);
}
```

Observe que, antes do cabeçalho da função `main()`, é necessário incluir o **protótipo** das funções invocadas. O protótipo de uma função é semelhante à declaração da função porém, não é necessário especificar os nomes dos argumentos, apenas os tipos de dados.

Como ocorre a execução de um programa C contendo funções? A execução inicia-se pela função `main()`. Quando uma função é invocada, a execução é transferida e retorna após o comando `return` ser alcançado. O fim da execução do programa acontece após a execução do comando `return` da função principal `main()` (Figura 7.1).

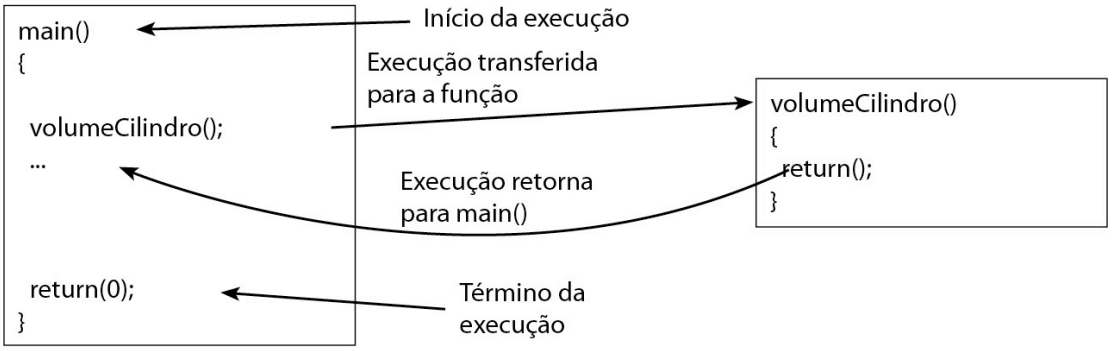

Figura 7.1: Encadeamento entre funções durante a execução de um programa C

É importante observar que não é necessário especificar o protótipo caso a função seja declarada antes da função principal `main()` (Exemplo 7.3). Entretanto, o emprego de protótipos é recomendado por tratar-se de uma boa prática de programação.

Exemplo 7.3. *Programa C que solicita ao usuário a digitação na entrada padrão do raio da base e da altura de um cilindro e que invoca a função que calcula e retorna o volume do cilindro (versão sem protótipo).*

```
#include <stdio.h>
float volumeCilindro(float r, float h) {
   const float PI = 3.1415926536;
   return (PI * r * r * h);
}
int main() {
   float raio;
   float altura;
   float volume;
   printf("Informe o raio da base do cilindro:");
   scanf("%f",&raio);
   printf("Informe a altura do cilindro:");
   scanf("%f",&altura);
   volume = volumeCilindro(raio,altura);
   printf("Volume do cilindro: %.2f",volume);
}
```

7.2 Argumentos de Entrada e Tipos de Retorno

Uma função em C pode ter um ou mais **argumentos de entrada**, fornecidos entre parênteses e separados por vírgulas, na declaração da própria função. Também é possível declarar uma função que não tenha argumento de entrada. Nesse caso, o argumento é declarado como `void`.

O **tipo de retorno** ou argumento de saída de uma função determina o tipo de dado que é passado para o encadeamento do programa quando a função termina a execução. O tipo de retorno pode ser, inicialmente, qualquer um dos tipos básicos da linguagem (`int`, `float`, `double`, `char`, etc.) ou `void`, caso a função não retorne nenhum valor.

O Exemplo 7.4 apresenta uma função declarada pelo usuário cujo tipo de retorno é `void` e cujo argumento de entrada é também `void`. Isso significa que a função não recebe nenhum argumento de entrada e também não retorna nenhum valor ao fim da execução.

Exemplo 7.4. *Função para exibir uma linha de asteriscos na saída padrão.*

```
#include <stdio.h>
void linha(void); // Protótipo da função.
int main() {
   linha(); // Invocação da Função
   printf("CENTRO DE ENSINO SUPERIOR DE JUIZ DE FORA\n");
   linha();
}
void linha(void) { // Declaração da função.
   int i;
```

```
   for(i=0;i<80;i++)
      printf("*");
   printf("\n");
}

**********************************************************************
CENTRO DE ENSINO SUPERIOR DE JUIZ DE FORA
**********************************************************************
```

O Exemplo 7.5 apresenta uma função declarada pelo usuário cujo tipo de retorno é `void` e que possui um único argumento de entrada cujo tipo é inteiro. Observe que o argumento de entrada é usado no corpo da função para controlar o número de asteriscos a serem exibidos na saída padrão.

Em uma função do tipo `void` é opcional a inclusão do comando `return` ao final do corpo da função.

Exemplo 7.5. *Função para exibir uma linha com um número determinado de asteriscos na saída padrão.*

```
#include <stdio.h>
void linha(int);
int main() {
   int num;
   printf("Numero de asteriscos: ");
   scanf("%d",&num);
   linha(num);
   printf("CENTRO DE ENSINO SUPERIOR DE JUIZ DE FORA\n");
   linha(num);
}
void linha(int n) {
   int i;
   for(i=0;i<n;i++)
      printf("*");
   printf("\n");
}

Numero de asteriscos: 41
*****************************************
CENTRO DE ENSINO SUPERIOR DE JUIZ DE FORA
*****************************************
```

Pode-se observar a partir do 7.5 que o nome do argumento na função não necessariamente deve ser igual ao nome da variável na chamada da função.

Exercício 7.1. *Escreva uma função que apresente na saída padrão uma frase informada pelo usuário na entrada padrão. A saída deve ser centralizada na linha. Considere que a frase digitada deve ter, no máximo,* 80 *caracteres.*

O Exemplo 7.6 apresenta uma função cujo tipo de retorno é `void` e que possui dois argumentos de entrada. O primeiro argumento é `int` e o segundo é `char`. Quando há mais de um argumento de entrada, a invocação da função deve respeitar rigorosamente o número e o tipo de argumentos e também a ordem entre eles.

Exemplo 7.6. *Função para exibir uma linha com um número determinado de caracteres na saída padrão. O caractere é informado pelo usuário na entrada padrão.*

```
#include <stdio.h>
void linha(int,char);
int main() {
   int num;
   char car;
   printf("Numero de caracteres: ");
   scanf("%d",&num);
   printf("Caractere: ");
   scanf(" %c",&car);
   linha(num,car);
   printf("CENTRO DE ENSINO SUPERIOR DE JUIZ DE FORA\n");
   linha(num,car);
}
void linha(int n, char ch) {
   int i;
   for(i=0;i<n;i++)
      printf("%c",ch);
   printf("\n");
}
```

```
Numero de caracteres: 41
Caractere: -
-----------------------------------------
CENTRO DE ENSINO SUPERIOR DE JUIZ DE FORA
-----------------------------------------
```

O Exemplo 7.7 apresenta uma função cujo tipo de retorno é `int` e que possui dois argumentos de entrada, ambos do tipo `int`. Observe que o valor retornado, do tipo inteiro, é compatível com o tipo de retorno esperado ao fim da execução da função.

Exemplo 7.7. *Define uma função cujo valor de retorno é o resto da divisão de dois números inteiros fornecidos como argumentos de entrada.*

```
#include <stdio.h>
int restoDivisao(int, int);
int main() {
   int num1, num2, resto;
   printf("Digite o primeiro numero: ");
   scanf("%d",&num1);
   printf("Digite o segundo numero: ");
   scanf("%d",&num2);
   resto = restoDivisao(num1,num2);
   printf("Resto da divisao de %d por %d: %d\n",num1,num2,resto);
}
int restoDivisao(int n1, int n2) {
   return (n1 % n2);
}
```

```
Digite o primeiro numero: 13
Digite o segundo numero: 3
Resto da divisao de 13 por 3: 1
```

Exercício 7.2. *Escreva uma função que calcule e retorne a soma da sequência* $1, 2, \ldots, N$*, onde* N *é um número inteiro fornecido como argumento de entrada.*

Exercício 7.3. *Escreva uma função que calcule e retorne a média aritmética de* 3 *números reais fornecidos como argumentos de entrada.*

Exercício 7.4. *Escreva uma função que calcule e retorne a média ponderada de* 3 *números inteiros fornecidos como argumentos de entrada. Os pesos devem ser* 1, 2 *e* 3 *respectivamente para o primeiro, segundo e terceiro números.*

Exercício 7.5. *Escreva uma função que calcule e retorne a média ponderada de* 3 *números inteiros fornecidos como argumentos de entrada. Os pesos devem também ser fornecidos como argumentos de entrada.*

Exercício 7.6. *Escreva uma função que receba como argumento de entrada um número inteiro no intervalo* $[1, 32767]$ *e que retorne o número invertido. Por exemplo, se a entrada for o número* 347 *a saída deve ser* 743*. Caso seja informado um argumento de entrada inválido, o retorno deve ser* 0*.*

Exercício 7.7. *Escreva uma função que receba como argumento de entrada um número inteiro no intervalo* $[1, 10]$ *e que retorne o fatorial do número. Caso seja informado um argumento de entrada inválido, o retorno deve ser* 0*.*

7.3 Funções e Estruturas Homogêneas

A passagem de argumentos entre funções pode ser realizada de duas maneiras, por valor ou por referência. Todos os exemplos apresentados nas seções anteriores utilizaram a

abordagem de **passagem por valor**. Isto significa que os argumentos de entrada e saída devem ser explicitamente indicados na declaração da função.

Em geral, ao trabalhar-se com os tipos primitivos da linguagem, utiliza-se a passagem de argumentos por valor.

Na **passagem por referência**, ao contrário, os argumentos são implicitamente referenciados pelas funções. A passagem de argumentos do tipo estruturas homogêneas, como um vetor ou uma matriz) sempre deve ser feita por referência (Exemplo 7.8).

Exemplo 7.8. *Define duas funções para tratamento de vetores. A primeira inicializa a estrutura e a segunda exibe o conteúdo na saída padrão.*

```
#include <stdio.h>
#define SIZE 10
void inicializa(int[]);
void imprime(int[]);
int main() {
   int vet[SIZE];
   inicializa(vet);
   imprime(vet);
}
void inicializa(int v[]) {
   int i;
   for(i = 0; i < SIZE; i++)
      v[i] = i;
}
void imprime(int v[]) {
   int i;
   for(i = 0; i < SIZE; i++)
      printf("%2d",v[i]);
   printf("\n");
}
```

```
 0 1 2 3 4 5 6 7 8 9
```

Como funciona a passagem por referência? Exatamente como sugere o nome dessa abordagem. É criada uma referência da estrutura de dados criada em `main()` e passada para as funções declaradas. Assim, estas funções acessam e manipulam uma única área na memória, definida na função principal do programa, onde encontra-se armazenados os elementos do vetor.

Uma vez que a estrutura não foi passada por valor, torna-se desnecessário um argumento de retorno na passagem por referência, que sempre será do tipo `void`.

Observe no Exemplo 7.8 que existe um vínculo indesejável entre as funções declaradas e a diretiva `#define`. O problema, nesse caso, é que as funções precisam "saber"

que SIZE representa o tamanho do vetor. Uma solução mais interessante é apresentada no Exemplo 7.9, onde o tamanho da estrutura de dados é passada como argumento de entrada para as funções declaradas.

Exemplo 7.9. *Define duas funções para tratamento de vetores. A primeira inicializa a estrutura e a segunda exibe o conteúdo na saída padrão (versão com passagem do tamanho do vetor como argumento de entrada).*

```
#include <stdio.h>
#define SIZE 10
void inicializa(int,int[]);
void imprime(int,int[]);
int main() {
   int vet[SIZE];
   inicializa(SIZE,vet);
   imprime(SIZE,vet);
}
void inicializa(int t,int v[]) {
   int i;
   for(i = 0; i < t; i++)
      v[i] = i;
}
void imprime(int t,int v[]) {
   int i;
   for(i = 0; i < t; i++)
      printf("%2d",v[i]);
   printf("\n");
}
```

Exercício 7.8. *Escreva uma função que receba como argumento de entrada um vetor de tamanho N, onde N também é passado como argumento de entrada. A função deve retornar a média aritmética dos elementos do vetor. Escreva também uma função para inicializar e exibir na saída padrão um vetor com valores aleatórios no intervalo* $[A, B]$*, onde* A *e* B *são números inteiros quaisquer e* $B \geq A$*. Escreva uma aplicação para gerar um vetor de tamanho* 100 *e exibir na saída padrão a média aritmética do conteúdo.*

Exercício 7.9. *Ainda considerando-se o exercício anterior, defina uma função para retornar o maior elemento do vetor.*

Exercício 7.10. *Escrever uma função para inicializar um vetor de caracteres com valores no intervalo* $[A, Z]$*. O tamanho do vetor deve ser passado como o argumento de entrada. Escrever uma outra função que retorne a quantidade de vogais no vetor. Escrever uma aplicação para gerar um vetor de tamanho* 100 *e exibir na saída padrão quantidade de vogais presentes no vetor.*

Exercício 7.11. *Escreva uma função para inicializar uma matriz quadrada de ordem N com valores inteiros gerados randomicamente no intervalo $[A, B]$, onde $B \geq A$. O valor de N deve ser passado como argumento de entrada para a função. Escreva também uma função para exibir na saída padrão uma matriz quadrada de ordem N, onde N é passado como argumento de entrada. Por fim, escreva uma função para calcular a matriz $C = A + B$, onde A e B são matrizes quadradas de ordem N. Escreva uma aplicação que declare e inicialize 2 matrizes quadradas de ordem 20 com valores aleatórios no intervalo $[-9, 9]$. A aplicação deve calcular $C = A + B$ e exibir a matriz C na saída padrão.*

Exercício 7.12. *Escreva uma função que receba 2 vetores A e B de inteiros de tamanho N como argumentos de entrada. A função deve calcular $B = \alpha \cdot A$, onde α é um número inteiro também passado como argumento para a função.*

7.4 Funções e Cadeias de Caracteres

Uma vez que cadeias de caracteres em C são manipuladas na forma de vetores do tipo `char`, o emprego de passagem por referência para os argumentos de funções também deve ser observado ao se tratar *strings*.

O Exemplo 7.10 apresenta uma função que recebe como argumento de entrada a passagem por referência de uma cadeia de caracteres.

Exemplo 7.10. *Aplicação que chama uma função para converter os caracteres de uma string para letras maiúsculas.*

```
#include <stdio.h>
#include <string.h>
void str(char[]);
int main() {
    char p[] = "Teste";
    str(p);
    printf("%s\n",p);
}
void str(char ch[]) {
    int i;
    for (i = 0; ch[i]!='\0'; i++)
        if(ch[i]>='a'&&ch[i]<='z')
            ch[i]-=32;
}
```

```
TESTE
```

Exercício 7.13. *Escrever uma função que receba como argumento de entrada uma string e que retorne a cadeia de caracteres "criptografada" conforme a regra: $ch' = ch + 10$. Escreva também uma outra função para descriptografar a string.*

Exercício 7.14. *Escrever uma função que receba como argumento de entrada uma string, que deve ser modificada de forma que cada palavra se inicie por letra maiúscula e as demais letras sejam grafadas em letras minúsculas.*

Exercício 7.15. *Escrever uma função que receba como argumento de entrada uma string e que retorne o número de palavras existentes na cadeia de caracteres.*

7.5 Tipos Especiais de Variáveis

7.5.1 Variáveis Globais

Uma variável pode ser declarada fora do corpo de uma função. Este tipo especial de variável é denominada **global**. Uma variável global possui a característica de ser visível a todas as funções que se seguem à sua declaração.

Outro aspecto importante é que uma variável global permanece alocada na memória enquanto o programa estiver em execução. Assim, seu conteúdo permanece disponível até o término da aplicação.

O Exemplo 7.11 exibe uma função que emprega duas variáveis globais do tipo inteiro.

Exemplo 7.11. *Função para calcular e retornar a soma e o produto de dois números inteiros atribuídos a variáveis globais.*

```
#include <stdio.h>
void somaprod(int, int);
int s, p;
int main() {
    somaprod(5,10);
    printf("Soma = %d\nProduto = %d\n", s, p);
}
void somaprod(int a, int b) {
    s = a + b;
    p = a * b;
}
```

```
Soma = 15
Produto = 50
```

O uso de variáveis globais apresenta vantagens e desvantagens. Conforme apresentado até o momento, uma função pode retornar apenas um único valor de um dos tipos de dados primitivos da linguagem através do comando `return`. O uso de variáveis globais contorna essa limitação ao permitir a manipulação direta de variáveis fora do corpo da função.

Como desvantagem, o emprego de variáveis globais aumenta a interdependência entre as funções utilizadas em um programa, dificultando a reutilização de código.

Exercício 7.16. *Escrever uma função para processar o swap (troca de valores) entre duas variáveis globais do tipo* `float`.

Exercício 7.17. *Escrever um programa C para exibir o maior e o menor valor contido em um vetor de inteiros de tamanho 100. O programa deve possuir os seguintes módulos ou funções:*

- *função para atribuição de valores randômicos ao vetor, no intervalo* $[0, N]$*, onde* N *é passado como argumento de entrada.*
- *função para exibição do vetor na saída padrão, com* M *elementos por linha, onde* M *é passado como argumento de entrada.*
- *função para calcular o maior e o menor elemento do vetor (use variáveis globais).*

7.5.2 Variáveis Estáticas

Uma **variável estática** deve ser declarada no corpo de uma função. Porém, seu conteúdo permanece disponível mesmo após o término da execução da função. Na declaração, uma variável estática deve conter o modificador `static`.

Variáveis estáticas compartilham algumas características tanto de variáveis locais (só podem ser acessadas no corpo da função que a declarou) quanto de variáveis globais (têm o seu conteúdo disponível mesmo após a execução da função).

Uma utilização importante de variáveis estáticas consiste em recuperar o valor de uma variável atribuída na última vez em que a função foi executada (Exemplo 7.12).

Exemplo 7.12. *Função que exibe a aplicabilidade de uma variável estática.*

```
#include <stdio.h>
void estatica(void); // Protótipo da função estática
int main() {
    int i;
    for (i = 1; i <= 5; i++)
        estatica();
}
void estatica(void) {
  int i = 0;
  static int j = 0; // declara e atribui valor a uma variável estática
  printf("Local = %d\tEstatica = %d\n", i++, j++);
}
```

```
Local = 0       Estatica = 0
```

```
Local = 0       Estatica = 1
Local = 0       Estatica = 2
Local = 0       Estatica = 3
Local = 0       Estatica = 4
```

7.6 Recursividade

Funções em C podem ser usadas de forma recursiva. Isto significa que uma função pode chamar a si própria. Em uma analogia com o mundo real, é como se você procurasse no dicionário a definição da palavra recursão e encontrasse a seguinte definição:

Recursão: (s.f.) Veja a definição em *recursão*.

Um exemplo de função que pode ser escrita com chamadas recursivas é o fatorial de um número inteiro não-negativo N, representado por $N!$ e que representa a seguinte definição:

$$0! = 1 \quad \text{e} \quad n! = (n-1)! * n \quad \text{para} \quad n > 0$$

Assim, $1! = 0! \times 1 = 1 \times 1 = 1$, $2! = 1! \times 2 = 1 \times 2 = 2$ e $3! = 2! \times 2 = 2 \times 3 = 6$.

É possível resolver o problema do fatorial sem o emprego de recursividade, conforme apresentado no Exemplo 7.13, que define uma função para esse fim.

Exemplo 7.13. *Função para o cálculo do fatorial de um número inteiro não-negativo.*

```
#include <stdio.h>
int fatorial(int);
int main() {
    int num;
    int fat;
    printf("Numero inteiro nao-negativo: ");
    scanf("%d",&num);
    fat = fatorial(num);
    printf("Fatorial: %d\n",fat);
}
int fatorial(int n) {
    int i;
    int fat = 1;
    for(i=1;i<=n;i++)
       fat*=i;
    return fat;
}
```

```
Numero inteiro nao-negativo: 6
Fatorial: 720
```

A partir da definição do fatorial apresentada anteriormente, pode-se pensar em uma solução recursiva para o problema, onde o fatorial de um número qualquer é igual ao próprio número multiplicado pelo fatorial do seu predecessor. O Exemplo 7.14 apresenta uma função recursiva para o cálculo do fatorial.

Exemplo 7.14. *Função recursiva para o cálculo do fatorial de um número inteiro não-negativo.*

```
#include <stdio.h>
int fatrec(int);
int main() {
    int num;
    int fat;
    printf("Numero inteiro nao-negativo: ");
    scanf("%d",&num);
    fat = fatrec(num);
    printf("Fatorial: %d\n",fat);
}
int fatrec(int n) {
    if(n==0)
       return 1;
    else
       return n * fatrec(n-1);
}
```

```
Numero inteiro nao-negativo: 6
Fatorial: 720
```

É importante observar que toda função recursiva deve prever como o processo de recursão será interrompido. No Exemplo 7.14, o processo termina quando o valor do número passado como argumento passa a valer 0. Se este teste não for incluído na função, as chamadas continuariam indefinidamente com valores negativos cada vez menores sendo passados como argumento de entrada.

Exercício 7.18. *Escrever uma função recursiva para calcular o valor de x^n, onde $n > 0$, onde x e n são passados como argumentos de entrada.*

7.7 Bibliotecas de Funções

Os compiladores C permitem que um programa utilize funções armazenadas em uma biblioteca do usuário. Nesse caso, a diretiva ao pré-processador `#include` deve ser utilizada

para informar ao programa o nome da biblioteca onde se encontra a função.

Além disso, no caso do Dev-C++, deve ser incluído no IDE o caminho completo do diretório onde se encontra a biblioteca de funções do usuário. Este requisito pode ser feito, a partir do menu principal, selecionando-se `Ferramentas` → `Opções do Compilador` → `Diretórios` → `C includes`. Na parte inferior direita da janela há um botão que deve ser usado para localizar a pasta na árvore de diretórios do computador.

A vantagem do uso de funções previamente definidas está diretamente relacionada à modularização e ao reaproveitamento de código-fonte.

O Exemplo 7.15 exibe uma função armazenada em uma biblioteca do usuário denominada `wander.h`.

Exemplo 7.15. *Função recursiva para o cálculo do fatorial.*

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

Uma vez armazenada na biblioteca do usuário e configurado o IDE para a busca no diretório determinado, é possível implementar o programa-fonte para acessar a função previamente definida.

O Exemplo 7.16 apresenta um programa C para calcular o fatorial de um número inteiro não-negativo a partir de uma função recursiva armazenada em uma biblioteca do usuário.

Exemplo 7.16. *Programa C para cálculo do fatorial.*

```
#include <stdio.h>
#include "wander.h"
int fatrec(int);
int main() {
    int num;
    int fat;
    printf("Numero inteiro nao-negativo: ");
    scanf("%d",&num);
    fat = fatrec(num);
    printf("Fatorial: %d\n",fat);
}
```

Observe no Exemplo 7.16 que uma biblioteca do usuário é delimitada por aspas duplas na diretiva ao pré-processador `#include`.

Exercício 7.19. *Criar uma biblioteca do usuário própria e nomeá-la como* `meunome.h`. *Incluir na biblioteca duas ou mais funções definidas nesse capítulo. Escrever um programa C que chame as funções armazenadas na biblioteca do usuário.*

Capítulo 8

Estruturas Heterogêneas

A mente que se abre a uma nova ideia
jamais voltara ao seu tamanho original – Albert Einstein

Nos capítulos anteriores foram apresentadas as estruturas de dados homogêneas: vetores, matrizes e strings. Nas estruturas homogêneas todos os elementos são de tipos de dados primitivos: inteiro, real, caractere.

No entanto, em muitos casos, é necessário armazenar um conjunto de informações relacionadas, formado por diversos tipos de dado primitivos. Exemplos comuns do dia-a-dia são endereços de clientes ou funcionários, dados de produtos ou serviços de uma empresa, etc.

8.1 Variáveis Compostas Heterogêneas

Quando uma determinada estrutura de dados for composta por diversos tipos diferentes, primitivos ou não, tem-se um conjunto heterogêneo de dados chamados de **variáveis compostas heterogêneas**. Estas variáveis compostas são chamadas de estruturas ou *structs* em C.

Uma **estrutura** pode ser definida como uma coleção de diversas variáveis relacionadas entre si, onde cada variável pode ser de um tipo primitivo distinto. A sintaxe de uma estrutura em linguagem C é:

```
struct <nome_estrutura>
{
    tipo_1 IDENTIFICADOR_1;
    tipo_2 IDENTIFICADOR_2;
    ...
    tipo_n IDENTIFICADOR_n;
};
```

Importante: a definição de uma estrutura em C deve ser feita fora do programa principal ou de qualquer função do usuário existente.

Em C, após a definição de uma estrutura, é necessário criar uma variável de memória e associá-la ao tipo de dado composto que foi definido. Isto é feito através do comando struct. Observe que este comando deve ser inserido no corpo do programa principal ou de alguma função do usuário.

Exemplo 8.1. *Escrever um programa C para criar uma estrutura chamada* `Ponto`*, contendo os membros* `x` *e* `y` *do tipo inteiro, cujos valores deverão ser fornecidos pelo usuário em tempo de execução. Apresentar na tela o conteúdo da estrutura.*

```
#include <stdio.h>
struct ponto {
    int x;
    int y;
};

int main() {
    struct Ponto ponto;
    printf("Digite as coordenadas x e y do ponto: ");
    scanf("%d%d",&ponto.x,&ponto.y);
    printf("\nCoordenadas: %d %d\n",ponto.x,ponto.y);
}
```

Campos ou membros de uma estrutura podem ser usados da mesma forma como as demais variáveis em C. Campos são acessados usando o operador de acesso (`.`), que deve ser inserido entre os nomes da estrutura e do campo.

Exemplo 8.2. *Escrever um programa C para criar uma estrutura denominada* `Empregado`*, contendo os membros matrícula, salário e nome, respectivamente do tipo* `int`*,* `float` *e* `char[21]`*, cujos valores serão fornecidos pelo usuário na entrada padrão em tempo de execução. Apresentar na saída padrão o conteúdo da estrutura.*

```
#include <stdio.h>
struct Empregado {
    int matricula;
    float salario;
    char nome[21];
};

int main(){
    struct Empregado emp;
    printf("Matricula: ");
    scanf("%d",&emp.matricula);
```

```
    printf("Salario..: ");
    scanf("%f",&emp.salario);
    printf("Nome.....: ");
    scanf("%20[^\n]",emp.nome);
    printf("\nEmpregado\nMatricula: %d",emp.matricula);
    printf("\nSalario: %.2f\nNome: %s\n",emp.salario,emp.nome);
}
```

8.2 Arranjo de Estruturas Heterogêneas

Exemplo 8.3. *Escrever um programa C para criar um vetor de tamanho 5 de* `Empregado`*, cujos valores serão fornecidos pelo usuário em tempo de execução. Apresentar na saída padrão o conteúdo do vetor.*

```
#include <stdio.h>
#define SIZE 5
struct Empregado {
    int matricula;
    float salario;
    char nome[21];
};

int main(){
    struct Empregado emp[SIZE];
    int i;
    for(i=0;i<SIZE;i++) {
        printf("\nEmpregado no. %d\nMatricula: ",(i+1));
        scanf("%d",&emp[i].matricula);
        printf("Salario..: ");
        scanf("%f",&emp[i].salario);
        printf("Nome.....: ");
        scanf(" %20[^\n]",&emp[i].nome);
    }
    printf("\nListagem de empregados\n");
    for(i=0;i<SIZE;i++) {
        printf("\nEmpreg. no. %d\nMatricula: ",(i+1),emp[i].matricula);
        printf("\nSalario: %.2f\nNome: %s\n",emp[i].salario,emp[i].nome);
    }
}
```

8.3 Funções e Estruturas Heterogêneas

Estruturas (structs)podem ser passadas como argumentos em funções. Também podem ser o tipo de dado de retorno em funções.

Exemplo 8.4. *Escrever um programa C para criar uma estrutura* `Ponto`*, contendo os membros* `x` *e* `y` *do tipo inteiro. Escrever uma função para solicitar ao usuário em tempo de execução a entrada de dados para a estrutura. Escrever uma função para apresentar na saída padrão o conteúdo da estrutura.*

```
#include <stdio.h>
struct Ponto {
    int x;
    int y;
};

struct Ponto entradaPonto(void);
void saidaPonto(struct Ponto);

int main(){
    struct Ponto pt;
    pt = entradaPonto();
    saidaPonto(pt);
}
struct Ponto entradaPonto(void) {
    struct Ponto pt;
    printf("Coordenadas x e y do ponto: ");
    scanf("%d%d",&pt.x,&pt.y);
    return pt;
}
void saidaPonto(struct Ponto pt) {
    printf("\nCoordenadas: %d e %d\n",pt.x,pt.y);
}
```

Exemplo 8.5. *Escrever um programa C para criar um vetor de estruturas* `Ponto`*, de tamanho 10, contendo os membros* `x` *e* `y` *do tipo inteiro. Escrever uma função para preencher o vetor com pontos aleatórios, no intervalo* $[10, 10)$*. Escrever uma função para apresentar na saída padrão o conteúdo do vetor de estruturas.*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 10
struct Ponto {
```

```
    int x;
    int y;
};

void entradaPonto(struct Ponto[], int);
void saidaPonto(struct Ponto[], int);

int main(){
    struct Ponto pt[SIZE];
    entradaPonto(pt,SIZE);
    saidaPonto(pt,SIZE);
}

void entradaPonto(struct Ponto pt[SIZE], int tam) {
    srand(time(0));
    int i;
    for(i=0; i<tam; i++) {
        pt[i].x = (rand()%20)-10;
        pt[i].y = (rand()%20)-10;
    }
}
void saidaPonto(struct Ponto pt[SIZE], int tam) {
    int i;
    for(i=0; i<tam; i++)
        printf("\nPonto %d: (%d e %d)\n",(i+1),pt[i].x,pt[i].y);
}
```

8.4 Definição de Novos Tipos

A linguagem C permite criar novos nomes para os tipos de dados. No exemplo seguinte, a declaração `typedef` define um mneumônico (*alias*, em inglês) para o tipo `float`.

```
#include <stdio.h>
int main() {
    typedef float real;
    real var = 1.5;
    printf("%f",var);
}
```

Na prática, os programadores C usam `typedef` para definir mneumônicos para tipos de dados complexos, como estruturas heterogêneas.

Exemplo 8.6.

```
#include <stdio.h>
struct ponto {
    int x;
    int y;
};
typedef struct ponto Ponto;

Ponto entradaPonto(void);
void saidaPonto(Ponto);

int main() {
    Ponto pt;
    pt = entradaPonto();
    saidaPonto(pt);
}

Ponto entradaPonto(void) {
    Ponto pt;
    printf("Coordenadas x e y do ponto: ");
    scanf("%d%d",&pt.x,&pt.y);
    return pt;
}
void saidaPonto(Ponto pt) {
    printf("\nCoordenadas: %d e %d",pt.x,pt.y);
}
```

No exemplo acima, as declarações `struct` e `typedef` podem também ser reunidas em um único comando:

```
typedef struct ponto {
    int x;
    int y;
} Ponto;
```

8.5 Aninhamento de Estruturas

Os membros de uma estrutura podem ser outras estruturas previamente definidas. Para exemplificar, considere uma estrutura `Circulo` contendo, como um dos membros, uma estrutura `Ponto`.

Exemplo 8.7.

```
#include <stdio.h>
typedef struct ponto {
    int x, y;
} Ponto;

typedef struct circulo {
    Ponto centro;
    float raio;
} Circulo;

Circulo entradaCirculo(void);
void saidaCirculo(Circulo);

int main(){
    Circulo circ = entradaCirculo();
    saidaCirculo(circ);
}

Circulo entradaCirculo(void) {
    Circulo circ;
    printf("Coordenadas do centro do circulo: ");
    scanf("%d%d",&circ.centro.x,&circ.centro.y);
    printf("Raio do circulo: ");
    scanf("%f",&circ.raio);
    return circ;
}

void saidaCirculo(Circulo circ) {
    printf("\nCoordenadas do centro do circulo: ");
    printf("%d e %d\n",circ.centro.x,circ.centro.y);
    printf("\nRaio do circulo: %.2f",circ.raio);
}
```

Exemplo 8.8. *Escreva um programa C para criar uma estrutura Ponto e uma estrutura* `Circulo`*, contendo os membros* `centro`*, do tipo ponto, e* `raio`*, do tipo float. Crie um círculo: escreva uma função para solicitar ao usuário os valores do centro e do raio, membros da estrutura* `Circulo`*. Crie um ponto: escreva uma função para solicitar ao usuário os valores de x e y, membros da estrutura* `Ponto`*. Escreva uma função para verificar se o ponto está contido no círculo.*

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

```
typedef struct ponto {
    int x, y;
} Ponto;

typedef struct circulo {
    Ponto centro;
    float raio;
} Circulo;

Ponto entradaPonto(void);
Circulo entradaCirculo(void);
int verificaPonto(Ponto, Circulo);

int main(){
    Circulo c = entradaCirculo();
    Ponto p = entradaPonto();
    if(verificaPonto(p,c))
        printf("O ponto esta dentro do circulo");
    else
        printf("O ponto esta fora do circulo");
}

Ponto entradaPonto(void) {
    Ponto p;
    printf("Coordenadas do ponto: ");
    scanf("%d%d",&p.x,&p.y);
    return p;
}
Circulo entradaCirculo(void) {
    Circulo c;
    printf("Coordenadas do centro do circulo: ");
    scanf("%d%d",&c.centro.x,&c.centro.y);
    printf("Raio do circulo: ");
    scanf("%f",&c.raio);
    return c;
}

int verificaPonto(Ponto p, Circulo c){
    return (sqrt(pow(p.x-c.centro.x,2)+pow(p.y-c.centro.y,2))) < c.raio;
}
```

Exercício 8.1. *Escreva um programa C para criar uma estrutura* `Aluno` *contendo os membros matricula (inteiro) e nota (real). Crie um vetor de alunos de tamanho 10. Preencha o vetor com números de matrícula sequenciais de 1 a 10 e notas geradas randomicamente*

no intervalo [0.0 10.0]. *Apresente na tela o vetor classificado em ordem decrescente de nota.*