

# Linguagem



Silvio do Lago Pereira



# SUMÁRIO

---

<b>1. INTRODUÇÃO .....</b>	<b>01</b>
1.1. A Origem da Linguagem <i>C</i> .....	01
1.2. O Ambiente <i>Turbo C</i> .....	01
1.3. A Estrutura Básica dos Programas.....	02
1.4. Tipos de Dados .....	04
1.4.1. Tipos de Dados Modificados .....	05
1.5. Entrada e Saída de Dados Formatada .....	06
1.5.1. Formatação de Campos de Exibição.....	07
1.6. Operadores Aritméticos.....	08
<b>2. COMANDOS DE DECISÃO .....</b>	<b>09</b>
2.1. Expressões Lógicas.....	09
2.2. Decisão Simples.....	10
2.2.1. Operador condicional.....	13
2.2.2. Condicionais Aninhados e Encadeados .....	14
2.3. Decisão Múltipla .....	15
<b>3. COMANDOS DE REPETIÇÃO .....</b>	<b>18</b>
3.1. Expressões Compactas .....	18
3.1.1. Operadores Aritméticos de Atribuição .....	18
3.1.2. Incremento e Decremento .....	19
3.2. Repetição com Contador.....	20
3.3. Repetição com Precondição .....	23
3.3.1. Deixando rastros na Tela .....	25
3.4. Repetição com Poscondição .....	26
3.5. Interrompendo uma Repetição.....	29
<b>4. MACROS E FUNÇÕES .....</b>	<b>31</b>
4.1. Preprocessamento .....	31
4.1.1. A diretiva <i>#define</i> .....	31
4.1.2. A diretiva <i>#include</i> .....	33

---

4.2. Definição e Uso de Funções .....	34
4.2.1. Funções que não Devolvem Resposta.....	35
4.2.2. Funções que Devolvem Resposta.....	38
4.3. Classes de Armazenamento .....	40
4.4. Recursividade .....	42
4.4.1. Funções Recursivas .....	44
4.4.2. Procedimentos Recursivos.....	46
<b>5. VETORES, STRINGS E MATRIZES .....</b>	<b>48</b>
5.1. Vetores .....	48
5.1.1. Inicialização de Vetores.....	50
5.1.2. Parâmetros do Tipo Vetor .....	52
5.2. Strings.....	55
5.2.1. Inicialização de Strings .....	56
5.2.2. Manipulação de Strings .....	57
5.3. Matrizes .....	60
5.3.1. Inicialização de Matrizes.....	62
5.3.2. Passando Matrizes a Funções .....	63
5.4. Métodos de Busca .....	65
5.4.1. Busca Linear .....	65
5.4.2. Busca Binária .....	67
5.5. Métodos de Ordenação .....	71
5.5.1. Ordenação por Trocas.....	71
5.5.2. Ordenação por Seleção .....	73
5.5.3. Ordenação por Inserção.....	75
<b>6. ESTRUTURAS E UNIÕES.....</b>	<b>77</b>
6.1. Estruturas .....	77
6.1.1. Inicialização e Aninhamento.....	79
6.1.2. Vetores de Estruturas .....	81
6.1.3. Ordenação e Busca em Tabelas.....	82
6.2. Uniões.....	84
6.2.1. Uniões Etiquetadas.....	84
6.3. Campos de Bits.....	86
6.3.1. Acessando um Dispositivo de Hardware.....	87

6.3.2. Economizando Espaço de Armazenamento .....	89
<b>7. PONTEIROS</b> .....	92
7.1. Definição e Uso.....	92
7.1.1. Passagem por Referência .....	94
7.1.2. Ponteiros para Ponteiros.....	96
7.1.3. Aritmética de ponteiros.....	97
7.2. Ponteiros e Vetores.....	99
7.2.1. Vetores de Strings .....	100
7.2.2. Argumentos da Linha de Comando.....	101
7.3. Ponteiros e Funções .....	103
7.3.1. Funções que Devolvem Ponteiros.....	103
7.3.2. Ponteiros para Funções.....	104
7.4. Ponteiros e Estruturas .....	107
7.4.1. Alocação Dinâmica de Memória .....	109
7.4.2. Listas Encadeadas.....	112
7.4.3. Tratamento Recursivo de Listas .....	115
<b>8. ARQUIVOS</b> .....	119
8.1. Ponteiros de Arquivo.....	119
8.2. Arquivos-Padrão.....	120
8.2.1. Redirecionamento de E/S padrão .....	121
8.3. Operações Básicas .....	122
8.3.1. Abertura de Arquivo.....	122
8.3.2. Fechamento de Arquivo.....	123
8.3.3. Verificação de Final de Arquivo .....	124
8.4. Modo Texto versus Modo Binário .....	125
8.4.1. E/S character .....	125
8.4.2. E/S Formatada .....	127
8.4.3. E/S Binária.....	128
<b>TABELA ASCII</b> .....	130
<b>BIBLIOGRAFIA</b> .....	131



# 1. INTRODUÇÃO

*C é geralmente citada como uma linguagem que reúne características tais como expressividade, portabilidade e eficiência. Embora seja uma linguagem de uso geral, C é especialmente indicada para o desenvolvimento de software básico. Nesse capítulo apresentamos a estrutura básica dos programas em C, funções de E/S, tipos de dados e operadores.*

## 1.1. A ORIGEM DA LINGUAGEM C

---

A linguagem C foi desenvolvida em 1972, nos Laboratórios *Bell*, por *Dennis Ritchie* e implementada pela primeira vez num computador DEC PDP-11 que usava o sistema operacional UNIX. Ela é o resultado da evolução de uma linguagem de programação mais antiga, denominada BCPL, desenvolvida por *Martin Richards*. Tendo sido desenvolvida por programadores, e para programadores, C tornou-se rapidamente uma ferramenta de programação bastante difundida entre os profissionais da área.

A popularidade da linguagem C deve-se, principalmente, ao fato dela ser uma linguagem flexível, portátil e eficiente. Sua flexibilidade lhe permite ser utilizada no desenvolvimento de diversos tipos de aplicação, desde simples jogos eletrônicos até poderosos controladores de satélites. Graças à sua portabilidade, os programas codificados em C podem ser executados em diversas plataformas, praticamente, sem nenhuma alteração. E, finalmente, sua eficiência proporciona alta velocidade de execução e economia de memória.

## 1.2. O AMBIENTE TURBO C

---

O *Borland Turbo C*® é o ambiente no qual desenvolveremos nossos programas. Ele é composto por um editor de textos, um compilador e um *linkeditor* que, juntos, nos permitem criar programas executáveis a partir de textos escritos em C. Os recursos oferecidos nesse ambiente poderão ser explorados à medida que estivermos mais familiarizados com a linguagem C. Para começar, é suficiente saber que:

- *F2* salva o código-fonte do programa num arquivo com extensão .c,
- *Ctrl+F9* compila, *linkedita* (gera arquivo .exe) e executa seu programa,
- *Alt+X* finaliza a execução do *Turbo C*.

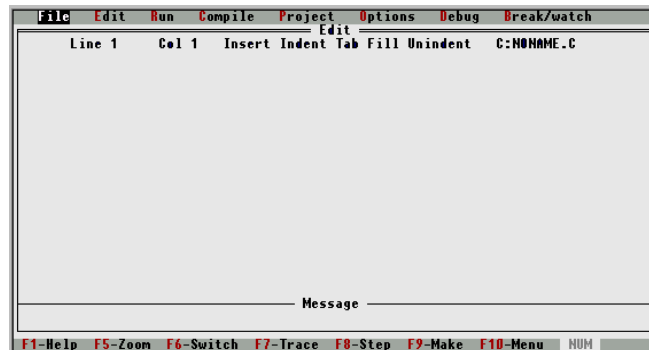


Figura 1.1 – A tela do ambiente integrado Turbo C

### 1.3. A ESTRUTURA BÁSICA DOS PROGRAMAS

Um programa C consiste de funções sendo que, necessariamente, uma delas deve ser denominada *main*. Essa é a função principal, por onde inicia-se a execução do programa, e sem ela o programa não pode ser executado.

**Exemplo 1.1.** Uma pessoa é obesa se seu índice de massa corpórea é superior a 30, tal índice é a razão entre seu peso e o quadrado da sua altura.

*/\* OBESO.C – informa se uma pessoa está ou não obesa \*/*

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define LIMITE 30
main() {
    float peso, altura, imc;
    clrscr();
    printf("\n Qual o seu peso e altura? ");
    scanf("%f %f", &peso, &altura);
    imc = peso/pow(altura,2);
    printf("\n Seu i.m.c. é %.1f", imc);
    if( imc <= LIMITE ) printf("\n Você não está obeso!");
    else                printf("\n Você está obeso!");
    getch();
}
```

Como esse programa é muito simples, ele consiste de uma única função: *main*. Essa função solicita os dados da pessoa, calcula o seu índice de massa corpórea e informa se ela está obesa ou não. ☒



Alguns pontos desse primeiro exemplo devem ser ressaltados:

- Todo texto delimitado por `/*` e `*/` é considerado como comentário, isto é, serve apenas para esclarecer algum ponto específico do programa.
- A diretiva `#include` causa a inclusão de *arquivos de cabeçalho* contendo declarações necessárias à compilação. Os arquivos `stdio.h`, `conio.h` e `math.h` declaram, respectivamente, comandos de E/S padrão, E/S console e funções matemáticas. A diretiva `#define` declara constantes simbólicas.
- Os parênteses após o nome de uma função, como em `main()`, são obrigatórios. Além disso, o compilador distingue maiúsculas e minúsculas e, portanto, o nome `main` é reservado, mas `Main` não o é.
- As chaves `{` e `}` servem para delimitar um bloco de instruções. As variáveis devem ser declaradas antes de serem usadas, logo no início do bloco.
- A função `clrscr()` serve para limpar a tela e as funções `scanf()` e `printf()` realizam entrada e saída de dados padrão.
- Cálculos e comparações são efetuados com os operadores aritméticos, funções matemáticas e operadores relacionais convencionais. A atribuição de valores às variáveis é realizada pelo operador `=`.
- A função `getch()` aguarda que uma tecla seja pressionada para que a execução do programa seja concluída. Isso permite que o usuário veja a saída do programa, antes de voltar à tela do ambiente integrado.

**Exercício 1.1.** Execute o programa `obeso.c` usando o *Borland Turbo C*.

**Exercício 1.2.** Descubra os erros no programa a seguir:

```
/* PERIM.C - informa o perímetro de uma circunferência */
#include <studio.h>
#define PI = 3.1415
Main() {
    float raio;
    clrscr;
    printf("\n Qual a medida do raio? ");          /* solicita o raio
    scanf("%f", &raio);                             da circunferência */
    float perim;                                     /* calcula o seu
    perim := 2*PI*raio;                               perímetro */
    printf("\n O perímetro é %f", perim);
    getch;
}
```

## 1.4. TIPOS DE DADOS

Programas servem essencialmente para manipular dados: ao executarmos um programa, fornecemos-lhes dados e esperamos que ele faça alguma coisa com eles. Os dados podem se apresentar em duas formas distintas: como *constantes* ou *variáveis*. No programa do exemplo 1.1, o número 30 é um dado constante, enquanto o peso e a altura da pessoa são dados variáveis, isto é, são dados cujos valores variam de uma execução para outra. Além da distinção entre as formas em que os dados podem se apresentar, existe também uma distinção entre os *tipos de dados* que o computador é capaz de manipular; como, por exemplo, números, letras, palavras, etc.

A linguagem *C* oferece cinco tipos de dados básicos:

Tipo	Espaço	Escala
<i>char</i>	1 byte	−128 a +127
<i>int</i>	2 bytes	−32768 a +32767
<i>float</i>	4 bytes	3.4e−38 a 3.4e+38
<i>double</i>	8 bytes	1.7e−308 a 1.7e+308
<i>void</i>	<i>nenhum</i>	<i>nenhuma</i>

Em *C* não existe muita distinção entre a representação gráfica de um caracter e o seu código ASCII. Como sabemos, o computador somente é capaz de manipular números. Então, quando atribuímos um caracter a uma variável, estamos na verdade armazenando o seu código ASCII (que na tabela padrão varia de 0 a 127). É por este motivo que a escala de valores do tipo *char* varia no intervalo de −128 a +127. Para representar um caracter constante, basta escrevê-lo entre apóstrofes como, por exemplo, 'A'.

As variáveis do tipo *int* podem armazenar números maiores que as variáveis do tipo *char*, mas também gastam mais espaço de memória. Valores fracionários podem ser armazenados em variáveis do tipo *float* ou *double*, conforme a necessidade. Já o tipo *void* é um tipo especial, que tem aplicação mais avançada, e seu uso será visto mais adiante.

A declaração de uma variável consiste em um *tipo* e um *identificador*. O tipo determina o espaço de memória que deverá ser alocado para ela e o identificador permitirá que ela seja referenciada no restante do programa.

☛ *Todo identificador deve iniciar-se com letra (maiúscula ou minúscula) e ser composto exclusivamente por letras, dígitos e sublinhas.*

**Exemplo 1.2.** Declaração de variáveis.

```
char tecla, opcao;
int x, y, z;
float comissao, desconto, salario;
```



#### 1.4.1. TIPOS DE DADOS MODIFICADOS

Além dos tipos básicos, *C* oferece também alguns tipos de dados modificados:

Tipo	Espaço	Escala
<i>unsigned char</i>	1 byte	0 a 255
<i>unsigned int</i>	2 bytes	0 a 65535
<i>long int</i>	4 bytes	-2 147 483 648 a +2 147 483 647

O *bit* mais à esquerda em uma variável do tipo *char* ou *int*, denominado *bit de sinal*, é normalmente utilizado pelo computador para distinguir entre valores positivos e negativos: se esse *bit* é zero, então o valor é positivo; caso contrário, ele é negativo. Usando o modificador *unsigned*, estamos informando ao compilador que somente valores sem sinal serão usados e que, portanto, não é necessário ter um *bit* de sinal. Com isso, podemos usar esse *bit* para representar valores e, conseqüentemente, a escala de valores dobra. Há também o modificador *signed*, que indica que os valores devem ser sinalizados; mas, como este é o caso normal, raramente ele é utilizado. Se a palavra *unsigned* (ou *signed*) é usada isoladamente, o compilador assume o tipo *int*.

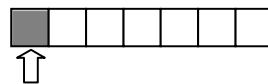


Figura 1.2 – O bit de sinal numa variável do tipo *char*

O modificador *long* faz com que o espaço de memória reservado para uma variável do tipo *int* seja duplicado e, conseqüentemente, aumenta a capacidade de armazenamento da variável. Já o modificador *short*, em algumas máquinas, faz com que esse espaço caia para a metade.

**Exemplo 1.3.** Algumas variáveis de tipos modificados.

```
unsigned char contador;
unsigned int a, b, c;
long int tam_arquivo;
```



Os modificadores podem prefixar apenas os tipos *char* e *int*. A única exceção feita é *long float*, que equivale ao tipo *double* e por isso é raramente utilizado

## 1.5. ENTRADA E SAÍDA DE DADOS FORMATADA

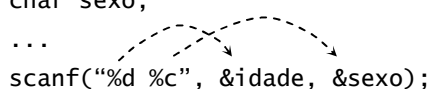
A função `scanf()` permite que um valor seja lido do teclado e armazenado numa variável. Sua sintaxe consiste numa *cadeia de formatação* seguida de uma lista de argumentos, cada um deles sendo o endereço de uma variável:

`scanf("formatação", arg1, arg2, ..., argn);`

A cadeia de formatação é composta por códigos especiais, denominados *especificadores de formato*, que indicam a quantidade e os tipos dos dados que serão lidos pela função.

**Exemplo 1.4.** Lendo dados com a função `scanf()`.

```
int idade;
char sexo;
...
scanf("%d %c", &idade, &sexo);
```



Como podemos observar, a cada especificação de formato na cadeia de formatação corresponde um endereço de variável na lista de argumentos. ☑

☛ O operador `&` informa o endereço de memória em que uma variável foi alocada.

Especificador	Representa
%c	um único caracter
%o, %d, %x	um número inteiro em octal, decimal ou hexadecimal
%u	um número inteiro em base decimal sem sinal
%ld	um número inteiro longo em base decimal
%f, %lf	um número real de precisão simples ou dupla
%s	uma cadeia de caracteres ( <i>string</i> )
%%	um único sinal de porcentagem

A função `printf()` nos permite exibir informações formatadas no vídeo. A sua sintaxe é essencialmente idêntica àquela da função `scanf()`. A principal diferença é que agora a lista de argumentos deve conter os valores a serem exibidos e não seus endereços:

`printf("formatação", arg1, arg2, ..., argn);`

Além disso, a cadeia de formatação pode conter também texto, que é exibido normalmente, e caracteres de controle, cuja exibição causa efeitos especiais.

Os principais caracteres de controle utilizados com a função *printf()* são:

Caracter de controle	Efeito
\a	soa o alarme do microcomputador
\b	o cursor retrocede uma coluna
\f	alimenta página na impressora
\n	o cursor avança para uma nova linha
\r	o cursor retrocede para a primeira coluna da linha
\t	o cursor avança para próxima marca de tabulação
\"	exibe uma única aspa
\'	exibe um único apóstrofo
\\	exibe uma única barra invertida

Desses caracteres, o mais usado é '\n'. Através dele podemos indicar quando uma nova linha deve ser utilizada ao se exibir alguma informação na tela.

**Exemplo 1.5.** Exibindo dados com a função *printf()*.

```
#include <stdio.h>
#define PI 3.1415
main() {
    double raio, perim;
    getch();
    printf("\n Qual a medida do raio?");
    scanf("%lf", &raio);
    perim = 2*PI*raio;
    printf("\n O perímetro da circunferência é %lf", perim);
    getch();
}
```



#### 1.5.1. FORMATAÇÃO DE CAMPOS PARA EXIBIÇÃO

A função *printf()* permite que os campos de exibição sejam formatados. As formatações mais usadas são o preenchimento com zeros à esquerda, para inteiros, e a especificação do número de casas decimais, para reais.

**Exemplo 1.6.** Formatação de campos com *printf()*.

```
int a = 678;
float b = 12.3456;
...
printf("\n|%5d|",a);
printf("\n|%06d|",a);
printf("\n|%7.3d|",b);
printf("\n|%7.2d|",b);
...
```

A saída produzida pela execução desse trecho de programa é:

```
| 678|  
|000678|  
| 12.346|  
| 12.34|
```

Note que valores reais são arredondados apenas quando a primeira casa decimal desprezada é igual ou superior a 5. ☒

## 1.6. OPERADORES ARITMÉTICOS

---

C oferece operadores para as quatro operações aritméticas e também um operador para calcular o resto da divisão entre dois números inteiros.

Operador	Resultado
+	soma de dois números quaisquer
−	diferença entre dois números quaisquer
*	produto de dois números quaisquer
/	quociente da divisão de dois números
%	resto da divisão de dois número inteiros

Destes operadores, apenas dois merecem atenção especial. Os demais funcionam conforme as regras usuais estabelecidas na matemática básica.

- *Divisão*: o operador de divisão fornece resultado inteiro apenas quando ambos os operandos são inteiros. Por exemplo,  $7 / 2 \Rightarrow 3$  e  $7.0 / 2 \Rightarrow 3.5$ .
- *Resto*: o operador de resto somente pode ser utilizado com operandos inteiros. Por exemplo,  $7 \% 2 \Rightarrow 1$  e  $7.0 \% 2 \Rightarrow \text{erro}$ .

**Exercício 1.3.** Dadas as duas notas de um aluno, informe a sua média final.

**Exercício 1.4.** Dados uma distância e o total de litros de combustível gasto por um automóvel para percorrê-la, informe o consumo médio.

**Exercício 1.5.** Dado um caracter, informe o seu código ASCII em octal, decimal e hexadecimal

**Exercício 1.6.** Dada um temperatura em graus *Fahrenheit*, informe o valor correspondente em graus *Celsius*. [Dica:  $C = (F - 32) * (5 / 9)$ ].

**Exercício 1.7.** Dadas as medidas dos catetos de um triângulo retângulo, informe a medida da hipotenusa. [Dica: para calcular a raiz quadrada use a função `sqrt()`, definida no arquivo `math.h`].

## 2. COMANDOS DE DECISÃO

*Nesse capítulo introduzimos os conceitos básicos relacionados à tomada de decisão, tais como valores lógicos, operadores relacionais e conectivos, e apresentamos os comandos condicionais oferecidos na linguagem C.*

### 2.1. EXPRESSÕES LÓGICAS

Em C, não existe um tipo específico para a representação de valores lógicos. Entretanto, qualquer valor pode ser interpretado como um valor lógico: “zero representa falso e qualquer outro valor representa verdade”. Por exemplo, os valores 5, -3, 1.2 e 'a' são *verdadeiros*, enquanto 0 e 4-4 são *falsos*.

☛ Em C, 0 representa o valor lógico “falso” e 1 representa o valor lógico “verdade”.

Para gerar um valor lógico, usamos os operadores *relacionais*. Através deles podemos comparar dois valores de diversas formas. O resultado da avaliação de um operador relacional é 0 se a comparação é *falsa* e 1 se *verdadeira*.

Operador relacional	Resultado
$x = y$	verdade se $x$ for igual a $y$
$x != y$	verdade se $x$ for diferente de $y$
$x < y$	verdade se $x$ for menor que $y$
$x > y$	verdade se $x$ for maior que $y$
$x <= y$	verdade se $x$ for menor ou igual a $y$
$x >= y$	verdade se $x$ for maior ou igual a $y$

**Exemplo 2.1.** Operadores relacionais e valores lógicos.

```
...
printf("%d %d", 5<6, 6>5);
...
```

A saída produzida pela instrução será 1 0.



Além dos operadores relacionais, C oferece também operadores lógicos. Com eles, podemos criar expressões lógicas compostas. Os operadores lógicos funcionam conforme as regras definidas na lógica matemática.

Operador lógico	Resultado
$! x$	<i>verdade</i> se e só se $x$ for <i>falso</i>
$x \ \&\& \ y$	<i>verdade</i> se e só se $x$ e $y$ forem <i>verdade</i>
$x \    \ y$	<i>verdade</i> se e só se $x$ ou $y$ for <i>verdade</i>

Numa expressão contendo operadores aritméticos, relacionais e lógicos, a avaliação é efetuada na seguinte ordem:

- ① primeiro avaliam-se todos os operadores aritméticos;
- ② em seguida, avaliam-se os operadores relacionais;
- ③ só então, avaliam-se os operadores lógicos.

**Exercício 2.1.** Qual a saída produzida pela instrução a seguir?

```
printf("%d %d %d %d", !3, !0, 3+'a'>'b'+2 && !'b', 1 || !2 && 3);
```

## 2.2. DECISÃO SIMPLES

A estrutura condicional ou de decisão simples serve para escolher um entre dois comandos alternativos, conforme ilustrado na figura 1.3. Em C, a estrutura condicional é codificada da seguinte forma:

*if( condição ) comando<sub>1</sub>; else comando<sub>2</sub>;*

e funciona, conforme ilustrado na figura 1.3, da seguinte maneira:

- ① avalia a *condição*, que deve ser uma expressão lógica;
- ② se a *condição* for verdadeira, executa apenas o *comando<sub>1</sub>*;
- ③ senão, executa apenas o *comando<sub>2</sub>*.

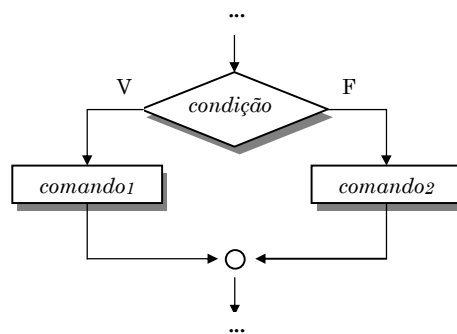


Figura 1.3 – A estrutura de decisão simples



**Exemplo 2.2.** O uso de decisão simples.

```
#include <stdio.h>

main() {
    float a, b, m;

    printf("\n Informe as duas notas obtidas: ");
    scanf("%f %f", &a, &b);

    m = (a+b)/2;

    if( m >= 7.0 ) printf("\n Aprovado");
    else printf("\n Reprovado");
}
```

O programa solicita as duas notas obtidas pelo aluno, calcula sua média e, em função desse valor, decide se o aluno está ou não aprovado. ☒

Pode ser que um dos comandos alternativos, ou ambos, seja composto por mais de uma instrução. Por exemplo, se o programa anterior tivesse que exibir *Aprovado* em azul e *Reprovado* em vermelho, então cada alternativa seria composta por duas instruções: uma para selecionar a cor e a outra para exibir a situação do aluno. Nesse caso, teríamos que usar *blocos*, agrupando as instruções em cada alternativa dentro de um par de chaves.

**Exemplo 2.3.** O uso de blocos de instruções.

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

main() {
    float a, b, m;

    clrscr();
    printf("\n Informe as duas notas obtidas: ");
    scanf("%f %f", &a, &b);

    m = (a+b)/2;

    if( m >= 7.0 ) {
        textcolor(BLUE);
        cprintf("\n Aprovado");
    }
    else {
        textcolor(RED);
        cprintf("\n Reprovado");
    }

    getch();
}
```

A função *textcolor()* seleciona a cor do texto e a função *cprintf()*, cuja sintaxe é idêntica à da *printf()*, exibe o texto na cor selecionada. Essas funções, assim como as constantes BLUE e RED, estão declaradas no arquivo *conio.h*. ☒

**Exercício 2.2.** Dados dois números distintos, informe qual dele é o maior.

**Exercício 2.3.** Dado um ano, informe se ele é ou não bissexto. [Dica: um ano é bissexto se é divisível por 4 mas não por 100].

Também pode acontecer de não haver duas alternativas numa decisão simples, ou seja, ou o comando é executado ou, então, nada é feito. O programa a seguir exemplifica esse caso, resolvendo o seguinte problema: “*Numa empresa paga-se R\$ 19,50 a hora e recolhe-se para o imposto de renda 10% dos salários acima de R\$ 1500,00. Dado o número de horas trabalhadas por um funcionário, informar o valor do seu salário líquido*”.

**Exemplo 2.4.** O uso de decisão simples com uma única alternativa.

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

main() {
    int h;
    float s;

    clrscr();
    printf("\nHoras trabalhadas? ");
    scanf("%d", &h);

    s = h*19.50;
    if( s>1500.00 )
        s = 0.90*s;

    printf("\nSalário líquido: R$ %.2f", s);
    getch();
}
```

Obviamente, para calcular o salário bruto, basta multiplicar o número de horas trabalhadas pelo valor pago por hora. O salário líquido será igual ao salário bruto, a menos que o seu valor exceda o limite de R\$ 1500,00. Nesse caso, o salário líquido será apenas 90% do salário bruto. ☒

 A parte "else" num comando "if" é opcional e pode ser omitida sem problemas.

**Exercício 2.3.** Dado um número real qualquer, informe seu valor absoluto.

**Exercício 2.4.** Uma empresa determinou um reajuste salarial de 5% a todos os seus funcionários. Além disto, concedeu um abono de R\$ 100,00 para aqueles que recebem até R\$ 750,00. Dado o valor do salário de um funcionário, informar para quanto ele será reajustado.

### 2.2.1. OPERADOR CONDICIONAL

C oferece também um operador que proporciona uma forma mais compacta de se representar decisões simples. O *operador condicional*, cuja sintaxe é *condição ? expressão<sub>1</sub> : expressão<sub>2</sub>*,

funciona da seguinte maneira:

- ① avalia a *condição*;
- ② se ela for verdadeira, o resultado final é o valor da *expressão<sub>1</sub>*;
- ③ senão, o resultado final é o valor da *expressão<sub>2</sub>*.

**Exemplo 2.5.** O uso do operador condicional.

```
...  
abs = n>0 ? n : -n;  
...
```

A instrução acima atribui à variável *abs* o valor absoluto da variável *n*. A expressão *n>0* é avaliada: se for verdadeira, *abs* recebe o próprio valor de *n*; caso contrário, *abs* recebe o valor de *n* com o sinal invertido. ☒

☛ Uma vantagem no uso do operador condicional é que, sendo um operador, podemos utilizá-lo em qualquer contexto em que uma expressão é permitida.

**Exemplo 2.6.** O uso do operador condicional como argumento de função.

```
#include <stdio.h>  
  
main() {  
    int x, y;  
    printf("\nInforme dois valores: ");  
    scanf("%d %d", &x, &y);  
    printf("\n Máximo = %d", x>y ? x : y);  
}
```

A instrução acima seleciona e exibe o máximo entre dois valores *x* e *y*. Note que não seria possível usar um *if-else* como argumento na função *printf()*. ☒

**Exercício 2.5.** Seja *e* uma variável contendo o número de erros detectados num certo processo. Codifique uma instrução capaz de exibir saídas como:

1 erro detectado.  
5 erros detectados.

**Exercício 2.6.** Codifique uma instrução para exibir valores lógicos como *true* e *false*. Para o valor 0 deve aparecer *false* e para qualquer outro, *true*.

### 2.2.2. CONDICIONAIS ANINHADOS E ENCADEADOS

Como vimos, a estrutura condicional serve para selecionar e executar um entre dois comandos alternativos. É possível que, algumas vezes, um destes comandos alternativos (ou ambos) sejam também condicionais. Nesse caso, dizemos que o primeiro condicional é o *principal* e o outro está *aninhado* ou *encadeado*, conforme indicado a seguir:

```
if( condição ) /* principal */
    if ...      /* aninhado */
else
    if ...      /* encadeado */
```

Para exemplificar o uso desses condicionais, vamos considerar o seguinte problema: “*Dados três números verificar se eles podem representar as medidas dos lados de um triângulo e, se puderem, classificar o triângulo em equilátero, isósceles ou escaleno*”.

Para codificar o programa, devemos lembrar das seguintes definições:

- Para que três números representem os lados de um triângulo é necessário que cada um deles seja menor que a soma dos outros dois.
- Um triângulo é equilátero se tem os três lados iguais, isósceles se tem apenas dois lados iguais e escaleno se tem todos os lados distintos.

**Exemplo 2.7.** O uso de condicionais aninhados e encadeados.

```
#include <stdio.h>
#include <conio.h>
main() {
    float a, b, c;
    clrscr();
    printf("\nInforme três números: ");
    scanf("%f %f %f", &a, &b, &c);
    if( a<b+c && b<a+c && c<a+b ) {
        printf("\né um triângulo: ");
        if( a==b && b==c ) printf("equilátero");
        else if( a==b || a==c || b==c ) printf("isósceles");
        else printf("escaleno");
    }
    else printf("\nNão é um triângulo");
    getch();
}
```



**Exercício 2.7.** Numa faculdade, os alunos com média pelo menos 7,0 são aprovados, aqueles com média inferior a 3,0 são reprovados e os demais ficam de recuperação. Dadas as duas notas de um aluno, informe sua situação. Use as cores azul, vermelho e amarelo para as mensagens *aprovado*, *reprovado* e *recuperação*, respectivamente.

**Exercício 2.8.** Dados os coeficientes ( $a \neq 0$ ,  $b$  e  $c$ ) de uma equação do 2º grau, calcule e informe suas raízes reais, usando a fórmula de *Báskara* a seguir:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4.a.c}}{2.a}$$

### 2.3. DECISÃO MÚLTIPLA

---

A estrutura de decisão múltipla é bastante adequada quando precisamos escolher uma entre várias alternativas previamente definidas, por exemplo, num menu. A decisão múltipla tem a seguinte forma básica:

```
switch( expressão ) {  
    case constante1 : comando1; break;  
    case constante2 : comando2; break;  
    ...  
    case constanten : comandon; break;  
    default          : comando;  
}
```

e funciona da seguinte maneira:

- ① Avalia a *expressão*, que deve ser do tipo *char* ou *int*;
- ② Encontra o *case* cuja constante é igual ao valor da *expressão* e executa todos os comandos seguintes até encontrar um comando *break*;
- ③ Se não existe tal caso, executa as instruções associadas ao caso *default*.



*O caso default é opcional e, embora seja geralmente posicionado no final do bloco switch, ele pode aparecer em qualquer posição entre os case's especificados.*

Note que, embora o comando *break* seja quase sempre usado juntamente com o comando *switch*, ele não faz parte da sintaxe desse comando. Se dois casos não são separados por um comando *break*, dizemos que o controle "vaza" de um caso para o outro, ou seja, quando o primeiro caso é selecionado para execução, não apenas o comando associado a ele é executado, mas também o comando associado ao segundo.

**Exemplo 2.8.** O uso da estrutura de decisão múltipla com vazamentos.

```
#include <stdio.h>

main() {
    int n;

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

    switch( n ) {
        case 1: putchar('A'); break;
        case 3: putchar('B');
        case 4: putchar('C'); break;
        default: printf('*');
        case 5: putchar('D');
    }
    putchar('.');
}
```

<i>n</i>	Saída
1	A.
2	*D.
3	BC.
4	C.
5	D.

A tabela ao lado mostra as saídas produzidas pelo programa para alguns valores atribuídos à variável *n*. Note que, como não há um caso rotulado com a constante 2, para *n*=2, o caso *default* é selecionado. Como o caso *default* não é finalizado com um *break*, o controle "vaza" para o caso seguinte, produzindo a saída \*D.. ☒

A seguir, a estrutura de decisão múltipla é usada para implementar uma simples calculadora: o usuário digita uma expressão da forma *val<sub>1</sub> oper val<sub>2</sub>* e o programa fornece-lhe seu valor como resposta.

**Exemplo 2.9.** O uso da estrutura de decisão múltipla.

```
#include <stdio.h>

main() {
    float x, y;
    char op;

    printf("\n Expressão? ");
    scanf("%f %c %f", &x, &op, &y);

    switch( op ) {
        case '+': printf("\n valor = %.2f", x+y); break;
        case '-': printf("\n valor = %.2f", x-y); break;
        case '*': printf("\n valor = %.2f", x*y); break;
        case '/': printf("\n valor = %.2f", x/y); break;
        default : printf("\n Operador inválido: %c",op);
    }
}
```

☒

Não existe restrição alguma quanto aos tipos de comandos que podem estar associados a um determinado *case* dentro do comando *switch*. Por exemplo, suponha que antes de efetuar uma divisão seja necessário verificar se o divisor é realmente diferente de zero. Nesse caso, teríamos que incluir um comando *if* dentro do *switch*; isso pode ser feito sem nenhum problema.

**Exemplo 2.10.** Usando *if* dentro do *switch*.

```
...
case '/' : if( y == 0 ) {
            printf("\n Impossível dividir por zero!");
            exit(1);
        }
        z = x/y;
        break;
...
```

Quando executada, a função *exit()* interrompe a execução do programa. ☒

Como toda a estrutura *switch* é envolvida por chaves, não é necessário usar bloco quando há mais que uma instrução associada a um determinado *case*.

**Exercício 2.9.** Altere o programa do exemplo 2.8 de modo que o usuário possa representar divisões usando também *dois-pontos* ( : ) [Dica: crie um caso sem um comando *break* correspondente].

**Exercício 2.10.** Dados a altura e o sexo de uma pessoa, determine seu peso ideal de acordo com as fórmulas a seguir:

- para homens o peso ideal é  $72.7 * altura - 58$
- para mulheres o peso ideal é  $62.1 * altura - 44.7$

**Exercício 2.11.** O perfil de uma pessoa pode ser determinado a partir da sua data de nascimento, conforme exemplificado a seguir. Dada uma data de nascimento, informe o perfil correspondente.

Exemplo: 13/06/1970

①  $1306 + 1970 = 3276$   
 ②  $32 + 76 = 108$   
 ③  $108 \div 5 = 21$   
 105 21  
 3

consulte a tabela ao lado para saber o perfil correspondente ao número 3!

R	Perfil
0	Tímido
1	Sonhador
2	Paquerador
3	Atraente
4	Irresistível

## 3. COMANDOS DE REPETIÇÃO

*Nesse capítulo introduzimos os operadores aritméticos de atribuição, de incremento e decremento, que nos permitem escrever expressões de forma mais compactas, e apresentamos os comandos de repetição e de interrupção de laço oferecidos em C.*

### 3.1. EXPRESSÕES COMPACTAS

Quando codificamos um programa, freqüentemente temos a necessidade de escrever expressões da forma *variável = variável operador expressão*. Para facilitar, C oferece um grupo especial de operadores de atribuição que nos permitem escrever essas expressões numa forma mais compacta.

#### 3.1.1. OPERADORES ARITMÉTICOS DE ATRIBUIÇÃO

Os *operadores aritméticos de atribuição* combinam, num único operador, uma operação aritmética e uma atribuição. Por exemplo, a expressão  $x = x + 3$  pode ser escrita como  $x += 3$ . O operador  $+=$  adiciona o valor da expressão à sua direita ao valor da variável à sua esquerda e armazena o resultado nessa mesma variável. Em geral, os compiladores geram um código executável mais rápido quando essas abreviações são utilizadas. Porém, a maior vantagem desses operadores é evitar erros decorrentes de redundância. Por exemplo, considere a expressão  $v[i+w[2*j]] = v[i+w[2*j]] + 3$ . Para manter a consistência, toda alteração feita na variável<sup>1</sup> do lado esquerdo deverá também ser feita na variável do lado direito. Evidentemente, escrevendo-a como  $v[i+w[2*j]] += 3$ , garantimos que toda alteração será sempre consistente.

Expressão	Forma compacta
$x = x + y$	$x += y$
$x = x - y$	$x -= y$
$x = x * y$	$x *= y$
$x = x / y$	$x /= y$
$x = x \% y$	$x \% = y$

**Exercício 3.1.** Explique por que motivo a expressão  $x = x * 2 + y$  não pode ser escrita como  $x *= 2 + y$ .

---

<sup>1</sup>  $v[i+w[2*j]]$  denota uma variável de vetor, conforme veremos no capítulo 5.



### 3.1.2. INCREMENTO E DECREMENTO

Se uma expressão incrementa ou decrementa o valor da variável, podemos então escrevê-la numa forma ainda mais compacta. Para incrementar usamos o operador ++ e para decrementar usamos o operador —. Esses operadores são unários e podem ser usados tanto na forma prefixa quanto posfixa.

- forma prefixa: ++*variável*, —*variável*
- forma posfixa: *variável*++, *variável*—

**Exemplo 3.1.** Operadores de incremento e decremento.

```
...
int x=5, y=5;
++x;
y--;
printf("\n x=%d y=%d", x, y);
...
```

Como esperado, a saída produzida pelo código acima será x=6 y=4. ☒

A diferença entre usar um operador na forma prefixa ou posfixa aparece somente quando ele é utilizado numa expressão, juntamente com outros operadores. Neste caso, o funcionamento é o seguinte:

- na forma prefixa, a variável é alterada e, depois, seu valor é usado.
- na forma posfixa, o valor da variável é usado e, depois, ela é alterada.

**Exemplo 3.2.** Operadores de incremento e decremento.

```
...
int x=5, y=5, v, w;
v = ++x;
w = y--;
printf("\n x=%d y=%d v=%d w=%d", x, y, v, w);
...
```

Agora, a saída produzida pelo código será x=6 y=4 v=6 w=5. ☒

☛ *Os operadores de incremento e decremento não podem ser aplicados a valores constantes, já que  $x++$  equivale, de certa forma, a  $x = x + 1$ .*

**Exercício 3.2.** Seja  $x=5$  e considere a instrução  $y = x++ + ++x$ . Quais os valores das variáveis  $x$  e  $y$  após a execução dessa instrução? Por quê?

### 3.2. REPETIÇÃO COM CONTADOR

---

A estrutura de repetição com contador tem seu funcionamento controlado por uma variável que conta o número de vezes que o comando é executado. Em C, essa estrutura é implementada pelo comando *for*, cuja forma básica é a seguinte:

*for*( *inicialização*; *condição*; *alteração*) *comando*;

A *inicialização* é uma expressão que atribui um valor inicial ao contador, a *condição* verifica se a contagem chegou ao fim e a *alteração* modifica o valor do contador. Enquanto a contagem não termina, o *comando* associado ao *for* é repetidamente executado. O funcionamento dessa estrutura pode ser acompanhado na figura a seguir:

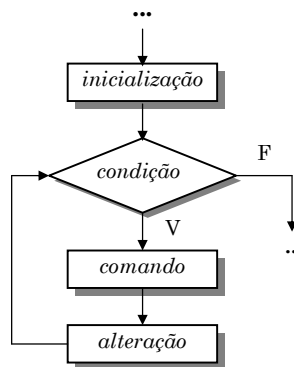


Figura 3.1 – A estrutura de repetição com contador

**Exemplo 3.3.** Uma contagem progressiva.

```
#include <stdio.h>
main() {
    int c;
    for(c=1; c<=9; c++) printf("%d ", c);
}
```

A saída produzida pelo código será 1 2 3 4 5 6 7 8 9.



**Exercício 3.3.** Dado um valor  $n$ , exiba uma contagem regressiva.

**Exercício 3.4.** Exiba uma tabela de conversão de polegadas em centímetros, variando as polegadas de 0 a 10 de meio em meio. [Dica: 1"  $\approx$  2,54 cm]

**Exercício 3.5.** Dados um número real  $x$  e um natural  $n$ , exiba a potência  $x^n$ .

**Exercício 3.6.** Dados um número natural  $n$ , exiba seu fatorial  $n!$ .

O exemplo anterior tem uma única instrução dentro do *for*. Porém, usando um bloco, podemos executar qualquer número de instruções.

**Exemplo 3.5.** Exibindo tabuadas.

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

main() {
    int n, c, r;
    clrscr();
    printf("\n Digite um número entre 1 e 10: ");
    scanf("%d", &n);
    for(c=1; c<=10; c++) {
        r = n*c;
        printf("\n %d x %2d = %3d", n, c, r);
    }
    getch();
}
```

Supondo que o usuário digite o número 7, o programa exibirá:

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```



**Exercício 3.7.** O quadrado de um número natural  $n$  é dado pela soma dos  $n$  primeiros números ímpares consecutivos. Por exemplo,  $1^2=1$ ,  $2^2=1+3$ ,  $3^2=1+3+5$ ,  $4^2=1+3+5+7$ , etc. Dado um número  $n$ , calcule seu quadrado usando a soma de ímpares ao invés de produto.

**Exercício 3.8.** A série de *Fibonacci* é 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... Os dois primeiros termos são iguais a 1 e, a partir do terceiro, o termo é dado pela soma dos dois termos anteriores. Dado um número  $n \geq 3$ , exiba o  $n$ -ésimo termo da série de *Fibonacci*.



Não há qualquer restrição quanto às instruções repetidas pelo comando *for*.

O próximo exemplo mostra o uso de um comando *if* dentro do comando *for*.

**Exemplo 3.6.** Exibe a tabela ASCII com pausas a cada 23 linhas.

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

main() {
    int c, n=0;
    for(c=0; c<=255; c++) {
        printf("\n%c ==> %d", c, c);
        n++;
        if( n==23 ) {
            printf("\n\nPressione uma tecla ...");
            n=0;
            getch();
        }
    }
}
```

O contador de linhas  $n$  é iniciado com o valor 0 e incrementado a cada linha exibida na tela. Quando seu valor torna-se igual a 23, o programa reinicia seu valor em 0 e aguarda o usuário pressionar uma tecla para prosseguir. ☒

O próximo exemplo mostra o uso de um *for* aninhado dentro de outro. O programa exibe um tabuleiro de xadrez cujo tamanho é indicado pelo usuário. Para determinar a cor dos quadros do tabuleiro, basta observar na figura a seguir que os quadros brancos correspondem a posições cuja soma de suas coordenadas é par e aqueles de cor preta, a posições cuja soma é ímpar.

	1	2	3	4
1				
2				
3				
4				

**Figura 3.2** – Um tabuleiro de xadrez 4×4

**Exemplo 3.7.** Exibe um tabuleiro de xadrez.

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

main() {
    int lin, col, n;
    clrscr();
    printf("\n Qual o tamanho do tabuleiro? ");
    scanf("%d", &n);
    for(lin=1; lin<=n; lin++) {
        printf("\n");
        for(col=1; col<=n; col++) {
```

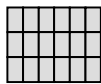
```

        if( (lin+col)%2 == 0 ) textcolor(YELLOW);
        else textcolor(GREEN);
        cprintf("%c%c",219,219);
    }
}
getch();
}

```

O código ASCII 219 corresponde ao caracter '▣'. Como sua altura é o dobro de sua largura, precisamos exibir dois deles para formar um quadrado. ☑

**Exercício 3.9.** Dados dois números naturais  $m$  e  $n$ , exiba um retângulo com  $m$  caracteres de altura e  $n$  caracteres de largura. Por exemplo, se forem dados os números 3 e 6, deverá ser exibido o seguinte desenho:



### 3.3. REPETIÇÃO COM PRECONDIÇÃO

A estrutura de repetição com precondição, ilustrada na figura 3.3, é mais genérica que aquela com contador. Em *C*, ela tem a seguinte forma:

*while( condição ) comando;*

Seu funcionamento é controlado por uma única expressão, sua *condição*, cujo valor deve ser verdadeiro para que o comando seja repetido. A repetição com precondição pára somente quando sua condição torna-se falsa.

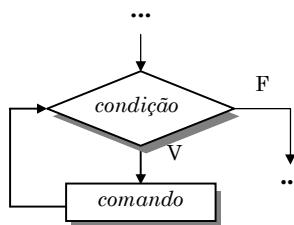


Figura 3.3 – A estrutura de repetição com precondição

Para exemplificar o uso de repetição com precondição, vamos resolver o seguinte problema: “*dado um número natural, exibir os seus dígitos*”. Por exemplo, dado o número 8503 como entrada, o programa deverá exibir como saída os dígitos 3, 0, 5 e 8. A estratégia será dividir o número sucessivamente

por 10 e ir exibindo os restos obtidos, um a um. O processo se repete enquanto o número for diferente de zero, conforme ilustrado a seguir:

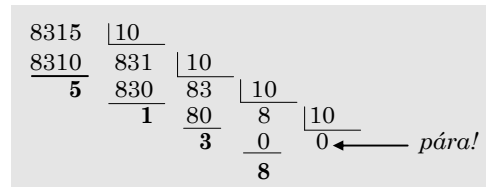


Figura 3.4 – Isolando os dígitos de um número

**Exemplo 3.8.** Exibe os dígitos de um número.

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

main() {
    unsigned n, d;

    clrscr();
    printf("\n Digite um número: ");
    scanf("%u", &n);
    printf("\n Os seus dígitos são: ");
    while( n != 0 ) {
        d = n % 10;
        n /= 10;
        printf("%u ", d);
    }
    getch();
}
```



**Exercício 3.10.** Numa certa agência bancária, as contas são identificadas por números de até seis dígitos seguidos de um dígito verificador, calculado conforme exemplificado a seguir. Dado um número de conta  $n$ , exiba o número de conta completo correspondente.

Seja  $n = 7314$  o número da conta.

1º Adicionamos os dígitos de  $n$  e obtemos a soma  $s = 4+1+3+7 = 15$ ;

2º Calculamos o resto da divisão de  $s$  por 10 e obtemos o dígito  $d = 5$ .

Número de conta completo: 007314-5

**Exercício 3.11.** Um número natural é *triangular* se é igual à soma dos  $n$  primeiros números naturais consecutivos, a partir de 1. Por exemplo, 1, 3, 6, 10, 15, ... são triangulares. Dado um natural  $n \geq 1$ , informe se ele é triangular.

### 3.3.1. DEIXANDO RASTROS NA TELA

O próximo exemplo ilustra uma outra forma em que o comando *while* é bastante usado. Nessa forma, sua condição é a constante 1 e, como essa condição é sempre verdadeira, temos *a priori* uma repetição infinita. Porém, entre as instruções que compõem o corpo do comando *while*, existe sempre uma que garante o término da repetição.

O programa deixa um rastro na tela enquanto pressionamos as teclas Norte, Sul, Leste ou Oeste. No início, o cursor está posicionado no meio da tela e, à medida que pressionarmos as teclas, o cursor vai se deslocando na direção correspondente. Para finalizar a execução, pressionamos a tecla Fim.

A teclas são lidas pela função *getch()*, o cursor é posicionado pela função *gotoxy()* e o caracter que forma o rastro é exibido pela função *putch()*. Essas três funções são definidas no arquivo *conio.h*. A função *toupper()*, definida no arquivo *ctype.h*, é usada para converter as letras para maiúsculas e a função *exit()*, definida em *stdlib.h*, é usada para finalizar a execução do programa.

**Exemplo 3.8.** Deixando um rastro na tela.

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

main() {
    int col=40, lin=12;
    clrscr();
    while( 1 ) {
        gotoxy(col,lin);
        putch(219);
        switch( toupper(getch()) ) {
            case 'N': if( lin>1 ) lin--; break;
            case 'S': if( lin<24 ) lin++; break;
            case 'L': if( col<80 ) col++; break;
            case 'O': if( col>1 ) col--; break;
            case 'F': exit(1);
        }
    }
}
```



Alguns pontos desse programa merecem esclarecimentos adicionais:

- A tela no modo texto é formada por 25 linhas de 80 colunas. Então, iniciando as variáveis *col* e *lin* com os valores 40 e 12, respectivamente, garantimos que o cursor aparecerá inicialmente no centro da tela.

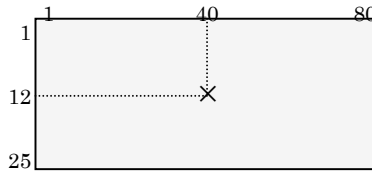


Figura 3.5 – As dimensões da tela no modo texto

- A função *gotoxy()* recebe dois parâmetros: o primeiro representa a coluna e o outro representa a linha em que o cursor deverá ser posicionado.
- Os condicionais *if*'s, dentro do *switch*, garantem que as coordenadas do cursor estarão sempre dentro dos limites da tela.
- Devido à forma como as linhas e colunas são numeradas, indo para o norte o número da linha diminui e indo para o sul, aumenta; analogamente, para oeste o número da coluna diminui e para leste, aumenta.

**Exercício 3.12.** As teclas especiais, quando pressionadas, geram dois *bytes*: o primeiro é sempre 0 e o segundo é um número que a identifica. Por exemplo, pressionando F1 ao executar as instruções *x=getch()* e *y=getch()*, obtemos *x* igual a 0 e *y* igual a 59. Descubra os números que identificam as *setas* no teclado e altere o programa do exemplo 3.8 de modo que o cursor possa ser movimentado através do pressionamento delas.

**Exercício 3.13.** Altere o programa de rastro de modo que ele implemente também o seguinte menu: F2 – *(des)ativa rastro*, F3 – *(des)ativa borracha*, F4 – *seleciona cor* e F5 – *fim*. [Dica: use a cor preta para o modo borracha] .

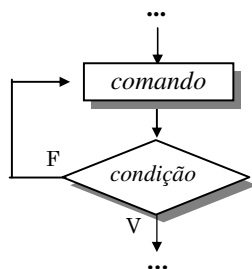
### 3.4. REPETIÇÃO COM POSCONDIÇÃO

A estrutura de repetição com poscondição é bastante semelhante àquela com precondição. A diferença é que nessa última a condição é verificada antes que o comando seja executado, enquanto na primeira a condição é verificada somente depois que o comando é executado. Conseqüentemente, a repetição com poscondição garante que o comando seja executado pelo menos uma vez.

*do { comando; } while( condição );*

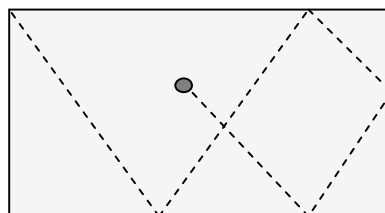
☛ Não é preciso usar chaves quando há um único comando a ser repetido, entretanto, elas são geralmente incluídas para evitar confusão entre *while* e *do-while*.





**Figura 3.6** – Estrutura de repetição com poscondição

Vamos criar um programa que simula o movimento de uma bola de ping-pong batendo nas bordas da tela, conforme ilustrado a seguir:



**Figura 3.7** – Estrutura de repetição com poscondição

O movimento da bola se repete até que uma tecla seja pressionada, fato que será verificado através da função *kbhit()*. Esta função, definida em *conio.h*, devolve 1 se alguma tecla foi pressionada e 0 em caso contrário. Também, para simular o som da bola batendo na borda da tela, usaremos as seguintes funções definidas em *conio.h*:

- *sound(n)*: emite som com frequência de *n* hertz;
- *nosound()*: cessa o som emitido pelo alto-falante;
- *delay(m)*: gera uma pausa de *m* milissegundos.

**Exemplo 3.9.** Simulando o movimento de uma bola de ping-pong.

```

#include <stdio.h>
#include <conio.h>

main() {
    int x=1, y=1;
    int dx=-1, dy=-1;
    clrscr();
    do {

```

```

gotoxy(x,y);
putch(01);
delay(800);
gotoxy(x,y);
putch(32);
if( x==1 || x==80 ) {
    sound(500);
    delay(800);
    nosound();
    dx = -dx;
}
if( y==1 || y==24 ) {
    sound(500);
    delay(800);
    nosound();
    dy = -dy;
}
x += dx;
y += dy;
} while( !kbhit() );
}

```



**Exercício 3.14.** Dada uma série de números positivos (finalizada com um valor nulo) que representam as idades das pessoas que moram num certo bairro, determine a idade da pessoa mais nova e a da pessoa mais velha.

**Exercício 3.15.** Um comerciante precisa informatizar o caixa de sua loja. Para isso ele precisa de um programa que leia uma série de valores correspondendo aos preços das mercadorias compradas por um cliente (o valor zero finaliza a entrada), calcule o valor total, subtraia deste valor o desconto devido (vide tabela ao lado) e, finalmente, mostre o valor a ser pago pelo cliente. Codifique esse programa.

<i>Total</i>	<i>Desconto</i>
abaixo de R\$ 50,00	5%
até R\$ 100,00	10%
até R\$ 200,00	15%
acima de R\$ 200,00	20%

**Exercício 3.16.** Faça um programa que calcule o saldo de uma conta bancária tendo como entrada o saldo inicial e uma série de operações de crédito e/ou débito finalizada com o valor zero. O programa deve apresentar como saída o total de créditos, o total de débitos, a C.P.M.F. paga (0,35% do total de débitos) e o saldo final. Veja um exemplo:

```

saldo inicial? 1000.00 ↵
operação? -200.00 ↵
operação? +50.00 ↵
operação? -320.00 ↵
operação? 100.00 ↵
operação? -200.00 ↵
operação? 0 ↵
Total de créditos .....: R$ 150.00
Total de débitos .....: R$ 520.00
C.P.M.F. paga .....: R$ 1.04
Saldo final .....: R$ 628.96

```

**Exercício 3.17.** Usando a tabela de notas musicais abaixo, codifique um programa para simular um piano de sete teclas. [Dica: use as funções *sound()*, *nosound()* e *delay()*, do exemplo 3.9, e o comando *switch*.]

Tecla	Nota	Frequência
A	Dó	262 Hz
S	Ré	294 Hz
D	Mi	330 Hz
F	Fá	349 Hz
J	Sol	392 Hz
K	Lá	440 Hz
L	Si	494 Hz

**Exercício 3.18.** A função *rand()*, definida em *stdlib.h*, gera números aleatórios no intervalo de 0 a 32767. Crie um programa para gerar sons com frequências e durações aleatórias, ininterruptamente, até que o usuário pressione uma tecla. Garanta que a duração de uma nota não seja demasiadamente longa, restringindo o intervalo dos números gerados. [Dica: use o operador de módulo; por exemplo, *rand()%100* gera um número entre 0 e 99.]

### 3.5. INTERROMPENDO UMA REPETIÇÃO

Às vezes é preciso parar uma repetição mesmo antes que sua condição torne-se falsa. Nessas ocasiões, podemos empregar o comando *break* que, quando não está associado ao *switch*, serve para interromper uma repetição.

Por exemplo, suponha que precisamos verificar se um dado número natural é primo ou não. Como sabemos, um número é primo se é divisível apenas por 1 e por ele mesmo. Então, se um número  $n$  é divisível por outro número  $k$ , sendo  $1 < k < n$ , ele não pode ser primo.

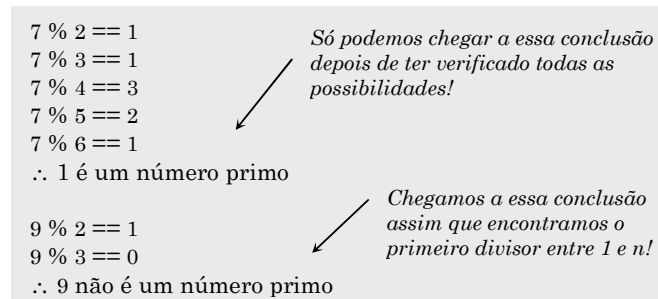


Figura 3.8 – Verificando se um número é primo

Para codificar o procedimento exemplificado acima, podemos usar um *for* para variar o valor do divisor  $k$  de 2 até  $n-1$ . Então, para cada valor atribuído a  $k$ , verificamos se o resto da divisão é 0, ou seja, verificamos se  $k$  é um divisor de  $n$ . Se nenhuma das tentativas resultar em sucesso, como acontece para  $n=7$ , então o número é primo. Entretanto, se em qualquer tentativa conseguirmos uma divisão exata, concluímos que o número não é primo e o processo pode ser interrompido.

**Exemplo 3.10.** Interrompendo uma repetição.

```
#include <stdio.h>

main() {
    int n, k;
    clrscr();
    printf("\nDigite um número natural: ");
    scanf("%u", &n);
    for(k=2; k<=n-1; k++)
        if( n%k == 0 ) break;
    if( k==n ) printf("\nO número é primo");
    else printf("\nO número não é primo");
    getch();
}
```

Observe que o *for* pode ser interrompido tanto pela condição  $k \leq n-1$  quanto pela condição  $n\%k == 0$ . O teste  $k == n$ , fora da repetição, é necessário para se determinar como a repetição foi interrompida. Claramente, se  $k$  chega a assumir o valor  $n$  é porque a repetição *for* não foi interrompida pelo *break* e, conseqüentemente, o número só pode ser primo. ☑

☛ O *break* também pode ser usado para interromper repetições *while* ou *do-while*.

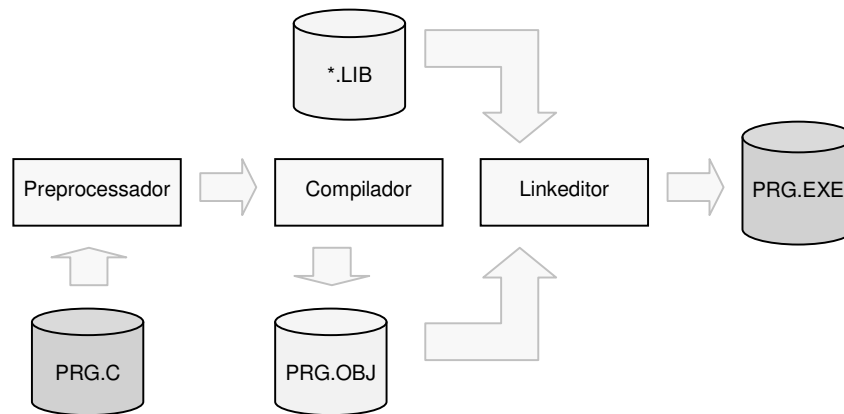
## 4. MACROS E FUNÇÕES

*Este capítulo introduz o conceito de preprocessamento e apresenta as duas principais diretivas utilizadas. Também mostra a criação e o uso de funções em C e discute as classes de armazenamento de variáveis.*

### 4.1. PREPROCESSAMENTO

---

Todo programa C, antes de chegar ao compilador é tratado por um módulo adicional denominado *preprocessador*.



**Figura 4.1** – O processo de geração de um executável

O preprocessador é capaz de realizar uma série de modificações no código-fonte de um programa, antes que ele seja analisado pelo compilador. Essas modificações são especificadas através de *diretivas* de preprocessamento que são embutidas no código-fonte do programa. O preprocessador oferece diversas diretivas, sendo que as principais são *#include* e *#define*.

#### 4.1.1. A DIRETIVA *#define*

A diretiva *#define* serve para definir constantes simbólicas que aumentam a legibilidade do código-fonte. Essa diretiva associa um identificador a um texto da seguinte maneira:

*#define* *identificador* *texto*

**Exemplo 4.1.** Definindo constantes simbólicas.

```
#include <stdio.h>
#define diga printf
#define oi      "\nOlá, tudo bem?"
main() {
    diga(oi);
}
```

Quando o programa passa pelo préprocessador, a diretiva *#define* é executada e, então, toda ocorrência da palavra *diga* é substituída por *printf* e toda ocorrência da palavra *oi* é substituída por “\nOlá, tudo bem?”. ☒

☛ O texto associado ao identificador pode ser inclusive uma palavra reservada.

**Exercício 4.1.** Inclua diretivas *#define* no programa a seguir de modo que ele possa ser compilado corretamente:

```
#include <stdio.h>
programa
inicio
    diga("Olá!");
fim
```

É também possível criar substituições parametrizadas, ou seja *macros*. Por exemplo, a macro a seguir serve para calcular o quadrado de um número *n*:

```
#define quad(n)  n*n
```

☛ Note que não pode haver espaço entre o identificador da macro e o parêntese que delimita a lista de parâmetros; caso haja, teremos um erro de substituição.

Ao passar pelo préprocessador, as ocorrências dessa macro são substituídas pela expressão *n\*n*, com o parâmetro *n* devidamente instanciado:

<i>ocorrência</i>	<i>instância</i>
<i>quad(x)</i>	<i>x * x</i>
<i>quad(2)</i>	<i>2 * 2</i>
<i>quad(f(x-3))</i>	<i>f(x-3) * f(x-3)</i>
<i>quad(x+4)</i>	<i>x+4 * x+4</i>

Observe que nenhum cálculo é realizado durante o préprocessamento e, portanto, ocorre uma simples substituição. Em virtude disso, resultados inesperados podem surgir se não tomamos alguns cuidados ao definir uma macro.

<i>ocorrência</i>	<i>esperado</i>	<i>instância</i>	<i>obtido</i>
<i>quad</i> ( 2+3 )	25	2+3 * 2+3	11
100 / <i>quad</i> (2)	25	100 / 2 * 2	100

A correta definição de uma macro requer que a expressão de substituição esteja completamente parentetizada. Então, uma definição correta para uma macro que calcula o quadrado de um número é:

```
#define quadrado(n) ((n) * (n))
```

<i>ocorrência</i>	<i>instância</i>	<i>obtido</i>
<i>quad</i> ( 2+3 )	((2+3) * (2+3))	25
100 / <i>quad</i> (2)	100 / ((2) * (2))	25

Observe que os parênteses colocados na expressão de substituição são mantidos na instância que é criada durante o préprocessamento e, sendo assim, os valores obtidos serão sempre corretos.

**Exercício 4.2.** Defina as macros descritas a seguir:

- a) *eh\_minuscula*(*c*): informa se o caracter *c* é uma letra minúscula.
- b) *eh\_maiuscula*(*c*): informa se o caracter *c* é uma letra maiúscula.
- c) *minuscula*(*c*): converte a letra *c* para minúscula.
- d) *maiuscula*(*c*): converte a letra *c* para maiúscula.

#### 4.1.2. A DIRETIVA *#include*

Outra diretiva bastante utilizada é *#include*. Essa diretiva, quando executada, faz com que uma cópia do arquivo cujo nome é dado entre < e > seja incluído no código-fonte. Por exemplo, suponha que definimos as macros a seguir e as salvamos num arquivo denominado *macros.h*:

```
#define quad(n)    ( (n)*(n) )
#define abs(n)     ( (n)<0 ? -(n) : (n) )
#define max(x,y)   ( (x)>(y) ? (x) : (y) )
```

Então, toda vez que precisarmos de uma destas macros, não será preciso digitá-las novamente; basta solicitar ao préprocessador que inclua uma cópia do arquivo *macros.h* no início do nosso programa.

**Exemplo 4.2.** Incluindo um arquivo de cabeçalho.

```
#include <stdio.h>
#include <conio.h>
#include "macros.h"

main() {
    int a, b;
    clrscr();
    printf("\nDigite dois números: ");
    scanf("%d %d", &a, &b);
    printf("\no máximo é %d!", max(a,b));
    getch();
}
```



Duas observações devem ser feitas a respeito da inclusão de arquivos:

- Qualquer arquivo, com qualquer extensão, pode ser incluído num programa fonte através da diretiva *#include*.
- A notação < e > é preferencialmente utilizada para arquivos de inclusão padrão da linguagem C. Para incluir arquivos definidos pelo usuário, utilize a notação " e ", conforme se observa no programa acima.

**Exercício 4.3.** Crie um arquivo com as macros definidas no exercício 4.2 e faça um programa que use esse arquivo para testar essas macros.

**Exercício 4.4.** Crie o arquivo *boolean.h* com as definições necessárias para que o programa a seguir possa produzir a saída: *true false true*.

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

main() {
    printf("%s ", bool(not false) );
    printf("%s ", bool(false and true) );
    printf("%s ", bool(true or false) );
}
```

## 4.2. DEFINIÇÃO E USO DE FUNÇÕES

---

Até agora, em todos os programas que criamos, codificamos uma única função: *main()*. Entretanto, em todos eles, diversas funções foram utilizadas: *printf()*, *scanf()*, *getch()*, *putch()*, etc. Essas funções estão disponíveis no sistema através de bibliotecas que acompanham o compilador *Turbo C*, mas podemos definir nossas próprias funções e utilizá-las da mesma maneira.



Para definir uma função, empregamos a seguinte forma básica:

```
tipo nome(parâmetros) {  
    declarações  
    comandos  
}
```

sendo que:

- *tipo* refere-se ao tipo de resposta que a função devolve e deve ser *void* (*vazio*) se a função não tem valor de resposta;
- *nome* é o identificador da função no resto do programa;
- *parâmetros* é uma lista de variáveis que representam valores de entrada para a função e deve ser *void* caso não haja valores de entrada;
- dentro do corpo da função, a primeira seção é destinada à declaração das variáveis e a segunda, aos comandos.

#### 4.2.1. FUNÇÕES QUE NÃO DEVOLVEM RESPOSTA

Nosso primeiro exemplo é uma função que não tem valor de resposta e não recebe argumentos ao ser chamada. Esta função tem como objetivo simular o som de um alarme e poderia ser utilizada, por exemplo, num programa que desejasse alertar o usuário sobre alguma operação indevida.

**Exemplo 4.3.** Simulando o som de um alarme.

```
void alarme(void) {  
    int f;  
    for(f=100; f<=5000; f+=20) {  
        sound(f);  
        delay(100);  
    }  
    nosound();  
}
```



Uma função do tipo *void*, quando executada, apenas produz um determinado efeito desejado, sem contudo devolver um valor de resposta. Sendo assim, o compilador não permite que funções *void* sejam usadas em expressões. Por exemplo, a chamada a seguir causaria um erro de compilação:

*x* = *alarme*() + 3;

Da mesma forma, se a lista de parâmetros de uma função é *void*, o compilador não permite que sejam passados argumentos à ela. Por exemplo, a chamada a seguir também causaria um erro de compilação:

*alarme*(5);

Uma vez codificada uma função, podemos utilizá-la como qualquer outra função previamente definida na linguagem C.

**Exemplo 4.4.** Usando a função *alarme()*.

```
#include <stdio.h>
#include <conio.h>
#define SENHA 1234

void alarme(void); /* declaração da função */

void main(void) {
    int s;

    clrscr();
    printf("\nSenha: ");
    scanf("%d", &s);

    if( s != SENHA ) {
        printf("\nSenha inválida!");
        alarme();
    }
    else printf("\nSenha ok!");

    getch();
}

void alarme(void) { /* definição da função */
    int f;

    for(f=100; f<=5000; f+=20) {
        sound(f);
        delay(100);
    }

    nosound();
}
```



*Para que uma função seja reconhecida durante a compilação devemos declará-la ou defini-la antes de qualquer referência que é feita a ela no resto do programa.*

A declaração de uma função, conhecida como *assinatura* ou *protótipo*, consiste em seu *tipo*, *nome* e lista de *parâmetros* seguida de ponto-e-vírgula. A declaração pode ser omitida se a definição da função ocorre antes que ela seja referenciada no restante do programa. Por exemplo, no programa acima, a declaração da função *alarme()* torna-se desnecessária se colocarmos a sua definição antes da definição da função *main()*. Em geral, por uma questão de comodidade, vamos preferir não usar protótipos e, assim, teremos que ordenar as definições em função das dependências existentes entre elas.


Vale lembrar que uma das finalidades dos arquivos de inclusão é manter os protótipos das funções pertencentes às bibliotecas que eles representam.

Nosso próximo exemplo é uma função que nos permite desenhar uma linha horizontal, a partir de uma determinada posição da tela, com um certo comprimento. Para isso, será necessário que as coordenadas da posição em questão, bem como o comprimento desejado, sejam passados à rotina sob a forma de uma lista de argumentos.

**Exemplo 4.5.** Desenhando linhas.

```
void linha(int x, int y, int c) {  
    int i;  
    gotoxy(x,y);  
    for(i=0; i<c; i++ ) putchar(196);  
}
```

A função inicia movendo o cursor para a posição indicada pelos parâmetros  $x$  e  $y$  e, em seguida, exibe  $c$  vezes o caracter ‘—’, cujo código ASCII é 196. ☒

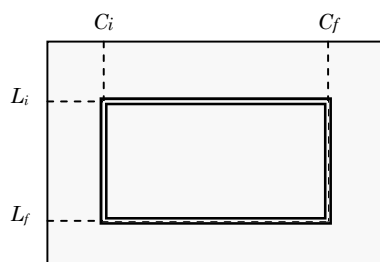
 *Note que, mesmo que todos os parâmetros sejam do mesmo tipo, a linguagem C exige que cada um deles seja declarado independentemente dos demais.*

A vantagem de se ter uma função parametrizada é que, a cada chamada, podemos lhe passar valores diferentes. Isto permite, no caso da função *linha()*, que as linhas possam ter qualquer comprimento viável e que sejam exibidas em qualquer ponto da tela que julgarmos adequado.

**Exercício 4.5.** Codifique a função *rodizio(placa)*, que recebe o número da placa de um veículo e exibe o dia em ele está no rodízio.

**Exercício 4.6.** Codifique a função *retangulo(m,n)*, que exibe um retângulo com altura  $m$  e largura  $n$ . [Dica: veja o exercício 3.9]

**Exercício 4.7.** Codifique a função *moldura(Ci,Li,Cf,Lf)*, que exibe uma moldura na tela conforme ilustrado a seguir. Utilize a diretiva *#define* para definir nomes para os caracteres utilizados na montagem da moldura.



#### 4.2.2. FUNÇÕES QUE DEVOLVEM RESPOSTA

Se uma função não é do tipo *void*, então ela deve, necessariamente, devolver um valor como resultado de sua execução. Para isso, a função deve empregar o comando *return*. Esse comando, além de especificar a resposta da função, faz com que o controle retorne ao ponto onde ela foi chamada no programa, interrompendo imediatamente sua execução.

**Exemplo 4.6.** Calculando a hipotenusa de um triângulo retângulo.

```
float hip(float a, float b) {  
    float h;  
    h = sqrt( pow(a,2)+pow(b,2);  
    return h;  
}
```

Essa função recebe as medidas dos catetos de um triângulo retângulo e devolve como resposta a medida da sua hipotenusa. ☑

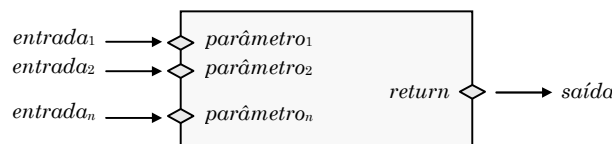


Figura 4.2 – Entradas e saída de uma função

☛ *Um comando *return* só devolve um único valor e, como ele interrompe a execução da função, não há como criar uma função que devolva mais que um valor por vez.*

Mesmo quando o comando *return* aparece em diversos pontos dentro de uma função, apenas uma dessas ocorrências pode ser executada a cada vez que a função é chamada. Como exemplo, considere a função a seguir que devolve o máximo entre os dois valores que lhe são passados como argumentos.

**Exemplo 4.7.** Determinando o máximo entre dois valores.

```
double max(double a, double b) {  
    if( a>b ) return a;  
    return b;  
}
```

Note que a execução do primeiro *return* impede a execução do segundo, mesmo sem termos colocado a parte *else* do *if*. ☑

**Exercício 4.8.** Codifique a função  $fat(n)$ , que devolve o fatorial de  $n$ .

**Exercício 4.9.** Codifique a função  $pot(x,n)$ , que devolve  $x$  elevado a  $n$ .

**Exercício 4.10.** Codifique a função  $quad(n)$ , que devolve o quadrado de  $n$  usando o método da soma de ímpares. [Dica: veja o exercício 3.7]

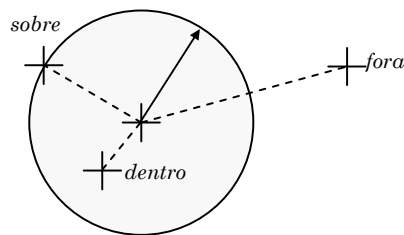
**Exercício 4.11.** Codifique uma função que receba um número real  $n$  e devolva sua raiz quadrada  $r$ . Para calcular  $r$ , use o método proposto por *Newton*:

1º chute-se um valor inicial para a raiz igual a 1;

2º caso  $|r^2 - n|$  seja inferior a 0.001,  $r$  é a resposta (fim);

3º caso contrário, aproxima-se  $r = (r^2 + n) / (2r)$  e retorna-se ao 2º passo

**Exercício 4.12.** São dadas as coordenadas  $(x_c, y_c)$  do centro de uma circunferência e a medida  $r$  de seu raio. Também são dadas as coordenadas  $(x, y)$  de uma série de pontos, sendo que o último deles é igual ao centro. Determine quantos pontos desta série estão dentro da circunferência, quantos estão fora e quantos estão sobre ela. Crie a função  $dist(x_1, y_1, x_2, y_2)$  que dá a distância entre os pontos  $(x_1, y_1)$  e  $(x_2, y_2)$  e, depois, utilize-a num programa que resolva o problema proposto.



**Exercício 4.13.** Codifique a função  $dv(n)$  que recebe um número  $n$  e devolve o seu dígito verificador. Essa função deve implementar o seguinte método:

Suponha  $n = 345702159$ .

1º calculamos  $s = 3*10 + 4*9 + 5*8 + 7*7 + 0*6 + 2*5 + 1*4 + 5*3 + 9*2 = 202$ .

2º calculamos  $x = 11 - s \% 11 = 11 - 202 \% 11 = 11 - 4 = 7$ .

3º o dígito verificador é 0 se  $x > 9$  e é o próprio  $x$ , caso contrário. Então,  $d = 7$ .

**Exercício 4.14.** Codifique a função  $cpf(n,d)$  que devolve verdade só se o CPF  $n$  tem dígito verificador  $d$ . Use o método descrito no exercício anterior para calcular o dígito verificador do CPF do seguinte modo:

Suponha  $CPF = 345702159$ .

1º calculamos o primeiro dígito  $a = dv(345702159) = 7$ .

2º calculamos o segundo dígito  $b = dv(3457021597) = 1$ .

Então, número completo do CPF é 345702159-71.

### 4.3. CLASSES DE ARMAZENAMENTO

---

A *classe de armazenamento* especifica dois atributos importantes de uma variável: quando ela será criada e destruída (*duração*) e em que parte do programa ela estará acessível (*escopo*).

As principais classes de armazenamento em *C* são:

- *Local ou Automática*

Uma variável é *local* se é declarada dentro de uma função. As variáveis locais somente são criadas no momento em que a função entra em execução e deixam de existir tão logo a execução seja concluída. Assim, essas variáveis são acessíveis apenas à função em que são declaradas. Variáveis locais também são denominadas *automáticas*, já que são automaticamente criadas quando a função é chamada e automaticamente destruídas quando a função termina. Podemos usar a palavra *auto* prefixando o tipo de uma variável para indicar que ela é automática; entretanto, como essa é a classe *default*, raramente isso é feito.



- *Global ou Externa*

Uma variável é *global* se é declarada fora de qualquer função. As variáveis globais são criadas no início da execução do programa e somente deixam de existir quando ele termina. Assim, essas variáveis são acessíveis a todo o programa, a partir do ponto em que são declaradas. Variáveis globais também são denominadas *externas*, já que são declaradas externamente às funções. A palavra *extern* pode ser usada para indicar que uma variável é externa, mas raramente usaremos variáveis globais.



- *Estática*

Uma variável *estática* tem o escopo de uma local e a duração de uma global. Isso quer dizer que essas variáveis só são acessíveis à função que as declara, mas existem durante toda a execução do programa. Variáveis estáticas têm a vantagem de serem privativas e, ao mesmo tempo, de manterem seu valor entre uma chamada e outra da função. A palavra *static* deve prefixar a declaração de variáveis estáticas.



- *Registrador*

Uma variável *registrador* é armazenada diretamente num registrador da CPU, agilizando assim o acesso ao seu valor. Apenas variáveis *char* e *int* podem ser dessa classe. Podemos usar a palavra *register* prefixando a declaração de variáveis que desejamos manter em registradores; entretanto, nos compiladores modernos essa otimização, sempre que possível, é feita automaticamente pelo compilador.

Temos a seguir uma tabela que resume estes conceitos:

<i>classe</i>	<i>palavra</i>	<i>escopo</i>	<i>duração</i>	<i>exemplo</i>
<i>local</i>	<i>auto</i>	<i>função</i>	<i>função</i>	<i>auto int n;</i>
<i>global</i>	<i>extern</i>	<i>programa</i>	<i>programa</i>	<i>extern int n;</i>
<i>estática</i>	<i>static</i>	<i>função</i>	<i>programa</i>	<i>static int n;</i>
<i>registrador</i>	<i>register</i>	<i>função</i>	<i>função</i>	<i>register int n;</i>

De todas as classes de armazenamento, as mais usadas são as classes *auto* e *static*. Como até agora todas as variáveis que usamos são, por *default*, da classe *auto*, nosso próximo exemplo ilustrará o uso de variáveis estáticas.

Vamos criar uma função para gerar números aleatórios conforme exemplificado ao lado. A *semente* é um número qualquer de quatro dígitos escolhido para iniciar o processo de geração da série. No exemplo, escolhemos o número 1234 para ser a semente. A cada execução, tomamos os dois últimos dígitos da semente para ser o número aleatório e alteramos o seu valor de modo que na próxima execução tenhamos um novo número aleatório.

```
Semente => 1234 +
              123
              1357 +
              135
              1492 +
              149
              1641 +
              164
              1805 +
              ...
```

**Exemplo 4.7.** O uso de variáveis estáticas.

```
int aleat(void) {
    static unsigned s = 1234;
    auto unsigned n = s%100;
    s += s/10;
    return n;
}
```



Uma variável estática é inicializada uma única vez, no momento em que é criada.

**Exemplo 4.8.** Exibindo uma série de números aleatórios.

```
#include <stdio.h>
#include <conio.h>
...
void main(void) {
    while( !kbhit() )
        printf("\n%d", aleat() );
}
```



Embora a função *aleat()* devolva números no intervalo de 0 a 99, é possível restringir estes valores a intervalos menores utilizando-se o resto da divisão.

**Exemplo 4.9.** Simulando um céu estrelado.

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

...

void main(void) {
    int col, lin, cor;
    clrscr();
    while( !kbhit() ) {
        col = aleat()%80 + 1; /* valores entre 1 e 80 */
        lin = aleat()%25 + 1; /* valores entre 1 e 25 */
        cor = aleat()%16;     /* valores entre 0 e 15 */

        gotoxy(col,lin);
        textcolor(cor);
        putchar('*');
    }
}
```



**Exercício 4.15.** Uma fraqueza da função *aleat()* é que, como a semente inicial é sempre a mesma, os números aleatórios gerados são sempre os mesmos. Para melhorar seu desempenho, seria interessante que a semente inicial também fosse aleatória. Para isso, podemos usar a função *time()*, definida em *time.h*. Quando chamamos *time(&t)*, sendo *t* uma variável do tipo *long int*, essa função armazena em *t* o número de segundos que se passaram desde 1 de janeiro de 1970. Altere a função *aleat()* de modo que o valor inicial da semente seja os últimos 4 dígitos do valor devolvido por *time()*. Tome o cuidado de não deixar a semente tornar-se 0 pois, nesse caso, todos os números gerados subsequentemente também serão 0.

#### 4.4. RECURSIVIDADE

---

A *recursividade* é um princípio que nos permite obter a solução de um problema a partir da solução de uma instância menor de si mesmo. Para aplicar esse princípio devemos supor que a solução da instância menor é conhecida. Por exemplo, suponha que desejamos calcular a potência  $2^{11}$ . Uma instância menor desse problema é  $2^{10}$  e, para essa instância, "sabemos" que a solução é 1024. Então, como  $2 \times 2^{10} = 2^{11}$ , concluímos que  $2^{11} = 2 \times 1024 = 2048$ .



A figura 4.3 ilustra o princípio de recursividade. De modo geral, procedemos da seguinte maneira: simplificamos o *problema original* transformando-o numa *instância menor*; então, *obtemos a solução* para essa instância e a *usamos* para construir a *solução final*, correspondente ao problema original.

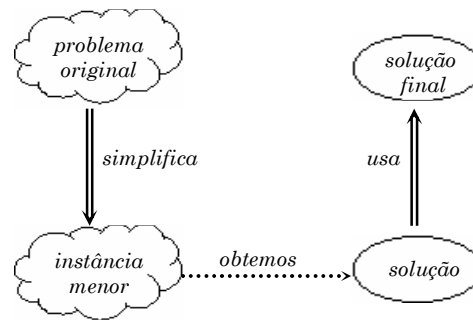


Figura 4.3 – O princípio de recursividade

A parte mais difícil de entender é como obtemos a solução para a instância menor. Mas, isso é justamente a função da *recursão*: é para isso que ela serve, para resolver a instância menor e nos fornecer a sua solução. Não precisamos nos preocupar com essa parte. Tudo o que precisamos fazer é encontrar uma simplificação que seja adequada para o problema em questão e descobrir uma maneira de usar a solução obtida por recursão para construir sua solução final.

Em computação, o uso de recursividade requer a definição de *funções recursivas*, i.e. funções que chamam a si mesmas.

**Exemplo 4.10.** Uma função recursiva descontrolada.

```
void loop(void) {
    printf("loop ");
    loop();
}
```

Teoricamente, uma chamada à função *loop()* desencadearia um processo que ficaria, infinitamente, exibindo a palavra "loop". Note que, a cada chamada da função *loop()*, a função *printf()* é executada e, em seguida, a própria função *loop()* é chamada novamente, *ad infinitum*. ☑

☞ Para ser útil, uma função recursiva deve ter um ponto de parada, ou seja, deve ser capaz de interromper as chamadas recursivas e executar em tempo finito.

#### 4.4.1. FUNÇÕES RECURSIVAS

Ao definir uma função recursiva devemos identificar:

1º *a base da recursão*, i.e. a instância mais simples do problema em questão.

2º *o passo da recursão*, i.e. como simplificar o problema em questão.

A *base* trata o caso mais simples, para o qual temos uma solução trivial, e o *passo* trata os casos mais difíceis, que requerem novas chamadas recursivas. Em geral, o passo da recursão engloba também a construção da solução final a partir da solução do problema simplificado. Retomando o exemplo do cálculo da potência, temos a seguinte definição recursiva:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x.x^{n-1}, & \text{caso contrário} \end{cases}$$

A base para o problema da potência é aquele em que o expoente é 0, pois qualquer número elevado a 0 é igual a 1. Por outro lado, se o expoente é maior que 0, então o problema deve ser simplificado, ou seja, temos que chegar um pouco mais perto da base. A chamada recursiva com o argumento  $n-1$  garante justamente isso. Então, após um número finito de passos, atingimos a base da recursão e, assim, obtemos o resultado esperado.

**Exemplo 4.11.** Cálculo da potência.

```
double pot(double x, unsigned n) {  
    if( n==0 ) return 1;  
    return x * pot(x,n-1);  
}
```



Uma maneira de entender o funcionamento das funções recursivas é através de *simulação por substituição*. Nessa simulação, substituímos a chamada da função pela expressão que ela devolve, até que toda chamada recursiva tenha sido substituída por uma expressão livre de chamadas recursivas.

**Exemplo 4.12.** Simulação por substituição.

```
p = pot(2,3)  
= 2 * pot(2,2)  
= 2 * 2 * pot(2,1)  
= 2 * 2 * 2 * pot(2,0)  
= 2 * 2 * 2 * 1  
= 8
```



Uma outra maneira de acompanhar o funcionamento de uma função recursiva é desenhar o *fluxo de chamadas e retornos* da função.

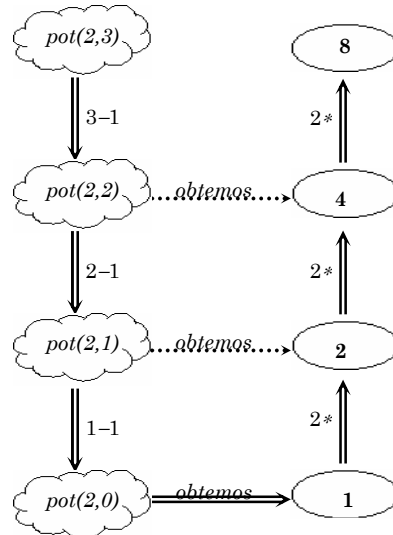


Figura 4.4 – Fluxo de chamadas e retornos para  $pot(2,3)$

A cada expansão, deixamos a oval em branco e rotulamos seta que sobe com a operação que fica pendente. Quando a base da recursão é atingida, começamos a preencher as ovas, propagando os resultados de baixo para cima e efetuando as operações pendentes. O caminho seguido é aquele indicado pelas setas duplas. As setas pontilhadas representam as hipóteses de recursão, que nos permitem assumir que temos a solução para cada uma das instâncias menores do problema.

**Exercício 4.16.** Para cada problema a seguir defina uma função recursiva, faça a simulação por substituição e desenhe o fluxo de chamadas e retornos:

- Calcular o fatorial de um número natural.
- Calcular o resto da divisão inteira usando subtração.
- Calcular o quociente da divisão inteira usando subtração.
- Calcular o produto de dois naturais usando adição.
- Calcular a soma de dois naturais usando as funções  $suc(n)$  e  $pred(n)$  que devolvem, respectivamente, o sucessor e o predecessor de um natural  $n$ .

**Exercício 4.17.** É importante conseguir "visualizar" um processo recursivo para que possamos entendê-lo bem. A simulação por substituição e o fluxo de chamadas e retornos são ferramentas úteis para essa finalidade, mas são muito trabalhosas. Outra possibilidade é *rastrear automaticamente*, ou seja, modificamos o código para que ele mesmo exiba as chamadas que são feitas e os valores que são devolvidos em cada etapa do processo recursivo. O

programa a seguir implementa rastreamento automático. Execute-o para ver como ele funciona e, inspirando-se nele, modifique o código das funções definidas no exercício anterior para implementar rastreamento automático.

```
/* rastreamento automático */
#include <stdio.h>
#include <conio.h>

void chama(float x, int n, int p) {
    int i;
    for(i=0; i<p; i++) putchar(' ');
    printf("pot(%.1f,%d)\n",x,n);
}

void devolve(float r, int p) {
    int i;
    for(i=0; i<p; i++) putchar(' ');
    printf("%.1f\n",r);
}

float pot(float x, int n, int p) {
    int r;
    chama(x,n,p);
    if(n==0) r = 1;
    else    r = x*pot(x,n-1,p+1);
    devolve(r,p);
    return r;
}

void main(void) {
    clrscr();
    pot(2,5,0);
    getch();
}
```

#### 4.4.2. PROCEDIMENTOS RECURSIVOS

Procedimentos recursivos são definidos basicamente da mesma forma que as funções recursivas. A única diferença é que, como eles não devolvem resposta, não precisamos usar o comando *return* com as chamadas recursivas.

Vamos começar com um procedimento recursivo bem simples que exibe uma contagem progressiva. Por exemplo, a chamada *prog(3)* produz a saída 1 2 3.

Sendo *n* o parâmetro desse procedimento, podemos defini-lo assim:

- *base*:  $n=0 \rightarrow$  nada a fazer
- *passo*:  $n>0 \rightarrow$  exibe a contagem progressiva para  $n-1$  e, depois, exibe  $n$ .

**Exemplo 4.13.** Contagem progressiva por recursão.

```
void prog(int n) {  
    if( n==0 ) return;  
    prog(n-1);  
    printf("%d ",n);  
}
```



O funcionamento dos procedimentos recursivos somente podem ser acompanhados pelo fluxo de chamadas e retornos. Para *prog()* o fluxo é o seguinte:

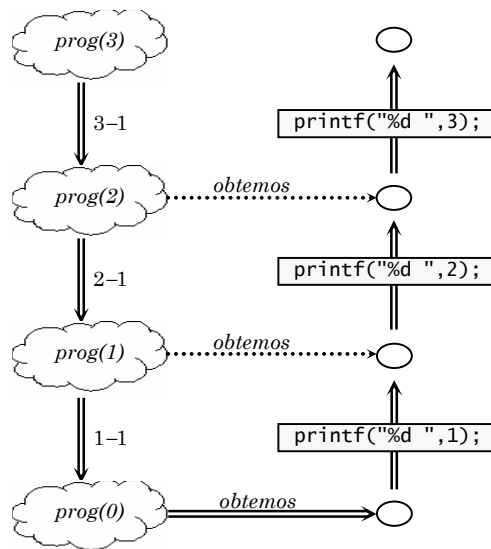


Figura 4.4 – Fluxo de chamadas e retornos para *prog(3)*

Cada vez que uma chamada recursiva a *prog()* é feita, a instrução *printf()* que vem em seguida fica pendente, sendo executada somente quando o controle retorna das chamadas recursivas. Instruções pendentes são mantidas numa estrutura de pilha, de modo que a última delas é a primeira a ser executada. Isso faz com que os valores sejam exibidos na ordem inversa àquela em foram recebidos pela função *prog()*. Além disso, como essa função é *void*, os pontos de retorno no fluxo ficam vazios e o único resultado que se obtém é o efeito produzido pela execução das instruções pendentes.

**Exercício 4.18.** Defina os seguintes procedimentos recursivos:

- regr(n)*, que exibe uma contagem regressiva a partir de *n*.
- bin(n)*, que exibe o número natural *n* em binário.

## 5. VETORES, STRINGS E MATRIZES

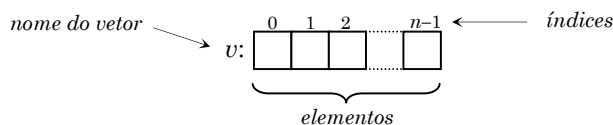
*O vetor é provavelmente um dos mais simples e importantes tipos agregados disponíveis na maioria das linguagens de programação. Através do seu uso, podemos armazenar e manipular grandes quantidades de dados.*

*Nesse capítulo, introduzimos o uso de vetores em C, mostramos como strings e matrizes são implementadas a partir deles e apresentamos alguns métodos de ordenação e busca em vetores.*

### 5.1. VETORES

---

Um *vetor* é uma coleção de variáveis de um mesmo tipo, que compartilham o mesmo nome e que ocupam posições consecutivas de memória. Cada uma dessas variáveis denomina-se *elemento* e é identificada por um *índice*. Se  $v$  é um vetor com  $n$  posições, seus elementos são  $v[0]$ ,  $v[1]$ ,  $v[2]$ , ...,  $v[n-1]$ .



**Figura 5.1** – Um vetor e seus elementos

☛ *Em C os vetores são sempre indexados a partir de zero e, portanto, o último elemento de um vetor de tamanho  $n$  ocupa a posição  $n-1$  do vetor.*

Para criar um vetor, basta declarar uma variável com sufixo  $[n]$ , sendo  $n$  uma constante indicando o número de elementos a serem alocados no vetor.

**Exemplo 5.1.** Um vetor para armazenar 5 números inteiros pode ser criado da seguinte maneira:

```
int v[5];
```

A palavra *int* indica que o vetor  $v$  é um grupo de variáveis inteiras e o sufixo  $[5]$  especifica que esse grupo possui cinco elementos. Como em C os vetores são indexados a partir de 0, os elementos de  $v$  são  $v[0]$ ,  $v[1]$ ,  $v[2]$ ,  $v[3]$  e  $v[4]$ . ☑

**Exercício 5.1.** Crie tipos de vetores para armazenar:

- a) as letras vogais do alfabeto.
- b) as temperaturas diárias de uma semana.
- c) o número de dias em cada um dos meses do ano.

Em geral, um vetor  $v$  pode ser indexado com qualquer expressão cujo valor seja de tipo integral<sup>2</sup>. Essa expressão pode ser uma simples constante, uma variável ou então uma expressão propriamente dita, contendo operadores, constantes e variáveis.

**Exemplo 5.2.** Seja  $i$  uma variável do tipo *int* e  $v$  o vetor criado no exemplo anterior. Se  $i=3$ , então  $v[i\%3] \equiv v[0]$ ,  $v['B'-'A'] \equiv v[1]$ ,  $v[i-1] \equiv v[2]$ ,  $v[i] \equiv v[3]$ ,  $v[i+1] \equiv v[4]$ . Entretanto,  $v[i/2.0]$  causará um erro de compilação; já que a expressão  $i/2.0$  tem valor igual a 1.5, que não é um índice permitido.  $\square$

**Exercício 5.2.** Seja  $w$  um vetor contendo 9 elementos inteiros. Supondo que  $i$  seja uma variável inteira valendo 5, que valores estarão armazenados em  $w$  após as atribuições a seguir?

- |                      |                             |
|----------------------|-----------------------------|
| ① $w[0] = 17;$       | ⑤ $w[i] = w[2];$            |
| ② $w[i/2] = 9;$      | ⑥ $w[i+1] = w[i] + w[i-1];$ |
| ③ $w[2*i-2] = 95;$   | ⑦ $w[w[2]-2] = 78;$         |
| ④ $w[i-1] = w[8]/2;$ | ⑧ $w[w[i]-1] = w[1]*w[i];$  |

Ao contrário da maioria das linguagens,  $C$  não faz consistência dos valores usados como índices num vetor. Isso quer dizer que qualquer valor integral pode ser usado como índice, mesmo que tal valor não seja adequado. Esse uso indevido, entretanto, pode causar resultados inesperados durante a execução do programa ou até mesmo travar o computador.

**Exemplo 5.3.** O programa a seguir ilustra o uso de índices inadequados<sup>3</sup>:

```
#include <stdio.h>
void main(void) {
    int x[3], y[4];
    x[2] = y[0] = 1;
    x[3] = 2;           /* x[+3] sobrepõe y[0] */
    y[-1] = 3;          /* y[-1] sobrepõe x[2] */
    printf("%d %d", x[2], y[0]);
}
```

---

<sup>2</sup> São integrais os tipos *char*, *int* e seus derivados como, por exemplo, *unsigned char* e *long int*.

<sup>3</sup> Esse exemplo supõe que as variáveis  $x$  e  $y$  sejam criadas em posições consecutivas de memória, de modo que a última posição de  $x$  seja adjacente à primeira de  $y$ , como ocorre no Turbo C<sup>®</sup>.

Embora os elementos  $x[3]$  e  $y[-1]$  não tenham sido alocados, o compilador não aponta erros no programa. Porém, quando executado, esse programa produz, surpreendentemente, a saída: 3 2. ☑



*O compilador C não faz consistência da indexação de vetores, deixando essa responsabilidade a cargo do programador.*

**Exercício 5.3.** Codifique um programa para solicitar 5 números, via teclado, e exibi-los na ordem inversa àquela em que foram fornecidos.

**Exercício 5.4.** Implemente um algoritmo para calcular o desvio padrão  $\delta$  de uma coleção de  $n$  números reais, usando a seguinte fórmula:

$$\delta = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}}, \text{ onde } \bar{x} = \frac{1}{n} \times \sum_{i=1}^n x_i.$$

**Exercício 5.5.** Dados os coeficientes  $a_0, a_1, a_2, \dots, a_n$  de um polinômio  $p(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n$ , e uma seqüência de valores de  $x$ , avaliar o polinômio para cada um dos valores de  $x$ .

### 5.1.1. INICIALIZAÇÃO DE VETORES

Em C, vetores globais e estáticos são automaticamente zerados pelo compilador. Mas, se for desejado, podemos inicializá-los explicitamente no momento em os declaramos<sup>4</sup>. Nesse caso, os valores iniciais devem ser fornecidos entre chaves e separados por vírgulas.

**Exemplo 5.4.** Inicialização de vetor estático:

```
#include <stdio.h>
void main(void) {
    static float moeda[5] = {1.00, 0.50, 0.25, 0.10, 0.05};
    ...
}
```

Os valores são armazenados, a partir da posição 0 do vetor, na ordem em que são fornecidos; por exemplo, o valor 0.25 é armazenado em *moeda*[2]. ☑

Note que apenas expressões e valores constantes são permitidas numa lista de valores iniciais. O uso de variáveis causa erro de compilação.

---

<sup>4</sup> Alguns compiladores não permitem a inicialização de vetores da classe *auto*.



Se a lista de valores iniciais tem mais elementos que a capacidade do vetor, ocorre um erro de compilação. Entretanto, se tem menos que o necessário, as posições excedentes do vetor permanecem zeradas.

**Exemplo 5.5.** Inicialização de vetor com valores iniciais insuficientes:

```
#include <stdio.h>

#define max 5

void main(void) {
    static int A[max] = {9, 3, 2, 7};
    auto int i;
    for(i=0; i<max; i++)
        printf("%d", A[i]);
}
```

Os valores iniciais 9, 3, 2 e 7 são armazenados em  $A[0]$ ,  $A[1]$ ,  $A[2]$  e  $A[3]$ , respectivamente, permanecendo as demais posições,  $A[4]$  e  $A[5]$ , zeradas. ☒

Quando um vetor é inicializado, o seu tamanho pode ser omitido. Nesse caso, o compilador determina o tamanho do vetor contando os elementos fornecidos na lista de valores iniciais.

**Exemplo 5.6.** Vetores de tamanho implícito:

```
#include <stdio.h>

void main(void) {
    static char ds[] = {'D', 'S', 'T', 'Q', 'Q', 'S', 'S'};
    ...
}
```

Como o tamanho é omitido, o compilador cria o vetor *ds* com 7 posições. ☒

**Exercício 5.6.** Codifique um programa que indique a quantidade mínima de cédulas equivalente a uma dada quantia em dinheiro. Considere apenas valores inteiros e cédulas de 1, 5, 10, 50 e 100 reais.

```
Quantia? R$ 209.␣
2 cédulas de R$100,00
1 cédula de R$5,00
4 cédulas de R$1,00
```

**Exercício 5.7.** Os pares (749,400), (749,400), (841,400), (749,400), (1000,400), (844,800), (749,400), (749,400), (841,400), (749,400), (1122,400), (1000,800), (749,400), (749,400), (1498,400), (1260,400), (1000,400), (944,400), (841,800), (1335,400), (1335,400), (1260,400), (1000,400), (1122,400) e (1000,800) representam as frequências e durações de cada uma das notas de uma canção. Usando as funções *sound()*, *delay()* e *nosound()*, definidas em *conio.h*, crie um programa para tocá-la!

### 5.1.2. PARÂMETROS DO TIPO VETOR

Um fato interessante a respeito de vetores em C é que o nome de um vetor representa o endereço em que ele está alocado na memória. Então, se for desejado saber o endereço de um vetor  $v$ , em vez de escrever  $\&v[0]$ , podemos escrever simplesmente  $v$ .

**Exemplo 5.7.** Endereços de vetores declarados consecutivamente:

```
#include <stdio.h>
void main(void) {
    int x[3], y[4];
    printf("\n x = %p e y = %p", x, y);
}
```

O formato `%p` é usado em C para a exibição de endereços<sup>5</sup>. Quando executado, esse programa exibe<sup>6</sup> `x=FFD0` e `y=FFD6`. Isso mostra que os vetores  $x$  e  $y$  realmente ocupam posições consecutivas de memória, como havíamos previsto no exemplo 5.3. ☑



*Em C, o nome de um vetor representa o endereço em que ele está na memória.*

A importância desse fato é que, quando passamos um vetor como argumento a uma função, estamos na verdade passando o seu endereço. E, ao contrário do que ocorre com outros tipos, a passagem de vetores é feita por referência<sup>7</sup>.

Para ilustrar o uso de parâmetros do tipo vetor, vamos considerar o seguinte problema: *"Dadas as temperaturas registradas diariamente, durante uma semana, determine em quantos dias a temperatura esteve acima da média."*

A solução desse problema envolve os seguintes passos:

- ① obter os valores que representam as temperaturas;
- ② calcular a média entre esses valores;
- ③ verificar quantos desses valores são maiores que a média.

Cada um desses passos representa um subproblema cuja solução contribui para a solução do problema originalmente proposto. Então, supondo que eles já estivessem solucionados, o programa poderia ser codificado como segue:

---

<sup>5</sup> Endereços também são conhecidos como ponteiros, daí o formato `%p`.

<sup>6</sup> Em seu computador os valores podem ser outros, mas a diferença entre eles deve indicar que são consecutivos. Por exemplo, `FFD6-FFD0=6` bytes, exatamente o espaço que o vetor  $x$  ocupa.

<sup>7</sup> A passagem de outros tipos de dados por referência é vista no capítulo 3.

**Exemplo 5.8.** Determina dias da semana com temperatura acima da média.

```
#include <stdio.h>
#define max 7
void main(void) {
    float temp[max], m;
    obtem(temp);
    m = media(temp);
    printf("Estatística: %d", conta(temp,m) );
}
```

O programa ficou extremamente simples; mas, para executá-lo, precisamos antes definir as rotinas *obtem()*, *media()* e *conta()*. ☒

Observe que, na chamada feita em *main()*, apenas o nome do vetor está sendo passado à função *obtem()*. Como o nome de um vetor representa o seu endereço, o que estamos passando à função *obtem()* é, na verdade, o endereço que o vetor *temp* ocupa na memória.

☛ *Em C, todos os parâmetros são passados por valor; exceto vetores, que são sempre passados por referência.*

Conhecendo o endereço do vetor *temp*, a função *obtem()* poderá acessá-lo e alterá-lo diretamente. Além disso, qualquer alteração nele feita será mantida, mesmo após a função ter terminado sua execução.

**Exemplo 5.9.** Obtém dados via teclado e os armazena no vetor.

```
void obtem(float t[]) {
    int i;
    puts("Informe as temperaturas: ");
    for(i=0; i<max; i++) {
        printf("%dº valor? ", i+1);
        scanf("%f", &t[i] );
    }
}
```

Como o parâmetro *t* da rotina *obtem()* é apenas uma referência ao argumento *temp*, seu tamanho não precisa ser informado<sup>8</sup> dentro dos colchetes. Note, porém, que a constante *max*, definida no programa principal, está sendo usada para controlar o acesso aos elementos do vetor. ☒

---

<sup>8</sup> O tamanho é necessário para que o compilador saiba quanto espaço deve ser alocado para o vetor. Como na passagem por referência o vetor não é duplicado e, portanto, nenhum espaço adicional de memória é alocado, o seu tamanho não precisa ser informado.

**Exemplo 5.10.** Calcula a média dos elementos armazenados no vetor.

```
float media(float t[]) {  
    int i;  
    float s=0;  
    for(i=0; i<max; i++)  
        s += t[i];  
    return s/max;  
}
```



**Exemplo 5.11.** Conta os dias com temperatura acima da média.

```
int conta(float t[], float m) {  
    int i, c=0;  
    for(i=0; i<max; i++)  
        if( t[i]>m )  
            c++;  
    return c;  
}
```



**Exercício 5.8.** Digite as funções apresentadas e teste o funcionamento do programa que resolve o problema das temperaturas acima da média.

**Exercício 5.9.** Uma variável simples pode ser interpretada como um vetor contendo um único elemento e, portanto, pode ser passada por referência se usarmos a notação de vetor<sup>9</sup>. Com base nessa idéia, codifique a rotina *minimax(t,x,y)*, que devolve através dos parâmetros *x* e *y*, respectivamente, a mínima e a máxima entre as temperaturas armazenadas no vetor *t*.

**Exercício 5.10.** Codifique a função *histograma(t)*, que exibe um histograma da variação da temperatura durante a semana. Por exemplo, se as temperaturas em *t* forem 19, 21, 25, 22, 20, 17 e 15°C, a função deverá exibir:

```
D: ██████████  
S: ██████████  
T: ████████████████████  
Q: ████████████████████  
Q: ████████████████████  
S: ████████████████████  
S: ████████████████████
```

Suponha que as temperaturas em *t* sejam todas positivas e que nenhuma seja maior que 50°C. [Dica: crie uma função para exibir uma linha.]

---

<sup>9</sup> Mais tarde veremos como isso pode ser feito com a notação de ponteiros, que é mais adequada.

**Exercício 5.11.** Usando as funções desenvolvidas nos exercícios anteriores, altere o programa apresentado de modo que sejam exibidos a temperatura média, a mínima, a máxima e também o histograma de temperaturas.

**Exercício 5.12.** Altere a rotina *histograma()* de modo que as linhas do gráfico sejam verticais. Além disso, faça com que as linhas que representam temperaturas iguais à média apareçam na cor verde, aquelas correspondentes a temperaturas abaixo da média em azul e aquelas acima da média, em vermelho. [Dica: use as funções *gotoxy()* e *textcolor()*, definidas em *conio.h*]

## 5.2. STRINGS

A *string* é talvez uma das mais importantes formas de armazenamento de dados na maioria das linguagens de programação. Em C, entretanto, ela não é um tipo de dados básico como é, por exemplo, em Pascal.

Em C, uma *string* é uma série de caracteres terminada com um caracter nulo<sup>10</sup>, representado por `'\0'`. Como o vetor é um tipo de dados capaz de armazenar uma série de elementos do mesmo tipo e a *string* é uma série de caracteres, é bastante natural que ela possa ser representada por um vetor de caracteres. Essa representação possibilita que os caracteres que formam a *string* sejam acessados individualmente, o que proporciona grande flexibilidade na sua manipulação.

Na forma de uma constante, a *string* aparece como uma série de caracteres delimitada por aspas; como por exemplo, `"verde e amarelo"`. Internamente, essa *string* é armazenada conforme ilustrado na figura a seguir.

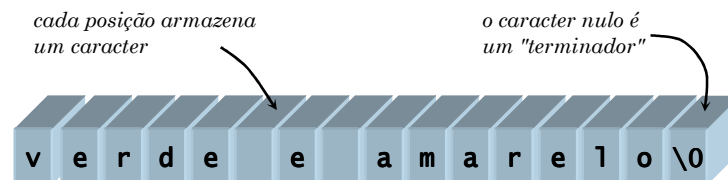


Figura 5.2 - Armazenamento de uma *string*

☛ Devido à necessidade do `'\0'`, os vetores que armazenam strings devem ter sempre uma posição a mais do que o número de caracteres a serem armazenados.

<sup>10</sup> O caracter nulo `'\0'` é o primeiro da tabela ASCII e tem código igual a zero. Cuidado para não confundi-lo com o caracter `'0'`, que tem código ASCII 48.

Quando a *string* é uma constante, o espaço adicional para o caracter '\0' é alocado automaticamente pelo compilador.

**Exemplo 5.12.** Inclusão automática do '\0' em *strings* constantes.

```
#include <stdio.h>
void main(void) {
    printf("\nEspaço alocado = %d bytes", sizeof("verde e amarelo") );
}
```

A saída produzida por esse programa, Espaço alocado = 16 bytes, comprova que o '\0' é realmente incluído automaticamente pelo compilador; pois a *string* "verde e amarelo" possui apenas 15 caracteres. ☒



Numa *string* constante, o '\0' é incluído automaticamente pelo compilador.

No caso de *strings* variáveis, é responsabilidade do programador reservar esse espaço adicional. Lembre-se de que o compilador C não faz consistência da indexação de vetores e, portanto, o dimensionamento inadequado de *strings* não é detectado pelo compilador<sup>11</sup>.

**Exemplo 5.13.** Leitura de *string* via teclado.

```
#include <stdio.h>
void main(void) {
    char n[21];
    printf("Qual o seu nome? ");
    gets(n);
    printf("Olá, %s!", n);
}
```

A chamada `gets(n)` lê uma *string* e a armazena no vetor *n*. O <enter>, digitado para finalizar a entrada, é automaticamente substituído por '\0'. ☒

### 5.2.1. INICIALIZAÇÃO DE STRINGS

Como qualquer outro vetor, *strings* também podem ser inicializadas quando são declaradas. Nessa inicialização, podemos usar a sintaxe padrão, em que os caracteres são fornecidos entre chaves e separados por vírgulas, ou então podemos usar a sintaxe própria para *strings*, na qual os caracteres são fornecidos entre aspas. A vantagem dessa segunda sintaxe é que, além de ser mais compacta, o caracter nulo é incluído automaticamente. Na sintaxe convencional, o '\0' deve ser fornecido explicitamente.

---

<sup>11</sup> O que não significa que o programa irá funcionar corretamente.

**Exemplo 5.14.** Problema com a inicialização padrão de *strings*.

```
#include <stdio.h>
void main(void) {
    char x[] = "um";
    char y[] = {'d','o','i','s'};
    printf("%s %s", x, y);
}
```

A saída da *string* *x* certamente termina após a letra *m* ter sido exibida, pois o '\0' é encontrado. Quanto à *string* *y*, entretanto, não há como saber quando a saída terminará pois, como não foi colocado o terminador, o compilador irá exibir todos os caracteres armazenados após o *s*, até que um '\0' seja encontrado. A saída do programa será algo do tipo: *um doisΩë5ßpŵ©...* ☑

### 5.2.2. MANIPULAÇÃO DE STRINGS

Como a *string* não é um tipo de dados básico da linguagem C, operações simples como atribuição e comparação não podem ser feitas diretamente com os operadores disponíveis.

**Exemplo 5.15.** Problema com o uso de operadores relacionais com *strings*.

```
#include <stdio.h>
void main(void) {
    char x[] = "um";
    char y[] = "um";
    printf("%s == %s resulta em %s", x, y, x==y ? "verdade" : "falso");
}
```

Apesar de *x* e *y* terem o mesmo valor e, portanto, serem iguais, a saída será *um == um resulta em falso*. Isso acontece porque na expressão *x==y* não estamos comparando o conteúdo dos vetores *x* e *y*, mas sim seus endereços que, obviamente, devem ser diferentes. ☑

Qualquer operação com uma *string* exige o processamento individual dos elementos do vetor que a representa. Assim, para verificar se uma *string* é igual a outra, é preciso comparar seus caracteres correspondentes, um a um.

**Exemplo 5.16.** Comparação entre *strings*.

```
int strcmp(char s[], char t[]) {
    int i=0;
    while( s[i]==t[i] && s[i]!='\0' ) i++;
    return s[i]-t[i];
}
```

O laço *while* pode parar somente em duas situações: a primeira é quando o caracter  $s[i]$  difere do seu correspondente  $t[i]$ ; a segunda, é quando eles não diferem, mas são ambos nulos. Nos dois casos, a função *strcmp()* devolve o valor  $s[i]-t[i]$ . Então, se as duas *strings* forem iguais, a resposta da função será o valor 0; caso contrário, a resposta será um valor positivo ou negativo. Se for positivo, então  $s[i]>t[i]$  e, portanto, a *string*  $s$  é maior<sup>12</sup> que a *string*  $t$ . Analogamente, se a resposta for negativa, a *string*  $s$  é menor que a  $t$ . ☒

De modo geral, para um operador relacional  $\diamond$ , a expressão *strcmp*( $x,y$ )  $\diamond$  0 equivale à expressão  $x \diamond y$ . Por exemplo, em vez de "Ana"  $\geq$  "Bia", devemos escrever *strcmp*("Ana","Bia")  $\geq$  0.

**Exemplo 5.17.** Usando a função *strcmp()*<sup>13</sup>.

```
#include <stdio.h>

void main(void) {
    char x[] = "um";
    char y[] = "um";
    char z[] = "dois";

    printf("\n %s = %s  $\equiv$  %s", x, y, strcmp(x,y)==0 ? "V" : "F" );
    printf("\n %s  $\neq$  %s  $\equiv$  %s", x, y, strcmp(x,y)!=0 ? "V" : "F" );
    printf("\n %s < %s  $\equiv$  %s", x, z, strcmp(x,z)< 0 ? "V" : "F" );
    printf("\n %s > %s  $\equiv$  %s", x, z, strcmp(x,z)> 0 ? "V" : "F" );
    printf("\n %s  $\leq$  %s  $\equiv$  %s", z, y, strcmp(z,y)<=0 ? "V" : "F" );
    printf("\n %s  $\geq$  %s  $\equiv$  %s", z, z, strcmp(z,z)>=0 ? "V" : "F" );
}
```

A saída do programa será:

```
um = um  $\equiv$  V
um  $\neq$  um  $\equiv$  F
um < dois  $\equiv$  F
um > dois  $\equiv$  V
dois  $\leq$  um  $\equiv$  V
dois  $\geq$  dois  $\equiv$  V
```

☒

**Exercício 5.13.** Codifique a função *strcpy*( $s,t$ ), que copia o conteúdo da *string*  $t$  para a *string*  $s$ . Essa função é útil quando precisamos realizar atribuição entre *strings*; por exemplo, para atribuir a constante "teste" a uma *string*  $x$ , basta escrever *strcpy*( $x$ , "teste").

<sup>12</sup> Pois o primeiro caracter de  $s$  que difere de seu correspondente em  $t$  tem código ASCII maior e, portanto, aparece depois na ordem lexicográfica.

<sup>13</sup> Para facilitar a manipulação de *strings*, C oferece uma série de funções, definidas em *string.h*, entre as quais estão *strcmp()*, *strcpy()*, *strlen()*, *strupr()* e *strcat()*.



**Exercício 5.14.** Codifique a função *strlen(s)*, que devolve o número de caracteres armazenados na *string* *s*. Lembre-se de que o terminador '\0' não faz parte da *string* e, portanto, não deve ser contado.

**Exercício 5.15.** Codifique a função *strupr(s)*, que converte a *string* *s* em maiúscula. Por exemplo, se *x* armazena "Teste", após a chamada *strupr(x)*, *x* estará armazenando "TESTE".

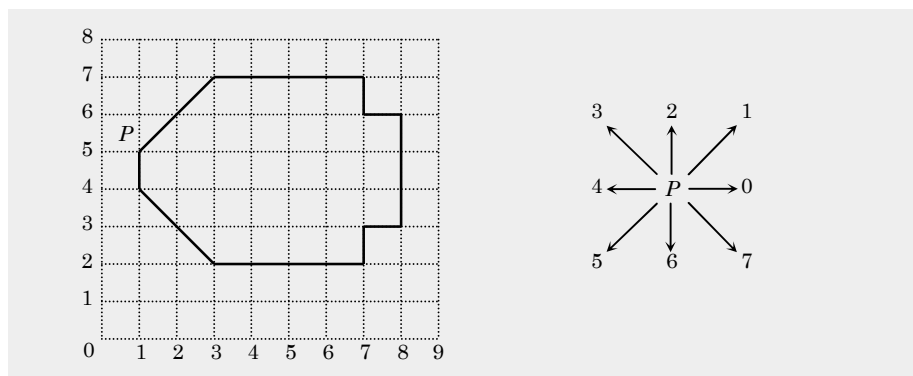
**Exercício 5.16.** Codifique a função *strcat(s,t)*, que concatena a *string* *t* ao final da *string* *s*. Por exemplo, se *x* armazena "facil" e *y* armazena "idade", após a chamada *strcat(x,y)*, *x* estará armazenando "facilidade".

**Exercício 5.17.** Codifique a função *strpos(s,c)*, que devolve a posição da primeira ocorrência do caracter *c* na *string* *s*; ou -1, caso ele não ocorra em *s*.

**Exercício 5.18.** Codifique a função *strdel(s,p)*, que remove o caracter existente na posição *p* da *string* *s*. Se a posição *p* não existe em *s*, nada é feito.

**Exercício 5.19.** Codifique a função *strins(s,c,p)*, que insere o caracter *s* na posição *p* da *string* *s*. Se a posição *p* não existe em *s*, o caracter deve ser inserido no final da *string*.

**Exercício 5.20.** As cadeias de *Freeman* são usadas para representar objetos a partir da codificação de seus contornos. Para montar a cadeia de *Freeman* para um determinado objeto, partimos do princípio de que ele pode ser enquadrado num plano reticulado e que, de um ponto *P* qualquer desse plano, podemos nos mover para oito posições distintas.



Escolhido um ponto *P* como origem, a cadeia é montada seguindo-se o contorno do objeto no sentido horário. Por exemplo, para o objeto representado na figura acima, a codificação de *Freeman* é: (1,5) 1100006066646444332.

- sabendo-se que os movimentos horizontais e verticais contribuem com 1 unidade e que os movimentos diagonais contribuem com 1.42 unidades de comprimento, codifique uma função que receba uma cadeia de *Freeman* e devolve o perímetro da figura representada pela cadeia.
- encontre uma forma de calcular a área de uma figura a partir da sua cadeia de *Freeman* e codifique uma função que implemente essa idéia.
- codifique uma função que desenha uma figura no vídeo a partir da sua cadeia de *Freeman*. [Dica: use os caracteres |, /, - e \.]

### 5.3. MATRIZES

Uma *matriz* é uma coleção homogênea bidimensional, cujos elementos são distribuídos em linhas e colunas. Se  $A$  é uma matriz  $m \times n$ , então suas linhas são indexadas de 0 a  $m-1$  e suas colunas de 0 a  $n-1$ . Para acessar um particular elemento de  $A$ , escrevemos  $A[i][j]$ , sendo  $i$  o número da linha e  $j$  o número da coluna que o elemento ocupa.

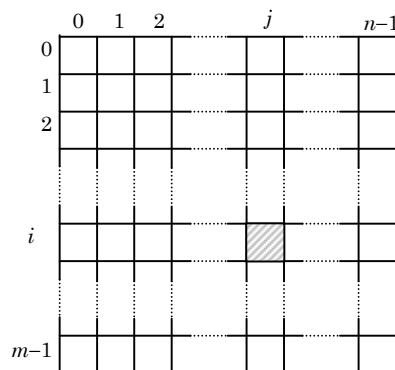


Figura 5.3 – Uma matriz bidimensional

☛ *Tecnicamente, matrizes não são suportadas diretamente em C e, para criar uma matriz, devemos declarar um vetor cujos elementos são vetores.*

**Exemplo 5.18.** Uma matriz  $3 \times 4$  de números inteiros.

```
int A[3][4];
```

Essa declaração cria um vetor  $A$  cujos elementos  $A[0]$ ,  $A[1]$  e  $A[3]$  são vetores contendo cada um deles 4 elementos do tipo *int*. ☑

Para processar uma matriz, usamos laços *for* encaixados: um para variar o índice das linhas e outro para o índice das colunas.

**Exemplo 5.19.** Gerando coordenadas para os elementos de uma matriz.

```
#include <stdio.h>

void main(void) {
    int i, j;
    for(i=0; i<3; i++) {
        putchar('\n');
        for(j=0; j<4; j++)
            printf("[%d][%d] ", i, j);
    }
}
```

A saída produzida pelo programa é:

```
[0][0] [0][1] [0][2] [0][3]
[1][0] [1][1] [1][2] [1][3]
[2][0] [2][1] [2][2] [2][3]
```

Ou seja, as coordenadas de cada um dos elementos de uma matriz 3×4. ☒

Um dos usos mais comuns de matrizes em C é quando precisamos armazenar uma coleção de *strings*. Como a *string* já é um vetor, devemos criar um vetor cujos elementos também sejam vetores.

**Exemplo 5.20.** Uma matriz de caracteres usada como um vetor de *strings*.

```
#include <stdio.h>

void main(void) {
    char n[5][11];
    int i;
    for(i=0; i<5; i++) {
        printf("%dª nome: ", i+1);
        gets(n[i]);
    }
    ...
}
```

Na chamada à *gets()*, cada linha da matriz é tratada como uma *string*. ☒

**Exercício 5.21.** Codifique um programa para ler uma matriz quadrada de ordem *n* e exibir apenas os elementos da diagonal principal.

**Exercício 5.22.** Complemente o programa do exemplo 5.20 de modo que os nomes lidos sejam exibidos em minúsculas, com inicial maiúscula.

### 5.3.1. INICIALIZAÇÃO DE MATRIZES

Se lembrarmos que uma matriz nada mais é que um vetor cujos elementos são vetores, a sintaxe para a sua inicialização não tem grandes novidades.

**Exemplo 5.21.** Inicializando e exibindo um labirinto.

```
#include <stdio.h>

void main(void) {
    static int lab[10][10] = {
        {1,1,1,1,1,1,1,1,1,1},
        {0,0,1,0,0,0,1,0,1,1},
        {1,0,1,0,1,0,1,0,1,1},
        {1,0,1,0,1,0,0,0,0,1},
        {1,0,1,1,1,0,1,1,0,1},
        {1,0,0,0,0,0,1,0,1,1},
        {1,0,1,0,0,1,1,0,1,1},
        {1,0,0,1,0,1,0,0,0,1},
        {1,0,1,1,0,0,0,1,0,0},
        {1,1,1,1,1,1,1,1,1,1}
    };
    int i, j;
    for(i=0; i<10; i++) {
        putchar('\n');
        for(j=0; j<10; j++)
            putchar(lab[i][j] ? 219 : 32);
    }
}
```

A saída do programa é:



Note que cada um dos 10 elementos da matriz é um vetor de 10 inteiros. ☒

Embora as duas dimensões da matriz *lab* tenham sido especificadas, o compilador C permite que a primeira dimensão de uma matriz seja omitida<sup>14</sup>. E, nesse caso, ele determina o seu valor contando os elementos fornecidos na lista de valores iniciais.

---

<sup>14</sup> As demais dimensões, entretanto, são obrigatórias e o motivo é esclarecido no capítulo 3.

**Exemplo 5.22.** Inicializando e exibindo um menu de opções.

```
#include <stdio.h>
void main(void) {
    static char menu[][7] = { "abrir", "editar", "salvar", "sair"};
    int i;
    for(i=0; i<4; i++)
        puts(menu[i]);
}
```

Note que a matriz *menu* é usada no programa como um vetor de *strings*. ☑

	0	1	2	3	4	5	6
0	a	b	r	i	r	\0	
1	e	d	i	t	a	r	\0
2	s	a	l	v	a	r	\0
3	s	a	i	r	\0		

Figura 5.4 – A matriz *menu* do exemplo 5.22

**Exercício 5.23.** Crie um programa que inicializa e exibe uma matriz representando um tabuleiro para o "jogo da velha", conforme a seguir:

```

  | o | x
---+---+---
o | x | o
---+---+---
x |   |

```

### 5.3.2. PASSANDO MATRIZES A FUNÇÕES

Assim como no caso de vetores, o nome de uma matriz representa o endereço que ela ocupa na memória. Então, quando passamos uma matriz como argumento, a função tem acesso direto aos valores armazenados nessa matriz e, portanto, não precisa fazer uma cópia dela.

**Exemplo 5.23.** Preenchendo um labirinto 10×10 com valores lidos do teclado.

```
void preenche(int L[10][10]) {
    int i, j;
    for(i=0; i<10; i++)
        for(j=0; j<10; j++) {
            printf("[%d][%d]= ");
            scanf("%d", &L[i][j]);
        }
}
```

}



**Exercício 5.24.** Usando as funções *randomize()* e *random()*, ambas definidas em *stdlib.h*, crie uma função para preencher, aleatoriamente, uma matriz 10×10 representando um campo minado 8×8. Os limites da matriz devem ser preenchidos com uns e o interior, com 0's nas posições livres e 9's nas posições contendo bombas. Utilize um parâmetro adicional na sua função para especificar a "facilidade" do campo minado, de tal modo que quanto maior for a facilidade, menor seja número de bombas no campo.

**Exercício 5.25.** Crie uma função que recebe um campo minado e marca, para cada posição livre, o número de posições adjacentes que contêm bombas, conforme exemplificado a seguir:

```
*1001*3*
331012*2
**100232
23212*2*
*11*3221
22323*21
1*4*223*
2**21011
```

**Exercício 5.26.** Codifique uma função que recebe como argumentos um campo minado marcado, vide exercício anterior, e as coordenadas de uma posição dele. A função deve marcar a posição indicada como "aberta" e exibir o campo minado no vídeo, conforme ilustração a seguir, mostrando apenas as posições já "abertas". Além disso, a função deve devolver 1 se a posição estiver livre e 0 se tiver uma bomba.

```
12345678
1 ♦♦♦♦♦♦♦♦
2 ♦♦♦01♦♦♦
3 ♦♦♦1002♦♦
4 ♦♦♦12♦♦♦
5 ♦11♦3221
6 ♦♦♦♦♦♦♦♦
7 ♦♦4♦♦♦♦♦
8 ♦♦♦♦♦♦♦♦
```

**Exercício 5.27.** Usando as funções desenvolvidas nos exercícios anteriores, codifique um programa para jogar campo minado. O programa deve solicitar uma posição ao usuário e mostrar o campo, repetidamente, até que a posição (0,0) seja fornecida ou que o usuário jogue numa posição minada.

## 5.4. MÉTODOS DE BUSCA

A *busca* é o processo em que se determina se um particular elemento  $x$  é membro de uma determinada lista  $\mathcal{L}$ . Dizemos que a busca tem *sucesso* se  $x \in \mathcal{L}$  e que *fracassa* em caso contrário.

### 5.4.1. BUSCA LINEAR

A forma mais simples de se consultar uma lista em busca de um item particular é, a partir do seu início, ir examinando cada um de seus itens até que o item desejado seja encontrado ou então que seu final seja atingido.

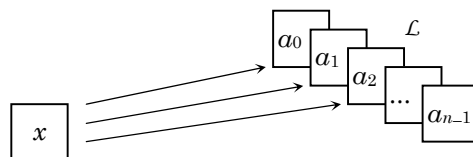


Figura 5.5 – O método de busca linear

Como os itens da lista são examinados linearmente, em seqüência, esse método é denominado *busca linear* ou *busca seqüencial*. Para exemplificar seu funcionamento, vamos implementar uma função que determina se um certo número  $x$  consta de uma lista de números inteiros  $\mathcal{L} = \langle a_0, a_1, a_2, \dots, a_{n-1} \rangle$ .

$$pertence(x, \mathcal{L}) = \begin{cases} \text{verdade se } x \in \mathcal{L} \\ \text{falso se } x \notin \mathcal{L} \end{cases}$$

**Exemplo 5.24.** Busca linear num vetor de inteiros.

```
int pertence(int x, int L[], int n) {
    int i;
    for(i=0; i<n; i++)
        if( x == L[i] )
            return 1;
    return 0;
}
```

Nesse código, a função *pertence()* recebe um parâmetro adicional  $n$  que serve para indicar o tamanho da lista. ☑

### **Análise da busca linear**

A vantagem da busca linear é que ela sempre funciona, independentemente da lista estar ou não ordenada. A desvantagem é que ela é geralmente muito lenta; pois, para encontrar um determinado item  $x$ , a busca linear precisa examinar todos os itens que precedem  $x$  na lista. Para se ter uma idéia, se cada item da lista fosse procurado exatamente uma vez, o total de comparações realizadas seria  $1+2+3+\dots+n=\frac{1}{2}(n^2+n)$ . Isso significa que, para encontrar um item, a busca linear realiza em média  $\frac{1}{2}(n+1)$  comparações, ou seja, examina aproximadamente a metade dos elementos armazenados no vetor.

No pior caso, quando o item procurado não consta da lista, a busca linear precisa examinar todos os elementos armazenados no vetor para chegar a essa conclusão. Dizemos então que o tempo gasto por esse algoritmo é da ordem de  $n$ , isto é,  $O(n)$ . Isso significa, por exemplo, que se o tamanho da lista é dobrado a busca linear fica aproximadamente duas vezes mais lenta.

**Exercício 5.28.** Codifique a função definida a seguir:

$$\text{posição}(x,L) = \begin{cases} i & \text{se } \exists i \text{ tal que } x=L[i] \\ -1 & \text{caso contrário} \end{cases}$$

**Exercício 5.29.** Em cada iteração do laço *for*, na função *pertence()*, são realizadas duas comparações:  $i < n$  e  $x == L[i]$ . Podemos reduzir esse número pela metade se empregarmos uma técnica, denominada *sentinela*, que consiste em deixar uma posição livre no final do vetor e, antes de iniciar a busca, armazenar nela o elemento procurado. Então, podemos percorrer o vetor enquanto tivermos  $x \neq L[i]$ . Quando o laço parar, se tivermos  $i < n$  significa que o elemento foi encontrado; caso contrário, o laço parou por ter encontrado a *sentinela* que foi posicionada no final do vetor e, portanto, ocorreu fracasso. Implemente uma função de busca linear usando essa técnica.

**Exercício 5.30.** Dados a lista de convidados de uma festa<sup>15</sup> e o nome de uma pessoa, determinar se essa pessoa é ou não convidada da festa. Codifique um programa completo para resolver esse problema. Crie um procedimento para fazer a entrada da lista de convidados e adapte a função *pertence()*, definida anteriormente, para verificar se o nome consta ou não da lista.

---

<sup>15</sup> Essa lista deve ser um vetor de strings e, como uma string também é um vetor, precisamos usar uma matriz bidimensional, da forma  $L[n][m]$ , onde  $n$  é o número de strings e  $m$  é o comprimento de cada uma delas. Para acessar uma string individual, escreve-se apenas  $L[i]$ .



### 5.4.2. BUSCA BINÁRIA

Se não sabemos nada a respeito da ordem em que os itens aparecem na lista, o melhor que podemos fazer é uma busca linear. Entretanto, se os itens aparecem ordenados<sup>16</sup>, podemos usar um método de busca muito mais eficiente. Esse método é semelhante àquele que usamos quando procuramos uma palavra num dicionário: primeiro abrimos o dicionário numa página aproximadamente no meio; se tivermos sorte de encontrar a palavra nessa página, ótimo; senão, verificamos se a palavra procurada ocorre antes ou depois da página em que abrimos e então continuamos, mais ou menos do mesmo jeito, procurando a palavra na primeira ou na segunda metade do dicionário...

Como a cada comparação realizada o espaço de busca reduz-se aproximadamente à metade, esse método é denominado *busca binária*.

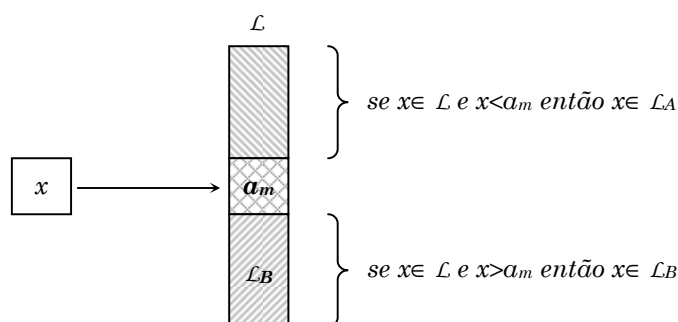


Figura 5.6 – O método de busca binária

Suponha que a lista<sup>17</sup>  $L = L_A \circ \langle a_m \rangle \circ L_B$  esteja armazenada num vetor e que  $a_m$  esteja aproximadamente no meio dele. Então temos três possibilidades:

- $x = a_m$ : nesse caso, o problema está resolvido;
- $x < a_m$ : então  $x$  deverá ser procurado na primeira metade; e
- $x > a_m$ : então  $x$  deverá ser procurado na segunda metade.

Caso a busca tenha que continuar, podemos proceder exatamente da mesma maneira: verificamos o item existente no meio da metade escolhida e se ele ainda não for aquele que procuramos, continuamos procurando no meio do quarto escolhido, depois no meio do oitavo e assim por diante até que o item procurado seja encontrado ou que não haja mais itens a examinar.

<sup>16</sup> Quando nada for dito em contrário, 'ordenado' quer dizer 'ordenado de forma ascendente'

<sup>17</sup> O operador  $\circ$  indica concatenação de seqüências.

**Exemplo 5.25.** Vamos simular o funcionamento do algoritmo de busca binária para determinar se o item  $x=63$  pertence à lista  $L=\langle 14, 27, 39, 46, 55, 63, 71, 80, 92 \rangle$ . Para indicar o intervalo em que será feita a busca, usaremos dois índices,  $i$  e  $f$ , e para indicar a posição central desse intervalo, o índice  $m$ .

Passo	$x$	$i$	$f$	$m$	$L[0]$	$L[1]$	$L[2]$	$L[3]$	$L[4]$	$L[5]$	$L[6]$	$L[7]$	$L[8]$
1ª	63	0	8	4	14	27	39	46	<b>55</b>	63	71	80	92
2ª	63	5	8	6						63	<b>71</b>	80	92
3ª	63	5	5	5						<b>63</b>			

No primeiro passo o intervalo de busca compreende toda a lista, desde a posição 0 até a posição 8, e o item central ocupa a posição 4. Como  $x > L[4]$ , a busca deve continuar na segunda metade da lista. Para indicar isso, atualizamos o índice  $i$  com o valor 5 e repetimos o procedimento. No segundo passo, o intervalo de busca foi reduzido aos itens que ocupam as posições de 5 a 8 e o meio<sup>18</sup> da lista é a posição 6. Agora  $x < L[6]$  e, portanto, a busca deve continuar na primeira metade; então o índice  $f$  é atualizado com o valor 5. No terceiro passo, resta um único item no intervalo de busca e o meio da lista é a posição 5. Finalmente, temos  $x = L[5]$  e a busca termina com sucesso.  $\checkmark$

Como em cada passo o intervalo de busca reduz-se aproximadamente à metade, é evidente que o processo de busca binária sempre termina, mesmo que o item procurado não conste da lista. Nesse caso, porém, o processo somente termina quando não há mais itens a examinar, ou seja, quando o intervalo de busca fica vazio. Como o início e o final do intervalo são representados pelos índices  $i$  e  $f$ , o intervalo estará vazio se e somente se tivermos  $i > f$ .

**Exemplo 5.26.** Seja  $x=40$  e  $L=\langle 14, 27, 39, 46, 55, 63, 71, 80, 92 \rangle$ . A tabela a seguir mostra o que acontece quando o item procurado não consta da lista.

Passo	$x$	$i$	$f$	$m$	$L[0]$	$L[1]$	$L[2]$	$L[3]$	$L[4]$	$L[5]$	$L[6]$	$L[7]$	$L[8]$
1ª	40	0	8	4	14	27	39	46	<b>55</b>	63	71	80	92
2ª	40	0	3	1	14	<b>27</b>	39	46					
3ª	40	2	3	2			<b>39</b>	46					
4ª	40	3	3	3				46					
5ª	40	<b>3</b>	<b>2</b>	?									

Note que, durante a busca, o valor que indica o início do intervalo fica maior que aquele que indica o seu final (veja 5ª passo na tabela acima). Isso significa que não há mais itens a considerar e que, portanto, o item procurado não consta da lista. Nesse caso, a busca deve terminar com fracasso.  $\checkmark$

<sup>18</sup> O meio da lista é dado pelo quociente inteiro da divisão  $(i+f)/2$ , isto é, a média é truncada.

**Exemplo 5.27.** Busca binária num vetor de inteiros.

```
int pertence(int x, int L[], int n) {
    int i, f, m;
    i = 0;
    f = n-1;
    while( i<=f ) {
        m = (i+f)/2;
        if( x == L[m] ) return 1;
        if( x < L[m] ) f = m-1;
        else i = m+1;
    }
    return 0;
}
```

Note que, em C, o operador / fornece resultado inteiro quando seus operandos são ambos inteiros. ☒

### Análise da busca binária

Ao contrário da busca linear, a busca binária somente funciona corretamente se o vetor estiver ordenado. Isso pode ser uma desvantagem. Entretanto, à medida em que o tamanho do vetor aumenta, o número de comparações feitas pelo algoritmo de busca binária tende a ser muito menor que aquele feito pela busca linear. Então, se o vetor é muito grande, e a busca é uma operação muito requisitada, esse aumento de eficiência pode compensar o fato de termos que ordenar o vetor antes de usar a pesquisa binária.

**Exemplo 5.28.** Quantas comparações o algoritmo de busca binária faz, no máximo, para encontrar um item numa lista com 5000 itens?

Seja  $n$  o número de itens na lista. Se  $n$  é ímpar, o item do meio divide a lista em duas partes iguais de tamanho  $(n-1)/2$ . Se  $n$  é par, a lista é dividida em uma parte com  $n/2-1$  itens e outra com  $n/2$  itens<sup>19</sup>. Sendo assim, à medida em que o algoritmo executa, o número de itens vai reduzindo do seguinte modo:

5000  $\Rightarrow$  2500  $\Rightarrow$  1250  $\Rightarrow$  625  $\Rightarrow$  312  $\Rightarrow$  156  $\Rightarrow$  78  $\Rightarrow$  39  $\Rightarrow$  19  $\Rightarrow$  9  $\Rightarrow$  4  $\Rightarrow$  2  $\Rightarrow$  1  $\Rightarrow$  0

Na sequência acima, cada redução implica numa comparação. Logo, são realizadas no máximo 13 comparações. ☒

Seja  $C(n)$  o número máximo de comparações realizadas pelo algoritmo de busca binária num vetor de  $n$  elementos. Claramente,  $C(1) = 1$ . Suponha  $n > 1$ . Como a cada comparação realizada o número de elementos cai para a metade, temos que  $C(n) = 1 + C(n/2)$ .

<sup>19</sup> Como queremos o número máximo de comparações, vamos considerar sempre a parte maior.

Iterando essa recorrência temos:

$$\begin{aligned}
 C(n) &= 1 + C(n/2) \\
 &= 1 + 1 + C(n/4) \\
 &= 1 + 1 + 1 + C(n/8) \\
 &\dots \\
 &= k + C(n/2^k)
 \end{aligned}$$

Supondo  $n=2^k$ , para algum inteiro  $k$ , temos que  $C(n/2^k) = 1$  e  $k = \lg n$  e, portanto,  $C(n) \approx \lg n + 1$ . Mais precisamente, temos que<sup>20</sup>  $C(n) = \lfloor \lg n \rfloor + 1$ . Assim, podemos dizer que o algoritmo de busca binária tem complexidade  $O(\lg n)$ .

A tabela a seguir mostra a relação entre o crescimento do tamanho do vetor  $n$  e do número de comparações realizadas pelos algoritmos de busca linear  $C_L(n)$  e binária  $C_B(n)$ . É espantosa a eficiência da busca binária.

$n$	$C_L(n)$	$C_B(n)$
1	1	1
2	2	2
4	4	3
8	8	4
16	16	5
32	32	6
64	64	7
128	128	8
256	256	9
1024	1024	10
2048	2048	11
4096	4096	12
8192	8192	13
16384	16384	14
32768	32768	15
65536	65536	16
131072	131072	17
262144	262144	18
524288	524288	19
1048576	1048576	20

**Figura 5.7** – Comparações na busca linear versus na binária

<sup>20</sup> Essa fórmula, em que a função  $\lfloor x \rfloor$  denota o maior inteiro menor ou igual a  $x$ , pode ser provada por indução finita.

**Exercício 5.31.** Simule o funcionamento do algoritmo de busca binária para determinar se os itens 33, 50, 77, 90 e 99 constam da lista  $L = \langle 10, 16, 27, 31, 33, 37, 41, 49, 53, 57, 68, 69, 72, 77, 84, 89, 95, 99 \rangle$ .

**Exercício 5.32.** Faça as alterações necessárias para que o algoritmo de busca binária funcione com vetores ordenados de forma decrescente.

## 5.5. MÉTODOS DE ORDENAÇÃO

Seja  $\mathcal{L} = \langle a_0, a_1, a_2, \dots, a_{n-1} \rangle$  uma lista cujos itens aparecem numa ordem aleatória. A *ordenação* é o processo em que se determina uma permutação,  $\rho(0), \rho(1), \rho(2), \dots, \rho(n-1)$ , dos índices de  $\mathcal{L}$  tal que  $a_{\rho(0)} \leq a_{\rho(1)} \leq a_{\rho(2)} \leq \dots \leq a_{\rho(n-1)}$ .

### 5.5.1. ORDENAÇÃO POR TROCAS

Talvez a estratégia mais simples para ordenar os itens de uma lista seja comparar pares de itens consecutivos e permutá-los, caso estejam fora de ordem. Se a lista for assim processada, sistematicamente, da esquerda para a direita, um item máximo será deslocado para a última posição da lista.

**Exemplo 5.29.** Veja essa estratégia aplicada a uma lista de números:

$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$	$v[6]$	$v[7]$	$v[8]$
71	63	46	80	39	92	55	14	27
63	71	46	80	39	92	55	14	27
63	46	71	80	39	92	55	14	27
63	46	71	80	39	92	55	14	27
63	46	71	39	80	92	55	14	27
63	46	71	39	80	92	55	14	27
63	46	71	39	80	55	92	14	27
63	46	71	39	80	55	14	92	27
63	46	71	39	80	55	14	27	92

À medida em que a lista vai sendo processada, cada número vai sendo deslocado para a direita, até que seja encontrado outro maior. Evidentemente, no final do processo, um valor máximo estará na última posição da lista.  $\square$

Em cada fase desse método, um item de maior valor é deslocado para sua posição definitiva na lista ordenada. Então, se uma lista tem  $n$  itens, após a primeira fase haverá  $n-1$  itens a ordenar. Usando a mesma estratégia, após a segunda fase, termos  $n-2$  itens, depois  $n-3$  e assim sucessivamente até que reste um único item.

Se considerarmos os números maiores como mais pesados e os menores como mais leves, veremos que, durante a ordenação por esse método, os números mais pesados "descem" rapidamente para o "fundo" do vetor, enquanto que os números mais leves "sobem" lentamente para a "superfície". Os números pesados descem como pedras e os leves sobem como bolhas<sup>21</sup> de ar.

Embora seja um dos métodos de ordenação menos eficientes que existem, o método da bolha é geralmente o primeiro algoritmo de ordenação que todo mundo aprende. A sua popularidade deve-se, principalmente, à sua simplicidade e facilidade de codificação. Métodos mais eficientes, em geral, são também mais complicados de se entender e de codificar.

**Exemplo 5.30.** Veja agora a ordenação completa de uma lista pelo método da bolha. A variável  $i$  indica a fase da ordenação e  $j$  indica a posição do par de itens consecutivos que serão comparados e, eventualmente, permutados.

<i>fase</i>	<i>i</i>	<i>j</i>	<i>v</i> [0]	<i>v</i> [1]	<i>v</i> [2]	<i>v</i> [3]	<i>v</i> [4]
1ª	1	0	46	39	55	14	27
		1	39	46	55	14	27
		2	39	46	55	14	27
		3	39	46	14	55	27
			39	46	14	27	55
2ª	2	0	39	46	14	27	55
		1	39	46	14	27	55
		2	39	14	46	27	55
			39	14	27	46	55
3ª	3	0	39	14	27	46	55
		1	14	39	27	46	55
			14	27	39	46	55
4ª	4	0	14	27	39	46	55
			14	27	39	46	55

Note que na última fase a lista já está ordenada mas, mesmo assim, uma última comparação precisa ser feita para que isso seja confirmado. ☑

Observando o exemplo acima, podemos constatar que para ordenar  $n$  itens bastam apenas  $n-1$  fases e que, numa determinada fase  $i$ , são realizadas  $n-i$  comparações.

<sup>21</sup> Daí esse método ser conhecido como método da bolha ou bubble sort.

**Exemplo 5.31.** Ordenação por trocas.

```
void trocas(int v[], int n) {
    int i, j;
    for(i=1; i<n; i++)
        for(j=0; j<n-i; j++)
            if( v[j]>v[j+1] ) {
                int x = v[j];
                v[j] = v[j+1];
                v[j+1] = x;
            }
}
```



### Análise da ordenação por trocas

Analisando a rotina *trocas()*, verificamos que o número total de comparações realizadas é  $(n-1)+(n-2)+(n-3)+\dots+2+1 = \frac{1}{2}(n^2-n)$ , ou seja, a sua complexidade de tempo é  $O(n^2)$ . Como a cada comparação corresponde uma troca em potencial, no pior caso, isto é, quando o vetor estiver em ordem decrescente, serão realizadas no máximo  $\frac{1}{2}(n^2-n)$  trocas.

**Exercício 5.33.** Simule a execução da função *trocas()*, no estilo do exemplo 5.30, para ordenar a lista  $L=\langle 92, 80, 71, 63, 55, 41, 39, 27, 14 \rangle$ .

**Exercício 5.34.** Adapte a função *trocas()* de modo que o vetor seja ordenado de forma decrescente.

### 5.5.2. ORDENAÇÃO POR SELEÇÃO

A estratégia básica desse método é, em cada fase, selecionar um menor item ainda não ordenado e permutá-lo com aquele que ocupa a sua posição na seqüência ordenada. Mais precisamente, isso pode ser descrito assim: para ordenar uma seqüência  $\langle a_i, a_{i+1}, \dots, a_{n-1} \rangle$ , selecione uma valor  $k$  tal que  $a_k = \min\{a_i, a_{i+1}, \dots, a_{n-1}\}$ , permuta os elementos  $a_i$  e  $a_k$  e, se  $i+1 < n-1$ , repita o procedimento para ordenar a subsequência  $\langle a_{i+1}, \dots, a_{n-1} \rangle$ .

**Exemplo 5.32.** Ordenação da seqüência  $\langle 46, 55, 59, 14, 38, 27 \rangle$  usando seleção.

Fase	$i$	$k$	$a_0$	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$
1ª	0	3	46	55	59	14	38	27
2ª	1	5	14	55	59	46	38	27
3ª	2	4	14	27	59	46	38	55
4ª	3	3	14	27	38	46	59	55
5ª	4	5	14	27	38	46	59	55
			14	27	38	46	55	59

Note que, em cada fase, um valor apropriado para  $k$  é escolhido e os itens  $a_i$  e  $a_k$  são permutados. Particularmente na 4ª fase, como os valores de  $i$  e  $k$  são iguais, a permutação não seria necessária. ☒

O que ainda não está muito evidente é como o valor de  $k$  é escolhido em cada fase. Para isso, vamos codificar a função *selmin*( $v, i, n$ ), que devolve o índice de um item mínimo dentro da sequência  $\langle v_i, v_{i+1}, \dots, v_{n-1} \rangle$ . A estratégia adotada supõe que o item  $v_i$  é o mínimo e então o compara com  $v_{i+1}, v_{i+2}, \dots$  até que não haja mais itens ou então que um item menor  $v_k$  seja encontrado. Nesse caso,  $v_k$  passa a ser o mínimo e o processo continua analogamente.

**Exemplo 5.33.** Selecionando um item mínimo numa sequência.

$k$	$j$	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
0	1	46	55	59	14	38	27
0	2	46	55	59	14	38	27
0	3	46	55	59	14	38	27
3	4	46	55	59	14	38	27
3	5	46	55	59	14	38	27
3	–	46	55	59	14	38	27

Inicialmente, o item  $v_0$  é assumido como o mínimo ( $k=0$ ). Então  $v_0$  é comparado aos demais itens da sequência até que  $v_3$  é encontrado. Como  $v_3 < v_0$ , o item  $v_3$  passa a ser o mínimo ( $k=3$ ) e o processo continua analogamente. ☒

**Exemplo 5.34.** Selecionando um item mínimo.

```
int selmin(int v[], int i, int n) {
    int j, k=i;
    for(j=i+1; j<n; j++)
        if( v[k]>v[j] )
            k=j;
    return k;
}
```

☒

**Exemplo 5.35.** Ordenação por seleção.

```
void selecao(int v[], int n) {
    int i, k, x;
    for(i=0; i<n-1; i++) {
        k = selmin(v,i,n);
        x = v[i];
        v[i] = v[k];
        v[k] = x;
    }
}
```

☒



### Análise da ordenação por seleção

Analisando a rotina *selecao()*, constatamos que ela realiza o mesmo número de comparações que a rotina *trocas()*, ou seja,  $(n-1)+(n-2)+\dots+2+1 = \frac{1}{2}(n^2-n)$ . Logo, a sua complexidade de tempo também é  $O(n^2)$ . Entretanto, como a subsequência a ordenar diminui de um item a cada troca feita, temos que o número máximo de trocas<sup>22</sup> na *selecao()* é  $n-1$ ; um número consideravelmente menor do que aquele encontrado para a rotina *trocas()*. Podemos dizer que o número de trocas na *trocas()* é  $O(n^2)$ , enquanto na *selecao()* é  $O(n)$ .

**Exercício 5.35.** Simule a execução da função *selecao()*, conforme no exemplo 5.32, para ordenar a lista  $L=\langle 82, 50, 71, 63, 85, 43, 39, 97, 14 \rangle$ .

**Exercício 5.36.** Codifique a função *selecao()* sem usar a função *selmin()*, ou seja, embutindo a lógica dessa função diretamente no código da *selecao()*.

**Exercício 5.37.** Adapte a função *selecao()* para ordenar um vetor de *strings*.

### 5.5.3. ORDENAÇÃO POR INSERÇÃO

Ao ordenar uma sequência  $L=\langle a_0, a_1, \dots, a_{n-1} \rangle$ , esse método considera uma subsequência ordenada  $L_o=\langle a_0, \dots, a_{i-1} \rangle$  e outra desordenada  $L_d=\langle a_i, \dots, a_{n-1} \rangle$ . Então, em cada fase, um item é removido de  $L_d$  e inserido em sua posição correta dentro de  $L_o$ . À medida em que o processo se desenvolve, a subsequência desordenada vai diminuindo, enquanto a ordenada vai aumentando.

**Exemplo 5.36.** Ordenação de um vetor  $v$  usando inserção.

$i$	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$
1	34	17	68	29	50	47
2	17	34	68	29	50	47
3	17	34	68	29	50	47
4	17	29	34	68	50	47
5	17	29	34	50	68	47
	17	29	34	47	50	68

Inicialmente, a parte ordenada<sup>23</sup> é  $\langle 34 \rangle$  e a desordenada é  $\langle 17, 68, 29, 50, 47 \rangle$ . Então, o item 17 é removido da parte desordenada e a posição liberada fica disponível no final da parte ordenada. Em seguida, o item 17 é comparado ao 34 que, sendo maior, é deslocado para a direita. Como não há mais itens, o 17 é inserido na posição liberada pelo item 34. Ao final dessa fase, a parte ordenada tem um item a mais e a desordenada, um item a menos... ☒

<sup>22</sup> As trocas de índices realizadas em *selmin()* não estão sendo consideradas.

<sup>23</sup> Evidentemente, qualquer sequência contendo um único item está trivialmente ordenada.

**Exemplo 5.37.** Ordenação por inserção.

```
void insercao(int v[], int n) {
    int i, j, x;
    for(i=1; i<n; i++) {
        x = v[i];
        for(j=i-1; j>=0 && x<v[j]; j--)
            v[j+1] = v[j];
        v[j+1] = x;
    }
}
```



### **Análise da ordenação por inserção**

A análise da rotina *insercao()* mostra que o número de comparações realizadas é no máximo  $\frac{1}{2}(n^2-n)$  e que, portanto, sua complexidade de tempo é  $O(n^2)$ . Entretanto, ao contrário do que acontece com os métodos de trocas e de seleção, para as quais esse número é fixo, no método da inserção o número de comparações varia de  $n-1$ , quando a seqüência é crescente, a  $\frac{1}{2}(n^2-n)$ , quando a seqüência é decrescente. Isso quer dizer que, no caso médio, o método da inserção pode ser um pouco mais eficiente que os outros dois.

**Exercício 5.38.** Simule a execução da função *insercao()*, preenchendo uma tabela como aquela apresentada no exemplo 5.36, para ordenar a seqüência  $\langle 82, 50, 71, 63, 85, 43, 39, 97, 14 \rangle$ .

**Exercício 5.39.** Crie uma programa para preencher aleatoriamente três vetores (com a mesma seqüência de números) e então ordenar cada um deles por um método distinto, marcando o tempo que cada um gasta. Faça seu programa tabular os tempos para tamanhos crescentes do vetor. [Dica: use a função *clock()*, definida em *time.h*]

## 6. ESTRUTURAS E UNIÕES

*Juntamente com vetores, as estruturas formam a base a partir da qual é possível implementar diversos outros tipos agregados mais complexos. Esse capítulo introduz o uso de estruturas e mostra como criar tipos de dados polimórficos usando uniões, um tipo especial de estrutura.*

### 6.1. ESTRUTURAS

---

Um *estrutura*<sup>24</sup> é uma coleção arbitrária de variáveis logicamente relacionadas. Como no vetor, essas variáveis compartilham o mesmo nome e ocupam posições consecutivas de memória. As variáveis que fazem parte de uma estrutura são denominadas *membros*<sup>25</sup> e são identificadas por *nomes*. Na figura a seguir, os membros da estrutura *x* são *x.a*, *x.b* e *x.c*.

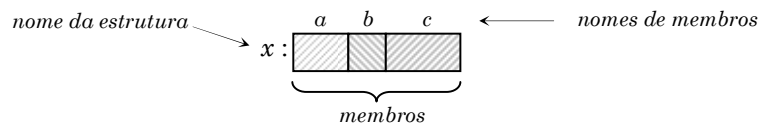


Figura 6.1 – Uma estrutura e seus membros

**Exemplo 6.1.** Criando uma estrutura para armazenar uma data.

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} hoje;
```

Esse fragmento de código declara a variável *hoje* como uma *struct* cujos membros são *dia*, *mes* e *ano*, todos do tipo *int*. Para atribuir valores a eles, podemos escrever:

```
hoje.dia = 19;  
hoje.mes = 7;  
hoje.ano = 2000;
```



---

<sup>24</sup> Estruturas também são conhecidas como registros em outras linguagens.

<sup>25</sup> Membros são também denominados campos.

Embora seja possível criar uma estrutura conforme visto no último exemplo, essa forma não é aconselhável. O exemplo cria um tipo de estrutura anônimo, que não pode ser referenciado em outras partes do programa. Isso quer dizer, por exemplo, que não é possível declarar outras variáveis do mesmo tipo da variável *hoje*. Para resolver esse problema, devemos usar um *rótulo*.

**Exemplo 6.2.** Criando um tipo de estrutura rotulada.

```
typedef struct data {  
    int dia;  
    int mes;  
    int ano;  
};
```

Esse fragmento de código cria um tipo de estrutura, cujo rótulo é *data*, através do qual podemos declarar variáveis da seguinte maneira:

```
struct data hoje;  
struct data ontem, amanha;
```



*O rótulo de uma estrutura, usado isoladamente, não é reconhecido pelo compilador com sendo um tipo de dados. Assim, o uso da palavra *struct* é obrigatório.*

Embora essa última forma seja melhor que a anterior, ela ainda é desconfortável; pois temos que usar a palavra *struct* precedendo o rótulo. Para evitar isso, temos uma terceira possibilidade que faz uso do comando *typedef*.

**Exemplo 6.3.** Criando um tipo de estrutura rotulada e nomeada.

```
typedef struct data {  
    int dia;  
    int mes;  
    int ano;  
} DATA;
```

Agora o tipo de estrutura, cujo rótulo é *data*, recebe o nome *DATA* e não precisamos mais usar a palavra *struct*:

```
DATA hoje;  
DATA ontem, amanha;
```



Finalmente, se o tipo de estrutura não é recursivo<sup>26</sup>, podemos omitir seu rótulo e declarar apenas seu nome.

---

<sup>26</sup> Uma estrutura é recursiva se usa seu próprio tipo para declarar um de seus campos.

**Exemplo 6.4.** Criando de um tipo de estrutura apenas nomeada.

```
typedef struct {  
    int dia;  
    int mes;  
    int ano;  
} DATA;
```



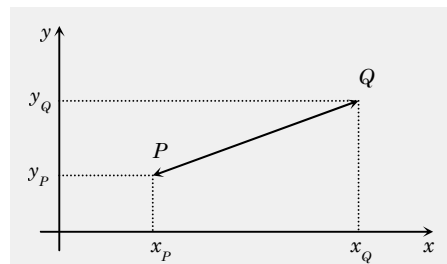
**Exercício 6.1.** Codifique um programa para criar uma variável de tipo anônimo, capaz de armazenar o título, o autor, a editora e o ano de publicação de um livro; atribuir valores aos seus campos e exibi-la no vídeo.

**Exercício 6.2.** Defina um tipo de estrutura rotulada para representar números complexos da forma  $a+b\cdot i$ , sendo  $a$  a parte real e  $b$  a imaginária. Em seguida, crie uma função para calcular a soma de dois números complexos e codifique um programa para testar seu funcionamento. Use atribuição para inicializar os campos membros das variáveis.

**Exercício 6.3.** Defina um tipo de estrutura nomeada para representar pontos no plano através de suas coordenadas cartesianas. Em seguida, crie uma função para calcular a distância entre dois pontos e codifique um programa para testar seu funcionamento. Use a função *scanf()* para inicializar os campos membros das variáveis com valores lidos do teclado.

Dica: a distância entre dois pontos  $P$  e  $Q$ , conforme ilustração ao lado, é dada pela seguinte fórmula:

$$PQ = \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2}$$



#### 6.1.1. INICIALIZAÇÃO E ANINHAMENTO

Assim como vetores, estruturas globais ou estáticas também podem ser inicializadas na declaração. Para isso, basta fornecer os valores iniciais de seus membros entre chaves e separados por vírgulas.

**Exemplo 6.5.** Inicialização de uma variável do tipo *DATA*.

```
static DATA hoje = { 19, 7, 2000 };
```

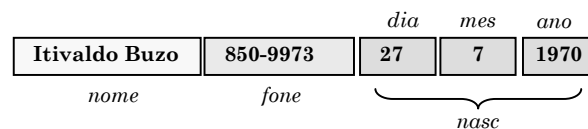
Os valores são atribuídos aos membros da estrutura *hoje* na ordem em que são fornecidos: o membro *dia* recebe o valor 19, o *mes*, 7 e o *ano*, 2000.

É possível criar um tipo de estrutura em que um ou mais de seus membros também sejam estruturas, desde que os tipos de tais estruturas tenham sido previamente declarados no programa<sup>27</sup>.

**Exemplo 6.6.** Uma estrutura para armazenar dados de uma pessoa.

```
typedef struct {
    char nome[31];
    char fone[21];
    DATA nasc;
} PESSOA;
```

Note que o campo *nasc* é uma estrutura *aninhada*. Quando a declaração do tipo *PESSOA* é compilada, se o tipo *DATA* ainda não foi definido no programa, o compilador emite uma mensagem de erro. ☑



**Figura 6.2** – A estrutura *DATA* aninhada na estrutura *PESSOA*

Ao acessar membros em estruturas aninhadas, podemos usar o operador de seleção de membro (.) quantas vezes forem necessárias.

**Exemplo 6.7.** Acessando os membros de estruturas aninhadas.

```
PESSOA amigo;
strcpy(amigo.nome, "Itivaldo Buzo");
strcpy(amigo.fone, "850-9973");
amigo.nasc.dia = 27;
amigo.nasc.mes = 7;
amigo.nasc.ano = 1970;
```

Para acessar os campos *dia*, *mes* e *ano*, primeiro temos que acessar o membro *nasc*. Por exemplo, seria um erro escrever *amigo.dia*, já que a variável *amigo* é do tipo *PESSOA*, e esse tipo não tem nenhum campo chamado *dia*. ☑

**Exemplo 6.8.** Inicialização de estruturas aninhadas.

```
PESSOA amigo = { "Itivaldo Buzo", "850-9973", {27, 7, 1970} };
```

Note que, como o terceiro membro também é uma estrutura, sua inicialização requer um par adicional de chaves delimitando seus membros. ☑

<sup>27</sup> Os tipos de estrutura devem ser declarados, preferencialmente, como globais.

**Exercício 6.4.** Defina um tipo de estrutura para armazenar um horário composto de hora, minutos e segundos. Crie e inicialize uma variável desse tipo e, em seguida, mostre seu valor no vídeo usando o formato "99:99:99".

**Exercício 6.5.** Defina um tipo de estrutura para armazenar os dados de um voo como, por exemplo, os nomes das cidades de origem e destino, datas e horários de partida e chegada. Crie uma variável desse tipo e atribua valores aos seus membros usando a notação de ponto e, depois, inicialização.

**Exercício 6.6.** Usando o tipo definido no exercício 6.3, para armazenamento de pontos do plano cartesiano, defina um tipo de estrutura para representar segmentos de retas através dos seus extremos. Crie uma função para determinar o comprimento de um tal segmento e faça um programa para testá-la.

### 6.1.2. VETORES DE ESTRUTURAS

É possível combinar vetores e estruturas de muitas maneiras interessantes. Por exemplo, podemos ter uma estrutura contendo um membro do tipo vetor ou então um vetor cujos elementos sejam estruturas.

**Exemplo 6.9.** Uma agenda com dados de 5 pessoas.

```
PESSOA agenda[5];
```

A variável *agenda* é um vetor cujos elementos são estruturas do tipo *PESSOA*. Por exemplo, para atribuir valores ao segundo<sup>28</sup> elemento do vetor *agenda*, podemos escrever:

```
strcpy(agenda[1].nome, "Roberta Soares");
strcpy(agenda[1].fone, "266-0879");
agenda[1].nasc.dia = 15;
agenda[1].nasc.mes = 11;
agenda[1].nasc.ano = 1971;
```

Como uma *string* é, na verdade, um vetor cujos elementos são caracteres, para alterar a sétima letra do nome armazenado no segundo elemento do vetor *agenda*, escrevemos:

```
agenda[1].nome[6] = 'o';
```



☛ *Um vetor cujos elementos são estruturas é denominado tabela e é, geralmente, representado com seus elementos dispostos em linhas e os campos em colunas.*

---

<sup>28</sup> Lembre-se de que a indexação de vetores em C inicia-se em zero.

	nome	fone	nasc
0	Itivaldo Buzo	850-9973	27/07/1970
1	Roberto Soares	266-0879	15/11/1971
2	Márcia Ueji	576-8292	09/05/1966
3	Silvio Lago	851-7715	18/03/1968
4	Mie Kobayashi	834-0192	04/12/1973

Figura 6.3 – Um vetor de estruturas

A inicialização de uma tabela é similar àquela de uma matriz; como podemos perceber no exemplo a seguir.

**Exemplo 6.10.** Inicializando uma agenda.

```
static PESSOA agenda[] = {
    { "Itivaldo Buzo", "850-9973", {27, 7, 1970} },
    { "Roberto Soares", "266-0879", {15, 11, 1971} },
    { "Márcia Ueji", "576-8292", { 9, 5, 1966} },
    { "Silvio Lago", "851-7715", {18, 3, 1968} },
    { "Mie Kobayashi", "834-0192", { 4, 12, 1973} }
};
```



**Exercício 6.7.** Usando o tipo *PESSOA*, definido no exemplo 6.6, crie uma função para preencher uma agenda com os dados de 5 pessoas. Crie também uma função que para procurar na agenda o telefone de uma determinada pessoa e codifique um programa para testar essas funções.

**Exercício 6.8.** Usando o tipo de estrutura definido no exercício 6.2, crie e inicialize uma tabela com os dados de todos<sup>29</sup> os vôos de um aeroporto e codifique uma rotina para exibí-la em vídeo.

### 6.1.3 ORDENAÇÃO E BUSCA EM TABELAS

Sejam  $k_0, k_1, k_2, \dots, k_{n-1}$  chaves distintas e  $T = \langle (k_0, d_0), (k_1, d_1), \dots, (k_{n-1}, d_{n-1}) \rangle$  uma tabela cujos registros associam a cada chave  $k_i$  uma informação  $d_i$ , para  $0 \leq i < n$ . Dizemos que  $T$  é *ordenada* se e só se  $k_0 < k_1 < \dots < k_{n-1}$ . Dada uma particular chave  $k$  e uma tabela  $T$ , a *busca* consiste em determinar um índice<sup>30</sup>  $i$  tal que  $(k_i, d_i) \in T$  e que  $k = k_i$ .

<sup>29</sup> Suponha um número qualquer fixo.

<sup>30</sup> Se o registro não existe, a busca deve devolver um índice inválido.



**Exercício 6.9.** Usando o algoritmo de ordenação por seleção, codifique uma função para ordenar uma tabela cujos registros contêm nomes de pessoas e seus respectivos números de telefones. Use o *nome* como chave de ordenação.

**Exercício 6.10.** Usando o algoritmo de busca binária, codifique uma rotina que receba como entrada a tabela ordenada no exercício anterior e o nome de uma pessoa e exiba como saída o número de telefone correspondente. Caso o nome não conste da tabela<sup>31</sup>, deverá ser exibida uma mensagem de erro.

### Ordenação por chaves múltiplas

Uma vantagem do método da inserção, visto em 1.5.3, é que ele é *estável*, ou seja, itens com chaves iguais na seqüência desordenada mantêm suas posições relativas na seqüência ordenada. Isso significa que podemos usar o método de inserção para ordenar tabelas por mais de uma chave.

Seja  $L = \langle a_0, a_1, a_2, \dots, a_{n-1} \rangle$  uma seqüência e  $\rho(0), \rho(1), \rho(2), \dots, \rho(n-1)$  uma permutação dos índices de  $L$  tal que  $a_{\rho(0)} \leq a_{\rho(1)} \leq a_{\rho(2)} \leq \dots \leq a_{\rho(n-1)}$ . O método que determina uma tal permutação é estável se  $\rho(i) < \rho(j)$ , sempre que  $a_{\rho(i)} = a_{\rho(j)}$  e  $i < j$ .

**Exemplo 6.11.** Seja  $L = \langle (c,2), (a,2), (b,3), (c,1), (a,3), (a,1), (c,3), (b,2) \rangle$  uma tabela cujos registros desejamos ordenar por ambos os campos. Ordenando  $L$  pelo segundo campo, obtemos a seqüência  $L_1 = \langle (c,1), (a,1), (b,1), (c,2), (a,2), (b,3), (a,3), (c,3) \rangle$ . Agora, usando inserção, ordenamos  $L_1$  pelo primeiro campo e obtemos  $L_2 = \langle (a,1), (a,2), (a,3), (b,1), (b,3), (c,1), (c,2), (c,3) \rangle$ . O resultado final  $L_2$  é uma tabela ordenada por ambos os campos. Se o método da inserção não fosse estável, a ordenação de  $L_1$  pelo primeiro campo destruiria a ordenação já obtida para o segundo campo. ☑

**Exercício 6.11.** Considere a tabela do último exemplo. Crie duas versões da função *insercao()*, uma para ordená-la pelo primeiro campo e outra para ordená-la pelo segundo campo. Faça um programa que use essas funções para obter a tabela ordenada por ambos os campos.

**Exercício 6.12.** Repita o exercício anterior usando o método da seleção, em vez da inserção, e verifique se ele também é estável.

---

<sup>31</sup> Certifique-se de que a rotina funcione corretamente, independentemente do nome ser fornecido em maiúsculas ou minúsculas.

## 6.2. UNIÕES

---

Uma *união* é um tipo especial de estrutura capaz de armazenar um único membro por vez. Ao contrário da estrutura normal, os membros de uma união não ocupam posições consecutivas. Na verdade, todos eles são armazenados a partir do mesmo endereço de memória e, por esse motivo, apenas um deles pode existir em um dado momento.

Uma união é um recurso que nos permite armazenar diferentes tipos de dados num mesmo local da memória. Quando aloca espaço para uma variável do tipo união, o compilador sempre se baseia no tamanho do seu maior membro. Assim, para o exemplo a seguir, o compilador alocaria um espaço de 4 *bytes*, que é o espaço necessário para o tipo *float*.

**Exemplo 6.12.** Um tipo mutante.

```
typedef union {  
    char  a;  
    int   b;  
    float c;  
} mutante;
```

Uma variável do tipo *mutante* pode armazenar um caracter, um inteiro ou um real. É responsabilidade do programador manter registro de qual dos três tipos de dados a variável está armazenando em cada momento. ☒

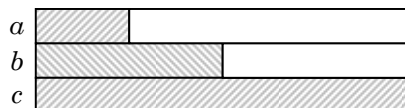


Figura 6.4 – Uma união do tipo *mutante*

**Exercício 6.13.** Alguns registradores do microprocessador 80286 são áreas de memória de 16 bits que podem também ser usadas como dois registradores de 8 bits. Por exemplo, o *byte* superior do registrador AX pode ser acessado como AH e o inferior, como AL. Assim, se o valor 0x1234 é armazenado em AX, o registrador AH fica valendo 0x34 e o AL, 0x12. Defina um tipo de dados capaz de representar um tal registrador e codifique um programa para testá-lo.

### 6.2.1. UNIÕES ETIQUETADAS

Um meio bastante conveniente de se usar uma união é criar uma estrutura contendo dois campos: a união propriamente dita e uma *etiqueta*, que especifica qual membro da união está sendo usado. Para cada membro da união, deve ser associado um valor distinto correspondente a sua etiqueta.

☛ *Através da etiqueta, em qualquer instante, o programador tem como saber exatamente qual membro da união está ativo. Isso evita que um valor seja inadvertidamente armazenado num formato e recuperado em outro.*

**Exemplo 6.13.** O tipo *mutante* melhorado.

```
typedef struct {
    int etiqueta;
    union {
        char a; /* 1 */
        int b; /* 2 */
        float c; /* 3 */
    } valor;
} mutante;
```

Agora o tipo *mutante* é uma estrutura composta de uma etiqueta e de uma união. A finalidade do campo etiqueta é indicar qual membro da união está em uso: caso seja o membro *a*, a etiqueta armazenará o valor 1; se for o membro *b*, a etiqueta receberá o valor 2; e se for *c*, o valor 3. ☑

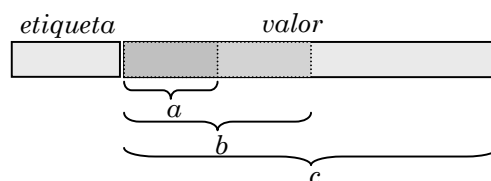


Figura 6.5 – Uma união etiquetada

**Exemplo 6.14.** Usando uma variável do tipo *mutante*.

```
mutante m;
```

Para usar o membro inteiro da união em *m*, por exemplo, fazemos:

```
m.etiqueta = 2;
m.valor.a = 459;
```

Se for preciso exibir o valor de *m*, podemos usar o código:

```
switch( m.etiqueta ) {
    case 1 : printf("%c", m.valor.a); break;
    case 2 : printf("%d", m.valor.b); break;
    case 3 : printf("%f", m.valor.c); break;
    default: printf("\netiqueta indefinida");
}
```

☑

**Exercício 6.14.** Usando uma união etiquetada, defina um tipo de dados polimórfico para representar figuras geométricas, como retângulos e círculos, e crie uma função para calcular a área de uma figura.

### 6.3. CAMPOS DE BITS

---

Ao contrário da maioria das linguagens, C oferece um método intrínseco para acessar um único *bit* dentro de um *byte*. Isso pode ser feito através da criação de um tipo especial de estrutura cujos campos são dimensionados em *bits*. Cada um desses campos deve ser declarado como *signed* ou *unsigned*<sup>32</sup>.

Para dimensionar um campo de *bits*, basta declará-lo com o sufixo *:n*, sendo *n* o seu tamanho.

**Exemplo 6.15.** Definindo campos de *bits*.

```
typedef struct {
    unsigned a : 1;
    signed   b : 3;
    unsigned c : 3;
} amostra;
```



Como o valor de um campo não pode invadir o espaço de outro, os *bits* de ordem superior de um valor são automaticamente truncados toda vez que ele excede a capacidade do campo. Então, o campo *a* só pode ter valor 0 ou 1, o campo *b* pode conter valores no intervalo de -4 a +3 e o campo *c*, de 0 a 7.

**Exemplo 6.16.** Ultrapassando os limites de um campo.

```
static amostra x = { 14, 14, 14};
printf("%d %d %d", x.a, x.b, x.c);
```

A saída produzida por esse fragmento de código é 0 -2 6. Para entender o porquê, lembre-se de que o valor 14 em binário é 1110. Quando esse valor é atribuído ao campo *a*, que tem um único *bit*, os seus três *bits* mais à esquerda são descartados, sobrando apenas o *bit* 0. Analogamente, para os campos *b* e *c*, o valor armazenado é 110. Como o campo *c* é sem sinal, seu valor será 6. No caso do campo *b*, entretanto, o *bit* mais à esquerda de 110 representa o sinal do número em complemento de dois<sup>33</sup> e, portanto, seu valor será -2.

☛ *Um campo de n bits sem sinal pode armazenar inteiros no intervalo de 0 a  $2^n - 1$  e, com sinal, em complemento de dois, no intervalo de  $-2^{n-1}$  a  $2^{n-1} - 1$ .*

A verificação dos limites é feita não só quando os campos são inicializados, mas também quando são alterados.

---

<sup>32</sup> Se o campo tem um único *bit*, então ele deve ser, necessariamente, *unsigned*.

<sup>33</sup> Para inverter o sinal de um número em complemento de dois, inverta todos os seus *bits* e some 1 ao resultado.

**Exemplo 6.17.** *Overflow e underflow.*

```
static amostra x = { 0, -4, 7};  
printf("%d %d %d", ++x.a, --x.b, ++x.c);
```

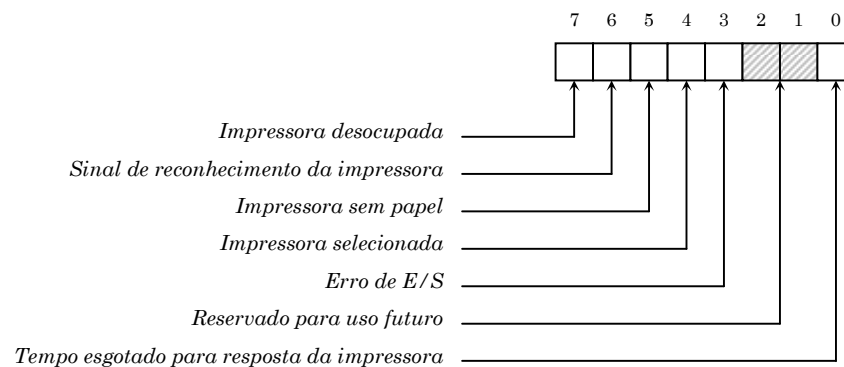
A saída produzida agora é 1 3 0. O valor do campo *a* é o que já esperávamos, mas os valores de *b* e *c* podem causar surpresa. Para entender como esses valores aparecem, é melhor trabalhar com binários. O valor do campo *c* em binário é 111 e, quando incrementado, passa a ser 1000. Como o campo *c* só comporta 3 bits, o seu valor fica sendo 000. No caso de números sinalizados, temos  $-4_D \equiv 100_B$  e  $-1_D \equiv 111_B$ . Então, quando o campo *b* é decrementado, obtemos a soma  $(-4_D) + (-1_D) \equiv (100_B) + (111_B) \equiv 1011_B$ , cujo *bit* superior é descartado. Logo, o valor do campo *b* fica sendo  $011_B \equiv 3_D$ . ☒

Embora bastante úteis, campos de *bits* estão sujeitos a várias restrições:

- não é possível tomar o endereço de um campo;
- os campos não podem ser organizados como vetores;
- um campo não pode ultrapassar os limites de uma palavra de memória;
- a ordem de alocação dos campos numa palavra depende da máquina.

### 6.3.1. ACESSANDO UM DISPOSITIVO DE HARDWARE

Campos de *bits* são tipicamente usados para forçar uma estrutura a corresponder exatamente a alguma representação fixa do *hardware*; por exemplo, para analisar a entrada de algum dispositivo.



**Figura 6.6** – *Interpretação do byte de status da impressora paralela*

A figura acima mostra a configuração de um *byte* lido de uma porta paralela onde está conectada uma impressora. Para representar essa configuração, podemos criar uma estrutura de campos de *bits*.

**Exemplo 6.18.** Estrutura para o *byte* de *status* da impressora paralela.

```
typedef struct {
    unsigned t_esgotado : 1;
    unsigned reservado : 2;
    unsigned erro_de_ES : 1;
    unsigned selecionada : 1;
    unsigned sem_papel : 1;
    unsigned reconhecida : 1;
    unsigned desocupada : 1;
} status;
```



A função *biosprint()*, definida em *bios.h*, tem o seguinte protótipo:

*int biosprint(int cmd, int abyte, int port)*, sendo que:

- o parâmetro *cmd* determina qual o comando a ser executado.  
0: enviar um byte; 1: inicializar a porta e 2: obter o status
- o parâmetro *abyte* indica o valor a ser enviado pela porta quando o comando for "enviar um byte".
- o parâmetro *port* especifica qual porta paralela deve utilizada  
0: LPT1; 1: LPT2; ...

Então, usando essa função, podemos acessar a porta paralela e preencher uma estrutura de campos de *bits* com o *status* da impressora.

**Exemplo 6.19.** Obtendo o *byte* de *status* da impressora.

```
status getstatus(int porta) {
    union {
        unsigned byte;
        status config;
    } s;
    s.byte = biosprint(2,0,porta);
    return s.config;
}
```

Através de uma união, os *bits* do valor retornado por *biosprint()* são transferidos para seus respectivos campos dentro de uma estrutura *status*.

**Exemplo 6.20.** Usando a função *getstatus()*.

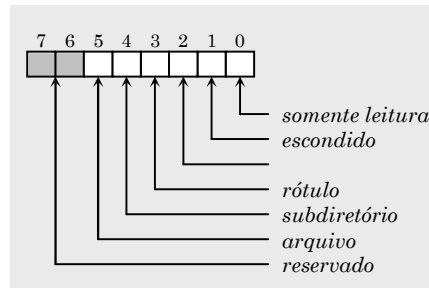
```
status lpt1 = getstatus(0);
if( lpt1.sem_papel )
    puts("Coloque papel na impressora!!!");
```

Observe como o uso de campos de *bits* aumentam consideravelmente a legibilidade desse fragmento de código.

**Exercício 6.15.** A cada arquivo no DOS está associado um *byte* de atributo cujo valor determina que operações podem ser realizadas com o arquivo. Por exemplo, um arquivo "somente leitura" não pode ser apagado pelo comando *del* e um "escondido" não aparece na listagem do diretório. Esse *byte* é acessível através da função *\_chmod()*, que recebe os seguintes argumentos:

- ① o *nome* do arquivo
- ② a *operação* desejada (0: *lê* e 1: *grava*)
- ③ um novo *atributo*, passado somente se a operação for de gravação

Usando essas informações, crie um programa para proteger contra gravação ou esconder arquivos visíveis (e vice-versa). Mantenha os demais *bits* inalterados e teste o seu programa com um arquivo que possa ser apagado.



### 6.3.2. ECONOMIZANDO ESPAÇO DE ARMAZENAMENTO

Outra aplicação típica de campos de *bits* é para economizar memória quando o espaço de armazenamento disponível é limitado. Por exemplo, no sistema operacional DOS, as datas de criação dos arquivos são armazenadas em 16 *bits*, num formato compactado, conforme ilustrado a seguir.

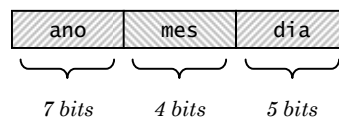


Figura 6.7 – Uma data compactada em campos de bits

O número de *bits* necessários para representar  $n$  valores distintos é aproximadamente igual a  $\log_2 n$ . Como o campo *dia* pode assumir no máximo 31 valores distintos, o espaço necessário para esse campo é  $\log_2 31 = 4.9 \approx 5$  *bits*. Analogamente, para o campo *mês*, precisamos de  $\log_2 12 = 3.6 \approx 4$  *bits*. Para o campo *ano*, entretanto, não há como estabelecer um valor máximo.

A solução adotada para a representação do ano baseia-se fato de que nenhum arquivo pode ter data de criação anterior a 1980, já que o DOS foi criado no início dos anos 80. Então, por convenção, o valor 0 foi escolhido para representar o ano de 1980. Assim, por exemplo, o ano 2000 é armazenado como  $2000 - 1980 = 20$ . Como restam ainda 7 *bits* livres, podemos representar valores de 0 a  $2^7 - 1$  e, portanto, esse esquema funcionará bem até o ano<sup>34</sup> de 2107.

<sup>34</sup> Se ainda existirem computadores, que nome será que vão inventar para esse bug?!!

**Exemplo 6.21.** Representando uma data compactada do DOS.

```
typedef struct {  
    unsigned dia : 5;  
    unsigned mes : 4;  
    unsigned ano : 7;  
} datacomp; ☑
```

A função *findfirst()* pode ser usada para obter a data de criação de um arquivo. Essa função exige como argumentos o *nome* do arquivo, uma estrutura do tipo *struct ffbk* e também o atributo do arquivo (o *default* é 0xFF).

**Exemplo 6.22.** A *struct ffbk* conforme definida em *dir.h*.

```
typedef struct {  
    struct ffbk {  
        char    ff_reserved[21];  
        char    ff_attrib;  
        unsigned ff_ftime;  
        unsigned ff_fdate;  
        long    ff_fsize;  
        char    ff_name[13];  
    }; ☑
```

Quando o arquivo desejado é encontrado, a função *findfirst()* preenche uma estrutura do tipo *struct ffbk* com seus dados e, então, podemos ter acesso a eles. A data de criação compactada fica disponível no campo *ff\_fdate* como um inteiro sem sinal, cujo valor deve ser transferido para os campos de *bits*.

A transferência é feita de maneira similar àquela que fizemos no exemplo 6.18, ou seja, definimos uma união com dois campos: um do tipo *unsigned* e outro do tipo *datacomp*. Então, usando essa união, armazenamos o valor o do campo *ff\_fdate* no formato de um inteiro e depois o recuperamos no formato de uma data compactada.

**Exemplo 6.23.** Exibindo a data de criação de um arquivo.

```
#include <stdio.h>  
#include <dir.h>  
  
typedef struct {  
    unsigned dia : 5;  
    unsigned mes : 4;  
    unsigned ano : 7;  
} datacomp;  
  
datacomp getfdate(char arq[]) {  
    struct ffbk dados;
```



```

    union {
        unsigned word;
        datacomp data;
    } d;
    findfirst(arq,&dados,0xFF);
    d.word = dados.ff_fdate;
    return d.data;
}

void main(void) {
    char arq[100];
    datacomp data;
    printf("\nArquivo? ");
    gets(arq);
    data = getfdate(arq);
    printf("Criado em %02d/%02d/%d", data.dia, data.mes, data.ano+1980);
}

```



**Exercício 6.16.** O horário de criação de um arquivo também é armazenado num formato compactado e está disponível no campo *ffftime* da estrutura *ffblk*. Analisando o valor binário desse campo para um arquivo conhecido, descubra qual o número de *bits* destinados ao armazenamento das horas, minutos e segundos. Declare um tipo de estrutura de campos de *bits* para armazenar uma hora compactada e altere o programa do exemplo 6.22 para exibir também a hora em que o arquivo foi criado.

**Exercício 6.17.** É possível listar todos os arquivos de um diretório procedendo da seguinte maneira: chame *findfirst()*, usando "\*" como nome de arquivo, para encontrar o primeiro deles; em seguida, use a função *findnext()* para encontrar os próximos arquivos do diretório. A função *findnext()* recebe como argumento apenas o endereço da estrutura *ffblk* que é preenchida quando um arquivo é encontrado. Para saber quando parar a listagem, verifique o valor retornado por essas funções, ambas devolvem -1 quando todos os arquivos já foram encontrados. Codifique um programa para listar todos os arquivos que foram criados numa data ou horário especificado pelo usuário.

# 7. PONTEIROS

*A implementação de estruturas de dados dinâmicas e seus respectivos algoritmos de manipulação requerem um conhecimento detalhado do uso de ponteiros. Nesse capítulo, apresentamos as principais idéias relacionadas ao uso de ponteiros em C e implementamos listas e árvores binárias.*

## 7.1. DEFINIÇÃO E USO

---

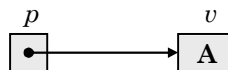
Um *ponteiro* é uma referência a uma posição de memória. Ponteiros podem ser constantes ou variáveis e através deles podemos acessar dados ou código. Em C, por exemplo, nomes de vetores são ponteiros constantes para dados e nomes de funções são ponteiros constantes para código.

**Exemplo 7.1.** Exibindo ponteiros constantes.

```
#include <stdio.h>
void main(void) {
    int v[2];
    printf("main = %p e v = %p", main, v);
}
```

A execução desse programa exhibe: main = 01FA e v = FFDA. ☑

Um ponteiro variável para dados contém o endereço<sup>35</sup> de uma outra variável. Se um ponteiro<sup>36</sup>  $p$  guarda o endereço de uma variável  $v$ , dizemos que  $p$  *aponta*  $v$  e representamos essa situação desenhando uma seta de  $p$  para  $v$ .



**Figura 7.1** – A representação gráfica de um ponteiro

O tipo de um ponteiro depende do tipo da variável que ele aponta. Na figura acima, por exemplo, o ponteiro  $p$  deve ser do tipo *char*; já que a variável apontada por ele armazena um caracter. Para indicar que uma variável é um ponteiro, devemos prefixar seu nome com um  $*$  na sua declaração.

---

<sup>35</sup> Endereços são números naturais que referenciam bytes específicos dentro da memória.

<sup>36</sup> Usaremos o termo *ponteiro* significando ponteiro variável para dados.

**Exemplo 7.2.** Sentenças em C correspondentes à figura 7.1.

```
char *p;  
char v = 'A';  
p = &v;
```

A primeira sentença declara  $p$  como ponteiro para caracter; a segunda cria uma variável  $v$  para armazenar a letra 'A' e a terceira atribui a  $p$  o endereço de  $v$ . Note que  $&v$  é um ponteiro constante para dados. ☒

Para acessar o local de memória apontado por um ponteiro, usamos também um  $*$  prefixando o nome da variável. Se  $p$  é um ponteiro,  $*p$  representa o conteúdo da área de memória apontada por  $p$ . Podemos pensar nesse asterisco como significando "siga a seta" e, assim,  $*p$  seria "siga a seta em  $p$ ".

**Exemplo 7.3.** Acesso através de ponteiro.

```
char x;  
x = *p;  
*p = 'B';
```

A primeira sentença declara  $x$  como caracter; a segunda armazena em  $x$  o conteúdo da área apontada por  $p$ , ou seja, a letra 'A' e a terceira atribui como novo conteúdo da área apontada por  $p$  a letra 'B'. ☒

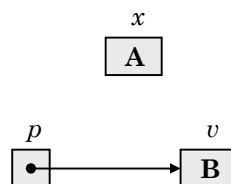


Figura 7.2 – A figura 7.1 após a execução das instruções no exemplo 7.3

Se um ponteiro  $p$  contém o endereço de uma variável  $v$ , então alterando o valor de  $*p$  estamos na verdade alterando o valor de  $v$ . É por esse motivo que o tipo de um ponteiro deve ser compatível com o tipo da variável que ele aponta. A quantidade de memória acessada através de um ponteiro depende do tipo do ponteiro e não do tipo da variável apontada.

☛ Se o tipo de um ponteiro  $p$  é diferente do tipo de uma variável  $v$ , então o endereço de  $v$  não deve ser atribuído a  $p$ , ou seja,  $p$  e  $&v$  não são compatíveis de atribuição.

**Exemplo 7.4.** Problemas com ponteiros de tipos incompatíveis.

```
#include <stdio.h>

void main(void) {
    int v = 0x4142;
    char *a;
    int *b;
    long *c;
    a = b = c = &v;
    printf("%c %x %lx", *a, *b, *c);
}
```

A saída do programa é 42 4142 FFDA4142. O acesso a partir de *b* é o único que consegue recuperar o dado corretamente. Através do ponteiro *a*, obtém-se apenas metade do valor e através de *c*, o dado é recuperado com lixo<sup>37</sup>. ☒

**Exercício 7.1.** Explique o significado de cada ocorrência de \* no fragmento de código a seguir e indique qual a saída exibida na tela.

```
int *p, x=5;
*p *= 2**p;
printf("%d", x);
```

#### 7.1.1. PASSAGEM POR REFERÊNCIA

Por *default*, argumentos em C são passados *por valor*. Isso quer dizer que quando uma função é chamada, um novo espaço de memória é alocado para cada um de seus parâmetros e os valores dos argumentos correspondentes são copiados nesses espaços. A consequência desse fato é que nenhuma alteração feita pela função em seus parâmetros pode afetar os valores dos argumentos que lhe foram passados.

Uma das vantagens da passagem por valor é que as funções ficam impedidas de acessar variáveis declaradas em outras funções. Entretanto, algumas vezes, vamos desejar que isso seja possível.

**Exemplo 7.5.** A necessidade da passagem por referência.

```
#include <stdio.h>

void perm(int p, int q) {
    int x;
    x = p;
    p = q;
    q = x;
}
```

---

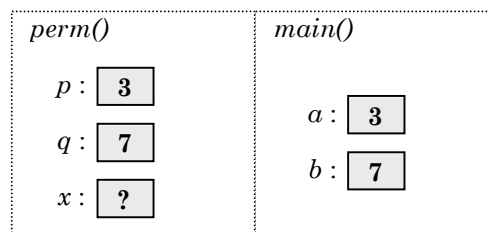
<sup>37</sup> Uma área de memória não inicializada contém lixo, ou seja, um valor indefinido. Nesse exemplo, o valor FFDA recuperado pelo ponteiro *c* é um lixo e, sendo assim, poderá ser diferente a cada vez que o programa for executado.

```

void main(void) {
    int a=3, b=7;
    perm(a,b);
    printf("%d %d", a, b);
}

```

O objetivo da função *perm()* é permutar os valores dos seus argumentos. Assim, a saída do programa deveria ser 7 3; entretanto, não é isso que acontece. Quando a chamada *perm(a,b)* é feita em *main()*, o compilador aloca os parâmetros *p* e *q* e copia para eles os *valores* dos argumentos *a* e *b*. Os valores de *p* e *q* são então permutados; porém, quando a função *perm()* termina, eles são desalocados. Retornando à função *main()*, os valores de *a* e *b* continuam inalterados e, portanto, *printf()* exibe 3 7. ☑



**Figura 7.3** – Alocação dos parâmetros da função *perm()*

A função *perm()* não funciona porque a passagem de argumentos é feita por valor. Os parâmetros *p* e *q* são independentes dos argumentos *a* e *b*; exceto, é claro, pelo fato de terem, inicialmente, os mesmos valores de *a* e *b*.

Para que uma função possa alterar os valores de seus argumentos, é preciso que eles sejam passados *por referência*. Nesse caso, em vez dos valores, os parâmetros recebem os endereços de seus respectivos argumentos.

**Exemplo 7.6.** Usando passagem por referência.

```

#include <stdio.h>
void permuta(int *p, int *q) {
    int x;
    x = *p;
    *p = *q;
    *q = x;
}
void main(void) {
    int a=3, b=7;
    permuta(&a,&b);
    printf("%d %d", a, b);
}

```

Nessa nova versão, a saída é 7 3. Quando a chamada `permuta(&a,&b)` é feita em `main()`, o compilador aloca os parâmetros `p` e `q` e copia neles os *endereços* dos argumentos `a` e `b`. Assim, as alterações feitas em `*p` e `*q` estão sendo, de fato, feitas em `a` e `b`. ☒

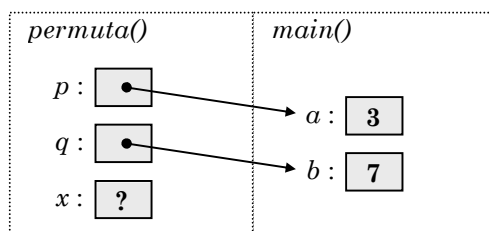


Figura 7.4 – Alocação dos parâmetros da função `permuta()`

**Exercício 7.2.** Codifique a função `minimax(v,n,a,b)`, que recebe um vetor `v` contendo `n` números reais e devolve em `a` e `b`, respectivamente, os valores mínimo e máximo entre aqueles armazenados em `v`.

**Exercício 7.3.** Explique o que aconteceria se na chamada `permuta(&a,&b)`, feita em `main()` no exemplo 7.6, o operador `&` fosse omitido.

### 7.1.2. PONTEIROS PARA PONTEIROS

Não é raro nos depararmos com uma situação em que um ponteiro precisa ser passado por referência a uma função. Nesse caso, o parâmetro dessa função deve ser declarado como um ponteiro para ponteiro. Embora a idéia possa parecer confusa, se tivermos o conceito de ponteiro sempre claro em mente, veremos que as coisas não são tão complicadas assim. Um ponteiro de ponteiro nada mais é que uma variável que contém o endereço de outra que, por sua vez, contém um endereço de memória onde há um dado.

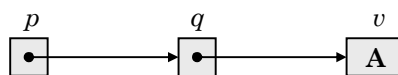


Figura 7.5 – Um ponteiro para ponteiro

Na figura acima, seguindo a seta de `p` chegamos a `q` e seguindo a seta de `q` chegamos a `v`. Para escrever isso em C, raciocinamos do seguinte modo:

- ① Para acessar `q`, seguimos a seta de `p`, ou seja,  $q \equiv *p$ .
- ② Para acessar `v`, seguimos a seta de `q`, ou seja,  $v \equiv *q$ .
- ③ Então, temos que  $v \equiv *(q) \equiv *(*p) \equiv **p$ .

**Exemplo 7.7.** Sentenças em C correspondentes à figura 7.5.

```
char v = 'A';  
char *q = &v;  
char **p = &q;
```



Ponteiros de ponteiros também são denominados *indireção múltipla*. Embora em situações práticas raramente tenhamos mais que dois níveis de indireção, a linguagem C não impõe nenhum limite à essa quantidade.

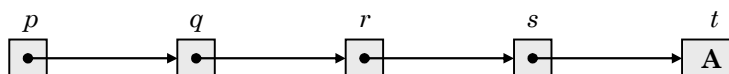


Figura 7.6 – Indireção múltipla



Quando usamos indireção múltipla, devemos colocar tantos asteriscos quantas forem as setas desenhadas na representação gráfica.

**Exercício 7.4.** Codifique um programa para criar a configuração representada na figura 7.6 e exibir a letra 'A' a partir de cada uma das variáveis.

### 7.1.3. ARITMÉTICA DE PONTEIROS

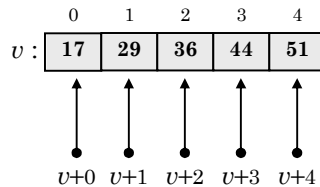
Uma operação bastante freqüente é a adição de ponteiros a números inteiros. Essa operação é implicitamente usada quando trabalhamos com vetores.

Conforme vimos, o nome de um vetor é um ponteiro constante que representa o endereço que ele ocupa na memória. Quando escrevemos  $v[i]$  para acessar o  $i$ -ésimo elemento de um vetor  $v$ , o compilador automaticamente executa a adição  $v+i$ . Essa adição representa o endereço do elemento que está a  $i$  posições do início do vetor  $v$ . Então, escrever  $v[i]$  equivale a escrever  $*(v+i)$ .

**Exemplo 7.8.** Usando notação de ponteiros com vetores.

```
#include <stdio.h>  
void main(void) {  
    static int v[5] = {17, 29, 36, 44, 51};  
    int i;  
    for(i=0; i<5; i++)  
        printf("%d ", *(v+i) );  
}
```

A saída produzida é 17 29 36 44 51; a mesma que teríamos obtido se escrevêssemos  $v[i]$  no lugar de  $*(v+i)$ , na instrução *printf()* acima.



**Figura 7.7** – Acesso ao vetor na notação de ponteiros

Um ponto importante sobre a adição de inteiros a ponteiros é que o resultado da adição depende do tipo do ponteiro. Isso acontece porque alguns tipos de dados requerem um espaço de memória maior do que os outros. Quando incrementado, um ponteiro deve saltar para o próximo elemento na memória e não para o próximo *byte*<sup>38</sup>.

Se um ponteiro  $p$ , de um determinado tipo  $t$ , é adicionado a um inteiro  $i$ , o endereço resultante é  $p+i*\text{sizeof}(t)$ , sendo  $\text{sizeof}(t)$  o tamanho do tipo  $t$  em bytes.

**Exemplo 7.9.** Com ponteiros, 1000+1 não é necessariamente 1001!

```
#include <stdio.h>
void main(void) {
    char  *a = (char *) 0x1000;
    int    *b = (int *) 0x1000;
    float  *c = (float *) 0x1000;
    double *d = (double *) 0x1000;
    printf("%p %p %p %p", a+1, b+1, c+1, d+1 );
}
```

A notação  $(t *) n$  representa um ponteiro constante, do tipo  $t$ , cujo valor é  $n$ . Sendo assim, todos os quatro ponteiros no programa acima apontam o mesmo endereço. Entretanto, como cada um deles tem um tipo diferente, a saída exibida será: 1001 1002 1004 1008. ☑

Como o nome de um vetor é um ponteiro constante, evidentemente, ele não pode ser alterado. Entretanto, se  $p$  é um ponteiro variável, então todos os operadores de adição e subtração podem ser usados com ele.

**Exemplo 7.10.** Operadores de adição e subtração com ponteiros e inteiros.

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

<sup>38</sup> Evidentemente, se o ponteiro é do tipo *char*, então o próximo elemento está no próximo *byte*.



```

    int *p = (int *) 0x1000;
    p++; printf("%p ", p );
    p-=3; printf("%p ", p );
    p+=2; printf("%p ", p );
    p--; printf("%p ", p );
}

```

A saída será 1002 0FFC 1000 0FFE. ☑

**Exercício 7.5.** Usando ponteiros, codifique a função *carrega(v,n)*, que preenche um vetor *v* com *n* valores lidos do teclado.

**Exercício 7.6.** Usando ponteiros, codifique a função *puts(s)*, que exibe uma *string s* no vídeo e, depois, muda o cursor de linha.

**Exercício 7.7.** Além da adição de ponteiros com inteiros, uma outra operação possível é a subtração entre ponteiros do mesmo tipo. Quando um ponteiro é subtraído de outro, o resultado é o número de elementos existentes no espaço de memória compreendido entre os endereços apontados por eles. Usando essa idéia, codifique a função *strlen(s)*, que devolve o número de caracteres existentes numa *string s*.

## 7.2. PONTEIROS E VETORES

---

Ponteiros podem ser organizados em vetores como qualquer outro tipo de dados. Se *v* é um vetor de ponteiros, então *\*v[i]* é o conteúdo da área apontada pelo *i*-ésimo elemento do vetor. Para declarar um vetor de ponteiros, basta prefixar seu nome com um asterisco.

**Exemplo 7.11.** Um vetor de ponteiros para inteiros.

```

#include <stdio.h>

void main(void) {
    static int a=10, b=20, c=30;
    static int *v[3] = {&a, &b, &c};
    int i;
    for(i=0; i<3; i++)
        printf("%d ", *v[i]);
}

```

A declaração `int *v[3]` cria um vetor cujos três elementos são ponteiros para inteiros. Note que a inicialização feita só é possível porque as variáveis *a*, *b* e *c* são estáticas; do contrário, o compilador não teria como saber seus endereços no momento de criar o vetor *v*. A saída do programa é 10 20 30. ☑

Embora possamos criar vetores de ponteiros para quaisquer tipos, geralmente, vetores de ponteiros são mais usados para o armazenamento de *strings*.

### 7.2.1. VETORES DE STRINGS

*Strings* podem ser armazenadas de duas maneiras distintas: como uma matriz de caracteres ou então como um vetor de ponteiros para caracteres.

**Exemplo 7.12.** Criando uma matriz de caracteres.

```
static char matcar[3][8] = { "azul", "verde", "amarelo" };
```

Na matriz, o espaço alocado para cada *string* é igual àquele necessário para armazenar a maior delas. Então, se a diferença entre os tamanhos das *strings* é muito grande, esse esquema pode consumir muito espaço. ☒

	0	1	2	3	4	5	6	7
0	a	z	u	l	\0			
1	v	e	r	d	e	\0		
2	a	m	a	r	e	l	o	\0

Figura 7.8 – Uma matriz de caracteres

**Exemplo 7.13.** Criando um vetor de ponteiros.

```
static char *vetpnt[3] = { "azul", "verde", "amarelo" };
```

No caso de vetor de ponteiros, as *strings* são armazenadas numa área contínua da memória e seus endereços são armazenados como elementos do vetor. Se as *strings* têm tamanhos iguais, ou muito próximos, esse esquema pode consumir mais memória; pois, além do espaço usado por elas, precisamos também do espaço para armazenar seus endereços. ☒

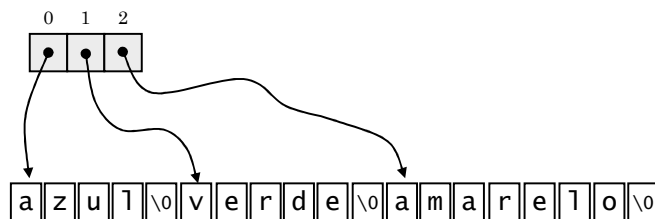


Figura 7.9 – Um vetor de ponteiros

**Exercício 7.8.** Suponha que você precise de uma variável para armazenar  $n$  *strings* lidas do teclado. É melhor declará-la como uma matriz de caracteres ou como um vetor de ponteiros? Justifique sua resposta.


### 7.2.2. ARGUMENTOS DA LINHA DE COMANDO

Toda vez que um programa C é executado a partir do sistema operacional, os argumentos digitados na linha de comando são coletados e disponibilizados à função *main()* sob a forma de dois parâmetros<sup>39</sup>: o primeiro deles, denominado *argc*, é um contador indicando o número de argumentos e o segundo, denominado *argv*, é um vetor que armazena os valores dos argumentos.

**Exemplo 7.14.** Argumentos para o comando *copy* do DOS.

```
C:\TC> COPY A.TXT A.BAK
```

Para essa linha de comando temos *argc*=3 e *argv*={"COPY", "A.TXT", "A.BAK"}. ☒

 *Note que os argumentos são disponibilizados na forma de strings e que argv é, de fato, um vetor contendo ponteiros para essas strings.*

Para criar um programa que possa receber argumentos da linha de comandos, basta declarar os parâmetros *argc* e *argv* em *main()*.

**Exemplo 7.15.** Um nova versão do comando *echo* do DOS.

```
/* ECO.C - o programa deve ser gravado com esse nome !!! */
#include <stdio.h>
void main(int argc, char *argv[]) {
    int i;
    for(i=1; i<argc; i++)
        puts(argv[i]);
}
```

Quando executado, esse programa exibe cada uma das palavras digitadas na sua linha de comando e termina:

```
C:\TC> ECO UM DOIS TRES
UM
DOIS
TRES
C:\TC> _
```

O primeiro argumento da linha é sempre o nome do programa e, portanto, *argv*[0] nesse exemplo é a *string* "ECO". Para evitar a exibição dessa *string*, o contador *i* foi iniciado com o valor 1. ☒

---

<sup>39</sup> Os nomes *argc* (*argument count*) e *argv* (*argument vector*) não são obrigatórios, mas já se tornaram um padrão.



*Se for necessário manipular numericamente os argumentos da linha de comando, podemos usar as funções de conversão `atoi()` e `atof()`, definidas em `stdlib.h`.*

**Exemplo 7.16.** Usando `atoi()` para exibir um gráfico de barras.

```
/* GRAFICO.C */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]) {
    int i, j;
    for(i=1; i<argc; i++) {
        putchar('\n');
        for(j=0; j<atoi(argv[i]); j++)
            putchar('*');
    }
}
```

Usando a função `atoi()` – *ASCII to integer* – cada argumento é convertido num número de asteriscos correspondente:

```
C:\TC> GRAFICO 5 6 9 7 4
*****
*****
*****
*****
*****
```



**Exercício 7.9.** A função `sleep()`, definida em `dos.h`, suspende a execução de um programa por um número específico de segundos. Usando essa função, implemente um programa com a sintaxe `CRONO <segundos>`, que exibe uma contagem regressiva dos segundos e, ao final, toca uma campainha.

**Exercício 7.10.** Usando `atof()` – *ASCII to float* – implemente um programa com a sintaxe `CALC <valor> <operador> <valor>`, que receba uma expressão aritmética simples, calcule e exiba seu valor.

**Exercício 7.11.** Usando a função `_chmod()`, vide exercício 2.11, codifique um programa com a sintaxe `PROTEGE <arquivo> [+|-|?]`, sendo que a opção `+` liga o atributo "somente leitura" do arquivo, a opção `-` desliga e `?` informa o seu estado corrente.

**Exercício 7.12.** Usando as funções `findfirst()` e `findnext()`, vide exercício 2.13, codifique um programa com a sintaxe `ENCONTRE <arquivo> <data>`, que lista o nome dos arquivos com data de criação anterior à data especificada.

## 7.3. PONTEIROS E FUNÇÕES

---

Já vimos que funções podem receber parâmetros do tipo ponteiro; de fato, isso é feito sempre que usamos passagem por referência. Agora, veremos também que podemos criar funções que devolvem ponteiros como resposta ou, ainda, que podemos criar ponteiros para apontar áreas de memória onde encontra-se o código de uma função.

### 7.3.1. FUNÇÕES QUE DEVOLVEM PONTEIROS

C não permite que vetores sejam devolvidos por uma função. Então, quando uma função precisa devolver um vetor, ela devolve seu endereço. Tipicamente, ponteiros são devolvidos por funções que precisam devolver *strings*. Para declarar que uma função devolve um ponteiro, basta prefixar o seu nome com um asterisco.

**Exemplo 7.17.** Obtendo o dia da semana por extenso.

```
char *dia_semana(int n) {
    static char *d[] = {
        "erro",
        "domingo",
        "segunda-feira",
        "terça-feira",
        "quarta-feira",
        "quinta-feira",
        "sexta-feira",
        "sábado"
    };
    return d[1<=n && n<=7 ? n : 0];
}
```

Quando chamada com um inteiro entre 1 a 7, a função devolve um ponteiro para uma *string* contendo o dia da semana correspondente por extenso. Para valores fora do intervalo, ela devolve um ponteiro para a *string* "erro". ☑

A chamada de uma função que devolve ponteiro não apresenta nenhuma novidade, como podemos ver no exemplo a seguir.

**Exemplo 7.18.** Usando a função *dia\_semana()*.

```
int n;
printf("\nDigite um número de 1 a 7: ");
scanf("%d", &n);
printf("\nDia da semana correspondente: %s", dia_semana(n));
```

Nesse fragmento de código, o ponteiro devolvido pela função é passado diretamente como argumento à função *printf()*. Uma outra alternativa seria armazenar essa resposta numa variável:

```
char *r;  
...  
r = dia_semana(n);
```

Nesse caso, a variável utilizada deve ter o mesmo tipo da função. ☑

**Exercício 7.13.** Codifique a função *strchr(s,c)* que devolve o endereço da primeira ocorrência do caracter *c* na *string s* ou, então, o valor *NULL* caso esse caracter não seja encontrado.

**Exercício 7.14.** Codifique a função *strsub(s,i,n)* que devolve a *substring* de *s* que inicia-se na posição *i* e tem, no máximo, *n* caracteres. A *string s* não deve ser alterada. [Dica: copie os caracteres para outro local de memória.]

**Exercício 7.15.** Codifique a função *monet(v)* que recebe um valor real e devolve uma *string* com a representação monetária desse valor. Por exemplo, a chamada *monet(1234.56)* deve devolver "R\$ 1.234,56" como resposta.

### 7.3.2. PONTEIROS PARA FUNÇÕES

Um poderoso recurso oferecido em C é a possibilidade de tomar o endereço de uma função, armazená-lo numa variável ponteiro e, depois, executá-la através desse ponteiro.

Para declarar um ponteiro de função, usamos a seguinte sintaxe:

*t (\*f)(t<sub>1</sub>, t<sub>2</sub>, ...),*

sendo *f* um ponteiro para uma função que devolve uma resposta do tipo *t* e recebe uma lista de parâmetros cujos tipos são *t<sub>1</sub>*, *t<sub>2</sub>*, etc.

Nem todas as operações permitidas com ponteiros de dados são permitidas com ponteiros de código; por exemplo, não é possível incrementar nem decrementar ponteiros de função. Em C, um ponteiro de código pode ser usado apenas para receber o endereço de uma função e chamar a função apontada.



Assim como ocorre no caso de vetores, o nome de uma função representa o endereço que em que seu código se encontra armazenado na memória.

Para chamar uma função a partir de um ponteiro, basta usar um par de parênteses<sup>40</sup> seguindo o nome desse ponteiro. Por exemplo, se *f* aponta uma

---

<sup>40</sup> Em C, os parênteses são, de fato, um operador de chamada de função.

função que recebe um argumento inteiro, então  $f(5)$  chama essa função passando a ela o valor 5 como entrada.

**Exemplo 7.19.** Usando um ponteiro para função.

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

void main(void) {
    double (*p)(double);
    p = sqrt;
    printf("%.11f", p(25) );
}
```

A declaração `double (*p)(double)` cria um ponteiro  $p$  capaz de apontar uma função do tipo *double*, que recebe um argumento do tipo *double*. Em seguida, o endereço da função `sqrt()` é atribuído a  $p$ . Quando a chamada `p(25)` é feita, estamos na verdade chamando a função `sqrt()` e, portanto, a raiz quadrada do número 25 é exibida como saída do programa. ☑

Ponteiros de função também podem ser agrupados num vetor. Isso nos permite criar programas cujo fluxo de execução pode ser desviado sem o uso de estruturas de controle convencionais<sup>41</sup>. Geralmente, vetores de ponteiros para funções são empregados na implementação de menus, em que a função executada depende de uma opção escolhida pelo usuário. Evidentemente, num vetor, todos os ponteiros devem ser do mesmo tipo; isso quer dizer que todas funções apontadas devem ter o mesmo protótipo.

**Exemplo 7.20.** Usando um vetor de ponteiros para funções.

```
#include <stdio.h>

void abrir(void) { puts("\nabrindo..."); }
void editar(void) { puts("\neditando..."); }
void salvar(void) { puts("\nsalvando..."); }
void fechar(void) { puts("\nfechando..."); }

void main(void) {
    static void (*f[])(void) = {abrir, editar, salvar, fechar};
    int i;
    do {
        printf("\n0. Abrir");
        printf("\n1. Editar");
        printf("\n2. Salvar");
        printf("\n3. Fechar");
    }
```

---

<sup>41</sup> Um poderoso recurso que deve ser usado com muito cuidado, pois pode levar à implementação de programas completamente desestruturados e de difícil manutenção.

```

        printf("\nopcao: ");
        scanf("%d", &i);
        if( i>=0 && i<=3)
            f[i]();
    } while( op!=3 );
}

```

A declaração `void (*f[])(void)` cria um vetor  $f$  cujos elementos são ponteiros para funções do tipo *void*, que não recebem argumentos. Então, a instrução `f[i]()` chama a função cujo endereço é o  $i$ -ésimo elemento do vetor  $f$ . ☒

Provavelmente, um das aplicações mais interessante de ponteiros de funções seja possibilitar que uma função seja passada a outra como argumento. Isso nos permite implementar rotinas cujo processamento executado depende do tipo de entrada que passamos a ela. Para entender a vantagem disso, vamos considerar primeiro um exemplo em que esse recurso não é empregado.

**Exemplo 7.21.** A redundância de código.

```

int min(int v[], int n) {
    int i, x=v[0];
    for(i=1; i<n; i++)
        if( x>v[i] )
            x = v[i];
    return x;
}

int max(int v[], int n) {
    int i, x=v[0];
    for(i=1; i<n; i++)
        if( x<v[i] )
            x = v[i];
    return x;
}

```

Essas funções devolvem respectivamente, o valor mínimo e o máximo entre aqueles armazenados num vetor de  $n$  elementos. Exceto pelos seus nomes e pelas comparações realizadas nos comandos *if*'s em cada uma delas, essas funções são idênticas. ☒

**Exemplo 7.22.** Eliminando a redundância através de polimorfismo.

```

#include <stdio.h>
int menor(int x, int y) { return x>y; }
int maior(int x, int y) { return x<y; }
int minmax(int v[], int n, int (*cmp)(int,int)) {
    int i, x=v[0];

```



```

        for(i=1; i<n; i++)
            if( cmp(x,v[i]) )
                x = v[i];
        return x;
    }

void main(void) {
    int w[5] = {78,34,12,45,51};
    printf("%d ", minmax(w,5,menor));
    printf("%d ", minmax(w,5,maior));
}

```

Criamos a função *minmax()* cujo terceiro parâmetro *cmp* é um ponteiro para uma função do tipo *int*, que recebe dois argumentos também do tipo *int*. Esse parâmetro ponteiro de função nos permite encontrar, com a mesma função *minmax()*, tanto o máximo quanto o mínimo elemento do vetor. ☑

☛ *Ponteiros de código permitem que funções sejam passadas como argumentos a outras funções; um recurso essencial na implementação de rotinas polimórficas em C.*

**Exercício 7.16.** A biblioteca padrão da linguagem C oferece uma função polimórfica para ordenação de vetores cujo protótipo é o seguinte:

```
void qsort(void *v, int n, int t, int (*c)(const void *,const void *));
```

sendo *v* um ponteiro para o vetor a ser ordenado, *n* o número de elementos no vetor, *t* o tamanho em *bytes* de cada elemento e *c* um ponteiro para a função de comparação a ser usada durante ordenação. A novidade nesse protótipo é o tipo *void\**, que serve para declarar ponteiros capazes de receber qualquer tipo de endereço. Isso quer dizer que, por exemplo, o parâmetro *v* pode receber como valor inicial tanto o endereço de um vetor de caracteres quanto o endereço de um vetor de números reais ou ainda um vetor de estruturas. O único problema com ponteiros *void* é que não é permitido o acesso a dados a partir deles e, portanto, temos que usar *casts* com eles. Com base nessas informações, crie um vetor de estruturas e tente ordená-lo por cada um de seus campos, um de cada vez, usando a função *qsort()*.

## 7.4. PONTEIROS E ESTRUTURAS

Um ponteiro para uma estrutura não é diferente de um ponteiro para um tipo de dados básico. Se *p* aponta uma estrutura que tem um membro *m*, então podemos escrever *(\*p).m* para acessar esse membro a partir de *p*.

**Exemplo 7.23.** Ponteiro para estrutura.

```
#include <stdio.h>

typedef struct {
    char nome[31];
    int idade;
} pessoa;

void main(void) {
    pessoa *p, x = {"Silvio", 32};
    p = &x;
    printf("{%s,%d}", (*p).nome, (*p).idade );
}
```

Note que o operador de seleção de membro (·) tem maior precedência que o operador de conteúdo (\*). Por esse motivo, o uso de parênteses é obrigatório em (\*p).nome e em (\*p).idade. A saída do programa será {Silvio,32}. ☒

Uma novidade com relação a estruturas é que temos um operador específico para acessar membros a partir de ponteiros. Usando esse operador, em vez de escrever (\*ponteiro).membro podemos escrever *ponteiro->membro*.

**Exemplo 7.24.** Usando o operador *seta*.

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

void main(void) {
    pessoa *p, x = {"Silvio", 32};
    p = &x;
    printf("{%s,%d}", p->nome, p->idade );
}
```



*Em geral, ponteiros para estruturas são usados como parâmetros por uma questão de eficiência. Se uma estrutura é muito grande, a passagem por valor pode consumir um tempo considerável durante a execução do programa.*

**Exercício 7.17.** Baseando-se no programa do exemplo 7.23, explique o que representa a expressão \*((\*p).nome). Rescreva-a, usando o operador ->.

**Exercício 7.18.** Defina um tipo de estrutura, para representar livros, contendo título, autor e ano de publicação. Em seguida, codifique uma função para preencher uma tal estrutura com dados obtidos via teclado.

**Exercício 7.19.** Baseando-se no exercício anterior, e usando notação de ponteiro, codifique uma função para preencher uma tabela cujos elementos são estruturas representando livros.

### 7.4.1. ALOCAÇÃO DINÂMICA DE MEMÓRIA

Uma das aplicações mais interessantes de ponteiros é a alocação dinâmica de memória, que permite a um programa requisitar memória adicional para o armazenamento de dados durante sua execução.

Em geral, após um programa ter sido carregado para a memória, parte dela permanece livre. Então, usando a função *malloc()*, o programa pode alocar um pedaço dessa área livre e acessá-lo através de um ponteiro.

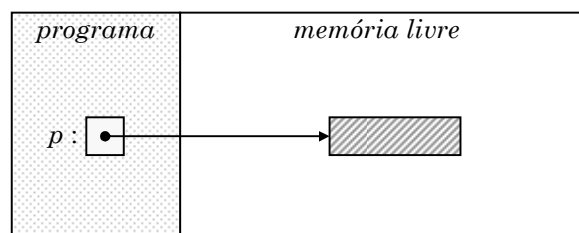


Figura 7.10 – Alocação dinâmica de memória

A função *malloc()* exige como argumento o tamanho, em *bytes*, da área de memória que será alocada. Então, se o espaço livre é suficiente para atender à requisição, a área é alocada e seu endereço é devolvido; senão, a função devolve um ponteiro especial, representado pela constante *NULL*. Além disso, como *malloc()* não tem conhecimento sobre o tipo dos dados que serão armazenados na área alocada, ela devolve um ponteiro do tipo<sup>42</sup> *void \**.

☛ *Ponteiros de dados do tipo `void *` são compatíveis de atribuição com ponteiros de quaisquer outros tipos e, por isso, são denominados ponteiros genéricos.*

Outro ponto importante é que, como cada tipo de dados pode requerer uma quantidade de memória diferente, dependendo da máquina que executa o programa, devemos usar o operador *sizeof*<sup>43</sup> para especificarmos a quantidade de memória a ser alocada.

Para usar a função *malloc()*, devemos incluir o arquivo *stdlib.h* ou *alloc.h*.

<sup>42</sup> Ponteiros desse tipo são compatíveis de atribuição com qualquer outro tipo de ponteiro.

<sup>43</sup> Esse operador aceita como argumento uma constante, uma variável, uma expressão aritmética ou um tipo de dados e devolve seu tamanho em bytes.

**Exemplo 7.25.** Vetores dinâmicos.

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

void main(void) {
    int *v, n, i;
    printf("\nTamanho do vetor? ");
    scanf("%d",&n);

    v = malloc( n*sizeof(int) );
    if( v!=NULL ) exit(1);
    for(i=0; i<n; i++) {
        printf("\n%do. Valor? ");
        scanf("%d",&v[i]);
    }
    while(i>=0 )
        printf("%d ",v[--i]);
}
```

O programa cria um vetor cujo tamanho é determinado pelo próprio usuário, em tempo de execução. Como o vetor deve ter capacidade para guardar  $n$  valores e cada um deles ocupa 2 *bytes*<sup>44</sup>, a área de memória alocada deve ter  $2n$  *bytes* de extensão; justamente o valor da expressão  $n*sizeof(int)$ . ☒

Quando um programa termina, todas as áreas alocadas pela função *malloc()* são automaticamente liberadas. Entretanto, se for necessário liberar explicitamente uma dessas áreas, podemos usar a função *free()*. Essa função exige como argumento um ponteiro para a área que será liberada.

**Exemplo 7.26.** Liberação de uma área alocada dinamicamente.

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

void main(void) {
    char *p;
    p = malloc(100*sizeof(char));
    if( p==NULL ) {
        puts("memória insuficiente");
        exit(1);
    }
    ...
    free(p);
    ...
}
```

☒

---

<sup>44</sup> No Turbo C, em outros compiladores e máquinas o *int* pode necessitar de 4 bytes.

**Exemplo 7.27.** Vetores, ponteiros, estruturas e *malloc()*, ufa!

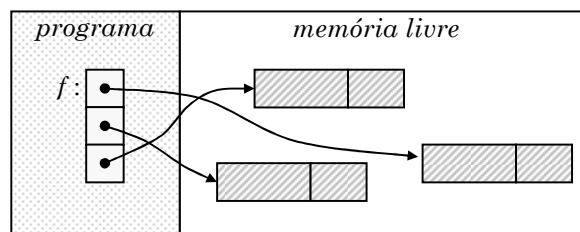
```
#include <stdio.h>
#include <alloc.h>

#define max 3

typedef struct {
    char nome[21];
    float salario;
} funcionario;

void main(void) {
    funcionario *f[max];
    int i;
    for(i=0; i<max; i++) {
        f[i] = malloc( sizeof(funcionario) );
        printf("\nNome: ");
        gets(f[i]->nome);
        printf("Salario: ");
        scanf("%f", &f[i]->salario);
    }
    ...
}
```

O programa cria um vetor *f* cujos elementos são ponteiros para estruturas do tipo *funcionario*. Dentro do laço *for*, as estruturas são alocadas dinamicamente e seus endereços são guardados nesse vetor. ☒



**Figura 7.11** – Um vetor de ponteiros para estruturas dinâmicas

**Exercício 7.20.** No exemplo 7.25 vimos o uso de vetores dinâmicos e em 7.27, vetores de ponteiros para estruturas dinâmicas. Altere o programa do exemplo 7.27 de modo que o vetor de ponteiros para estruturas *f* também seja alocado dinamicamente. [Dica: você vai precisar usar um *ponteiro de ponteiro*, para isso basta usar a declaração *funcionario \*\*f*. Antes de programar, tente esquematizar como ficará a configuração da memória.]

### 7.4.2. LISTAS ENCADEADAS

Uma *lista encadeada* é um conjunto de *nós*<sup>45</sup> contendo dois campos: um que armazena um *item* e outro que aponta o seu *sucessor*. Ao contrário dos elementos de um vetor, os nós de uma lista não são, necessariamente, armazenados em posições consecutivas de memória. Então, para acessar um nó arbitrário na lista, precisamos iniciar no seu primeiro<sup>46</sup> nó e ir seguindo os campos que apontam sucessores até que o nó desejado seja encontrado.

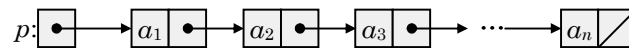


Figura 7.12 – Uma lista encadeada

Formalmente, uma lista encadeada é definida assim:

- (i) uma lista encadeada é um ponteiro para um nó.
- (ii) um nó é um par ordenado (*item, prox*) onde *item* é um dado e *prox* é uma lista encadeada.

☛ A lista encadeada é uma estrutura recursiva: ela é um ponteiro para um nó que, por sua vez, aponta uma lista encadeada. A base dessa recursão é a lista vazia.

**Exemplo 7.28.** Definição de lista encadeada de caracteres em C.

```
typedef struct no {  
    char item;  
    struct no *prox;  
} *lista;
```

Esse código declara o tipo *lista* como ponteiro para o tipo *struct no*. ☑

Como apenas o primeiro nó de uma lista encadeada é imediatamente acessível, a implementação das operações de inserção, remoção e acesso é mais simples quando essas operações são restritas ao início da lista.

**Exemplo 7.29.** Inserção no início da lista encadeada.

```
void insere(lista *p, char x) {  
    lista n = malloc(sizeof(struct no));  
    assert( n!=NULL );
```

<sup>45</sup> Um nó também é denominado *nodo* ou *nódulo*.

<sup>46</sup> Para saber onde inicia a lista, é preciso manter um ponteiro para seu primeiro nó.

```

    n->item = x;
    n->prox = *p;
    *p = n;
}

```

Primeiro, um novo nó  $n$  é alocado. Então, o item  $x$  é armazenado nesse nó e o seu campo *prox* passa a apontar o nó que se encontra no início da lista  $*p$ . Finalmente, o valor de  $*p$  é atualizado e o novo nó  $n$  passa a ocupar o início da lista. Note que a lista deve ser passada por referência, já que é alterada pela função. O processo pode ser acompanhado na figura 9.2. ☑

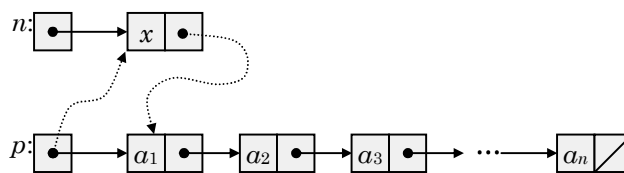


Figura 7.13 – Inserção no início da lista encadeada

Usando a função *insere()*, podemos criar uma lista iniciando com a lista vazia<sup>47</sup> e inserindo, repetidamente, novos elementos no seu início. Nesse caso, porém, a ordem dos elementos na lista será inversa àquela em que foram inseridos; o que pode ser indesejável em algumas aplicações.

**Exemplo 7.30.** Criação de uma lista usando inserção no início.

```

lista L = NULL;
insere(&L, 'a');
insere(&L, 'b');
insere(&L, 'c');

```

Esse fragmento de código cria a lista  $\langle c, b, a \rangle$ . Como o ponteiro  $L$  é passado por referência, chamando *insere()*, devemos fornecer seu endereço  $\&L$ . ☑

**Exemplo 7.31.** Remoção do início da lista encadeada.

```

void remove(lista *p) {
    lista n = *p;
    if( n==NULL ) return;
    *p = n->prox;
    free(n);
}

```

O nó do início da lista é apontado por  $n$ . Então, o ponteiro  $*p$  passa a apontar o segundo nó e o primeiro, apontado por  $n$ , é liberado. A função *remove()* também altera a lista e, portanto, deve receber seu endereço. ☑

<sup>47</sup> A lista vazia é representada por um ponteiro cujo valor é *NULL*.

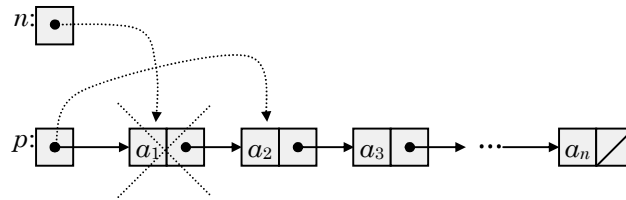


Figura 7.14 – Remoção do início da lista encadeada

**Exemplo 7.32.** Acesso ao  $i$ -ésimo nó da lista encadeada.

```
lista acessa(lista p, int i) {
    while( i-->1 && p!=NULL) p = p->prox;
    return p;
}
```

Como a função de acesso não precisa alterar a lista, o parâmetro  $p$  é passado por valor. Inicialmente apontando o começo da lista, o ponteiro  $p$  é sucessivamente deslocado para a próxima posição, até que o nó desejado seja encontrado; quando então seu valor é devolvido. ☑

A função *acessa()*, aliada às funções *insere()* e *remove()*, nos permite realizar operações de inserção, remoção ou acesso em qualquer posição de uma lista.

**Exemplo 7.33.** Inserindo um item na 3ª posição de uma lista  $L$ .

```
lista p = acessa(L,2);
insere(&p->prox, '*');
```

Primeiro a função *acessa()* é usada para se obter um ponteiro  $p$  para o 2º nó de uma lista encadeada  $L$ . Em seguida, a função *insere()* é chamada para acrescentar o novo item '\*' como sendo o sucessor daquele apontado por  $p$ . ☑

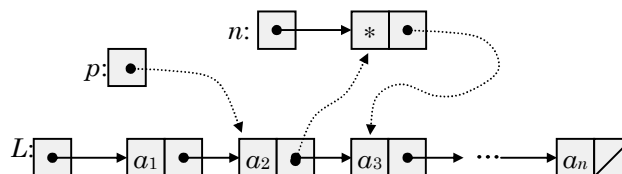


Figura 7.15 – Inserção no meio da lista encadeada

**Exemplo 7.34.** Removendo o 4º item de uma lista  $L$ .

```
lista p = acessa(L,3);
remove(&p->prox);
```

Primeiro obtém-se um ponteiro  $p$  para o 3º nó da lista  $L$ . Depois, o sucessor do nó apontado por  $p$  é removido. ☑



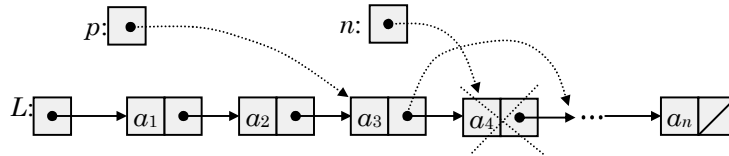


Figura 7.16 – Remoção do meio da lista encadeada

**Exemplo 7.35.** Alterando o 2º item de uma lista  $L$ .

```
lista p = acessa(L,2);
p->item = '*';
```

Obtém-se um ponteiro  $p$  para o 2º nó da lista  $L$  e altera-se o item desejado. ☒

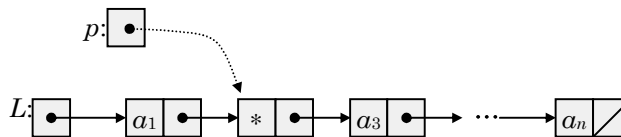


Figura 7.17 – Alteração de um item da lista

**Exercício 2.21.** Altere a função *remove()* de modo que o valor do item removido da lista seja devolvido como resposta da função.

**Exercício 7.22.** Verifique o que acontece quando a função *acessa()* é chamada para acessar uma posição antes da primeira ou então após a última.

**Exercício 7.23.** Codifique a função *destroi(x)*, que libera todos os nós existentes numa lista encadeada  $x$ .

**Exercício 7.24.** Codifique a função *concatena(x,y)*, que anexa a lista encadeada  $y$  no final da lista  $x$ .

#### 7.4.3. TRATAMENTO RECURSIVO DE LISTAS

Uma lista pode ser vista como um par ordenado (*cabeça,cauda*) onde *cabeça* é o seu primeiro item e *cauda* é a sublista obtida com a exclusão da cabeça da lista. Decompor uma lista nesses dois componentes é trivial, pois há uma correspondência direta entre o par (*cabeça,cauda*) e um nó (*item,prox*). Por uma questão de legibilidade, entretanto, vamos definir as macros:

```
#define cabeca(p) ((p)->item)
#define cauda(p) ((p)->prox)
```

Essa forma de ver uma lista torna mais fácil não só "enxergar", como também descrever algoritmos recursivos para realizar a sua manipulação.

### Calculando o comprimento de uma lista.

Vamos começar com uma função que define, recursivamente, o comprimento<sup>48</sup> de uma lista:

$$\text{comprimento}(L) = \begin{cases} 0 & \text{se } L \text{ é vazia} \\ 1 + \text{comprimento}(\text{cauda}(L)) & \text{caso contrário} \end{cases}$$

Nessa definição, a base da recursão é o caso em que a lista  $L$  está vazia e, obviamente, o comprimento dela é 0. Se a lista não está vazia, então ela tem pelo menos um item, a cabeça. Portanto, se tivermos o comprimento da cauda de  $L$ , o comprimento de  $L$  será apenas uma unidade a mais, ou seja, será  $1 + \text{comprimento}(\text{cauda}(L))$ .

Em geral, as funções recursivas ficam mais claras quando *simulamos* o seu funcionamento para uma entrada específica. De qualquer forma, sua corretude pode ser provada facilmente usando indução finita.

**Exemplo 7.36.** Cálculo recursivo do comprimento da lista  $\langle a, b, c \rangle$ .

$$\begin{aligned} \text{comprimento}(\langle a, b, c \rangle) &\equiv 1 + \text{comprimento}(\langle b, c \rangle) \\ &\equiv 1 + 1 + \text{comprimento}(\langle c \rangle) \\ &\equiv 1 + 1 + 1 + \text{comprimento}(\langle \rangle) \\ &\equiv 1 + 1 + 1 + 0 \\ &\equiv 3 \end{aligned}$$

Note que, como a cada chamada a sublista considerada é menor, fatalmente, a lista ficará vazia e, então, a recursão terminará. ☒

**Exemplo 7.37.** Implementação recursiva da função *comprimento()*.

```
int comprimento(lista L) {
    if( L==NULL ) return 0;
    else          return 1+comprimento(cauda(L));
}
```

☒

### Acessando o $n$ -ésimo item da lista.

Um outra função recursiva interessante é aquela que acessa o  $n$ -ésimo item de uma lista.

$$n\_ésimo(L, n) = \begin{cases} \text{cabeça}(L) & \text{se } n = 1 \\ n\_ésimo(\text{cauda}(L), n-1) & \text{se } n > 1 \end{cases}$$

---

<sup>48</sup> O mesmo que número de nós ou itens.

A base da função é o caso em que o item desejado é o primeiro da lista e, portanto, se  $n=1$ , a função devolve a *cabeça* da lista como resposta. Por outro lado, se  $n>1$ , a cabeça da lista não é o que nos interessa e, assim, podemos descartá-la, ficando apenas com a cauda da lista. Agora, acessar o  $n$ -ésimo item da lista  $L$  equivale a acessar o  $(n-1)$ -ésimo item da cauda de  $L$ .

**Exemplo 7.38.** Acesso recursivo ao 4º item da lista  $\langle a, b, c, d, e \rangle$ .

$$\begin{aligned} n\_ésimo(\langle a, b, c, d, e \rangle, 4) &\equiv n\_ésimo(\langle b, c, d, e \rangle, 3) \\ &\equiv n\_ésimo(\langle c, d, e \rangle, 2) \\ &\equiv n\_ésimo(\langle d, e \rangle, 1) \\ &\equiv d \end{aligned} \quad \checkmark$$

Note que chamar a função  $n\_ésimo()$  com uma lista vazia como argumento é considerado erro, já que ela não está definida para esse caso. Ao contrário do que ocorre no cálculo do comprimento da lista, no acesso aos itens, a lista vazia não é um caso minimal do problema.

**Exemplo 7.39.** Função recursiva para acesso ao  $n$ -ésimo item da lista.

```
char n_esimo(lista L, int n) {
    assert( L!=NULL );
    if( n==1 ) return cabeca(L);
    else      return n_esimo(cauda(L), n-1);
}
```

A função  $assert()$ , definida em *assert.h*, serve para garantir que o programa seja abortado caso a lista  $L$  esteja vazia. \checkmark

**Obtendo o valor de um item máximo da lista.**

Na definição da função  $máximo()$  a seguir, a função  $max()$  que devolve o maior entre dois valores.

$$máximo(L) = \begin{cases} cabeca(L) & \text{se } L \text{ é unitária} \\ max(cabeca(L), máximo(cauda(L))) & \text{caso contrário} \end{cases}$$

Se a lista tem um único item, esse item é o máximo da lista. Caso contrário, o máximo é o maior entre a cabeça e o máximo da cauda da lista.

**Exemplo 7.40.** Função recursiva para obter um item máximo lista.

```
char maximo(lista L) {
    assert( L!=NULL );
    if( cauda(L)==NULL ) return cabeca(L);
    else                  return max(cabeca(L), maximo(cauda(L)));
} \checkmark
```

**Exemplo 7.41.** Calculando o máximo da lista  $\langle d, b, e, a, c \rangle$ .

$$\begin{aligned} \text{máximo}(\langle d, b, e, a, c \rangle) &\equiv \max(d, \text{máximo}(\langle b, e, a, c \rangle)) \\ &\equiv \max(d, \max(b, \text{máximo}(\langle e, a, c \rangle))) \\ &\equiv \max(d, \max(b, \max(e, \text{máximo}(\langle a, c \rangle)))) \\ &\equiv \max(d, \max(b, \max(e, \max(a, \text{máximo}(\langle c \rangle)))) \\ &\equiv \max(d, \max(b, \max(e, \max(a, c)))) \\ &\equiv \max(d, \max(b, \max(e, c))) \\ &\equiv \max(d, \max(b, e)) \\ &\equiv \max(d, e) \\ &\equiv e \end{aligned}$$



**Exercício 7.25.** Defina e implemente funções recursivas para:

- a) acessar o *último* item de uma lista.
- b) verificar se um dado item *pertence* a uma lista.
- c) determinar o total de *ocorrências* de um dado item.
- d) calcular a *soma*<sup>49</sup> de todos os itens da lista.
- e) verificar se duas listas são *iguais*.

**Exercício 7.26.** Codifique procedimentos recursivos para.

- a) exibir uma lista.
- b) exibir uma lista em ordem inversa.
- c) inserir um item numa lista ordenada.
- d) remover um item de uma lista ordenada.
- e) liberar todos os nós de uma lista

---

<sup>49</sup> Suponha que os itens sejam de tipo numérico.

## 8. ARQUIVOS

*Se precisamos armazenar dados de modo que eles sejam mantidos após o término do programa, então o único meio é usar um arquivo. Nesse capítulo, mostramos como os arquivos são manipulados em C.*

### 8.1. PONTEIROS DE ARQUIVO

---

Um *arquivo* é uma coleção homogênea de itens que reside em disco. Arquivos são semelhante a vetores, diferindo apenas em dois aspectos:

- 1º vetores residem na memória RAM, enquanto arquivos residem em disco.
- 2º vetores têm tamanho fixo e predefinido, enquanto arquivos não.

A vantagem no uso de arquivos é que, diferentemente do que ocorre com vetores, os dados não são perdidos entre uma execução e outra. A desvantagem é que o acesso a disco é muito mais lento do que o acesso à memória e, conseqüentemente, o uso de arquivos torna a execução do programa mais lenta.

Para melhorar a eficiência, o sistema operacional usa uma área de memória denominada *buffer*. Os dados gravados pelo programa são temporariamente armazenados no *buffer*; quando ele fica cheio, o sistema o descarrega de uma só vez no disco. Analogamente, durante a leitura, o sistema se encarrega encher o *buffer* toda vez que ele fica vazio. Isso diminui o número de acessos a disco e, portanto, aumenta a velocidade de execução do programa.

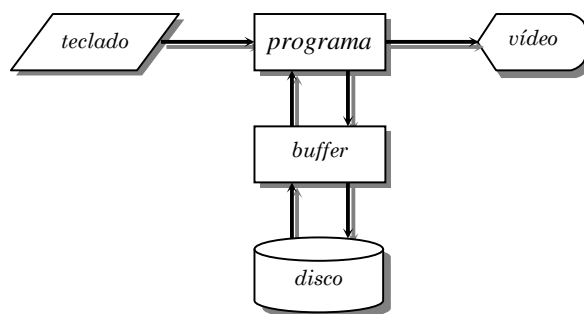


Figura 8.1 – Sistema de E/S bufferizada

Para usar um arquivo, precisamos criar um ponteiro do tipo *FILE*. Esse tipo, definido em *stdio.h*, é uma estrutura contendo, entre outras coisas, um ponteiro para o *buffer* e um ponteiro para a posição corrente dentro dele.

**Exemplo 8.1.** Criando variáveis de arquivo.

```
FILE *arq, *entrada, *saida;
```

Podemos criar diversas variáveis de arquivo, cada um delas podendo ser associada a um arquivo distinto no disco. ☒

☛ *Note que a variável de arquivo é apenas um ponteiro. A estrutura a ser apontada e o buffer serão alocados, de fato, somente no momento em que o arquivo for aberto.*

## 8.2. ARQUIVOS-PADRÃO

Ao entrar em execução, todo programa *C* tem à sua disposição cinco arquivos que são abertos, automaticamente, pelo sistema operacional. Esses arquivos são acessíveis através de ponteiros globais definidos em *stdio.h*.

Ponteiro	Finalidade	Dispositivo	Arquivo
<i>stdin</i>	entrada padrão	teclado	<i>con</i>
<i>stdout</i>	saída padrão	vídeo	<i>con</i>
<i>stderr</i>	saída padrão de erros	vídeo	—
<i>stdaux</i>	saída auxiliar	porta serial	<i>aux</i>
<i>stdprn</i>	saída de impressora	porta paralela	<i>prn</i>

Na verdade, quando usamos as funções *scanf()* e *printf()* estamos, implicitamente, usando dois desses arquivos: *stdin* e *stdout*. Para usá-los de maneira explícita, podemos empregar as funções *fscanf()* e *fprintf()*. Essas funções são idênticas às suas correlatas, exceto pelo fato de exigirem como primeiro argumento um ponteiro de arquivo.

**Exemplo 8.2.** Enviando dados à impressora.

```
#include <stdio.h>
void main(void) {
    fprintf(stdprn,"1, 2, 3, testando...\n");
}
```

☒

☛ *Chamar *printf(...)* equivale a chamar *fprintf(stdout,...)*. Analogamente para *scanf()*.*

### 8.2.1. REDIRECIONAMENTO DE E/S PADRÃO

Freqüentemente, programas precisam gravar ou acessar dados armazenados em disco. Uma forma simples de se conseguir isso é usar *redirecionamento* de E/S padrão. Considere o programa a seguir:

**Exemplo 8.3.** Uma versão do comando *cat* do sistema operacional *UNIX*.

```
/* cat.c */
#include <stdio.h>
void main(void) {
    int c;
    while( (c=getchar())!=EOF )
        putchar(c);
}
```

Esse programa é extremamente simples. Ele simplesmente lê caracteres da entrada padrão e os exibe na saída padrão, até que o final do arquivo seja detectado, quando então a função *getchar()* devolve o valor *EOF*<sup>50</sup>. Além do valor *EOF*, um arquivo pode conter qualquer um dos 256 caracteres da tabela ASCII. Então, quando chamamos a função *getchar()*, podemos obter um entre 257 valores possíveis. Como uma variável do tipo *char* só armazena valores no intervalo de -128 a +127, a função *getchar()* teve que ser definida como sendo do tipo *int*. É por isso que a variável *c* foi declarada com esse tipo. ☑

**Exemplo 8.4.** Usando o programa *cat*.

```
C:\> cat
um.
um
texto.
texto
qualquer.
qualquer
^Z.
```

Note que as linhas em negrito são exibidas pelo programa e as demais são digitadas pelo usuário. Cada linha digitada é imediatamente ecoada no vídeo e o caracter ^Z é usado para indicar final de arquivo no *DOS*. ☑

Para redirecionar a entrada ou a saída de um programa usamos os sinais < e >, respectivamente, seguidos do nome do arquivo que será lido ou gravado. Os próximos exemplos mostram como podemos redirecionar a entrada e a saída do programa *cat* de modo a implementar operações bastante úteis.

---

<sup>50</sup> Abreviação de "end of file"

**Exemplo 8.5.** Usando o programa *cat* para criar um arquivo em disco.

```
C:\> cat > arq.txt
um.
texto.
qualquer.
^Z.
```

Como a saída foi redirecionada, as linhas digitadas pelo usuário, que antes eram enviadas ao vídeo, são agora enviadas para o arquivo *arq.txt*. ☒

**Exemplo 8.6.** Usando o programa *cat* para ler um arquivo do disco.

```
C:\> cat < arq.txt
um
texto
qualquer
```

Agora, a entrada que normalmente viria do teclado é lida diretamente do arquivo *arq.txt* e exibida na tela. ☒

**Exercício 8.1.** Mostre como o programa *cat* pode ser usado para:

- a) *imprimir um arquivo de texto na impressora.*
- b) *fazer uma cópia de um arquivo de texto.*

**Exercício 8.2.** Implemente um programa para ler um texto da entrada padrão, convertê-lo em maiúsculas ou minúsculas e exibi-lo na saída padrão. O tipo de conversão deve ser especificado através de uma chave (+ para maiúscula ou – para minúscula), passada como argumento na linha de comandos.

### 8.3. OPERAÇÕES BÁSICAS

---

Diferentemente dos arquivos-padrão, os arquivos criados pelo usuário precisam ser mantidos pelo próprio programa. Então, para criar e usar nossos próprios arquivos, precisamos antes conhecer algumas operações básicas

#### 8.3.1. ABERTURA DE ARQUIVO

A *abertura* de arquivo é a operação que estabelece a conexão entre o programa na memória e o arquivo residente em disco. É durante a abertura de arquivo que a estrutura do tipo *FILE* e o *buffer* de transferência de dados que ela aponta são alocados. Para abrir um arquivo precisamos de uma instrução da seguinte forma:

```
a = fopen("nome", "modo");
```



sendo que:

- *a* é uma variável cujo valor será o endereço da estrutura alocada para representar o arquivo ou *NULL*, caso o arquivo não possa ser aberto;
- *nome* especifica o nome pelo qual o arquivo é referenciado em disco;
- *modo* especifica se o arquivo será lido (*r*) ou gravado (*w*).

**Exemplo 8.7.** Abrindo um arquivo para leitura.

```
...
FILE *e;
e = fopen("entrada.dat", "r");
if( e==NULL ) {
    printf("\nArquivo não pode ser aberto");
    exit(1);
}
...
```

Esse código abre o arquivo *entrada.dat* para leitura, conforme indicado pelo modo "r" utilizado. Note que, para garantir que nenhum erro ocorreu e que o arquivo foi realmente aberto, o valor devolvido por *fopen()* é testado. ☑

**Exemplo 8.8.** Abrindo um arquivo para gravação.

```
...
FILE *s;
if( (s=fopen("saida.dat", "w")) == NULL ) {
    printf("\nArquivo não pode ser aberto");
    exit(1);
}
...
```

Agora o código abre o arquivo para gravação, já que foi usado o modo "w". Além disso, como o teste de erro é sempre necessário, é costume codificar a operação de abertura embutida diretamente dentro do condicional. ☑

### 8.3.2. FECHAMENTO DE ARQUIVO

Depois de usado um arquivo, precisamos fechá-lo. O *fechamento* de um arquivo é importante por dois motivos:

- 1º se o arquivo foi aberto para gravação e o *buffer* não foi completamente preenchido, o fechamento garante que ele seja descarregado em disco.
- 2º o fechamento libera a área de conexão alocada na abertura do arquivo.

Para fechar um arquivo específico, usamos uma instrução da forma *fclose(a)*, onde *a* é o ponteiro para esse arquivo. Para fechar todos os arquivos abertos, de uma só vez, usamos a função *fcloseall()*, que não requer argumentos.

**Exemplo 8.9.** Fechando arquivos.

```
...
FILE *a, *e, *s;
...
fclose(s); /* fecha somente o arquivo s */
fcloseall(); /* fecha os arquivos restantes */
...
```



### 8.2.3. VERIFICAÇÃO DE FINAL DE ARQUIVO

Durante a leitura de um arquivo, freqüentemente, precisamos saber se todos os seus dados já foram lidos. Para isso, usamos a função *feof()*, que informa quando o final de arquivo foi atingido.

**Exemplo 8.11.** Exibindo o conteúdo de um arquivo em vídeo.

```
#include <stdio.h>

void main(void) {
    FILE *e;
    char nome[100];
    int c;
    printf("\nArquivo? ");
    gets(nome);
    if( (e=fopen(nome,"r"))==NULL ) {
        printf("Arquivo não pode ser aberto\n");
        exit(1);
    }
    while( 1 ) {
        c = fgetc(e);
        if( feof(e) ) break;
        putchar(c);
    }
    fclose(e);
}
```

A função *fgetc()*, definida em *stdio.h*, lê um caracter do arquivo cujo ponteiro lhe é passado como argumento e devolve seu código ASCII.



**Exercício 8.3.** Crie um programa para fazer cópias de arquivos texto, transferindo caracter por caracter. O programa deve receber os nomes dos arquivos de origem e destino via argumentos da linha de comando e deve usar as funções *fgetc()* e *fputc()* para leitura e gravação, respectivamente. [Dica: a função *fputc()* deve receber como argumentos o caracter e o ponteiro do arquivo no qual ele será gravado, nessa ordem.]

## 8.4. MODO TEXTO VERSUS MODO BINÁRIO

---

Um arquivo pode ser aberto em modo texto ou em modo binário. Esses modos diferem em três pontos principais:

- *Armazenamento de números.* Em modo texto, para ser gravado no arquivo, o número precisa antes ser convertido em uma *string*. Quando é lido, o sistema o converte novamente em número. Já em modo binário, os números são armazenados em disco da mesma forma que aparecem na memória, sem que nenhuma conversão precise ser feita. Arquivos em modo texto têm a vantagem de serem legíveis, mas consomem mais espaço. Por outro lado, os arquivos em modo binário não são legíveis para nós, mas são mantidos num formato muito mais compacto.
- *Mudança de linha.* Em modo texto, o caracter '\n' é transformado no par de caracteres CR/LF<sup>51</sup>, antes de ser gravado, e esse par é transformado novamente em '\n', quando é lido do disco. Se o arquivo é aberto em modo binário, essa transformação não é feita.
- *Final de arquivo:* Em modo texto, o final de arquivo é indicado pela presença do caracter ASCII 26, gerado pela combinação de teclas ^Z. Se esse caracter é encontrado durante a leitura do arquivo, a função de leitura devolve *EOF*. Em modo binário o final de arquivo só é sinalizado quando todos seus setores já foram lidos. O número de setores ocupados pelo arquivo é mantido pelo próprio sistema operacional.



*O modo texto é o default, mas podemos indicá-lo explicitamente usando os modos "rt" ou "wt". Para o modo binário, precisamos usar "rb" ou "wb".*

### 8.4.1. E/S CHARACTER

A gravação e leitura de caracteres é feita pelas funções *fgetc()* e *fputc()*, tanto em modo texto quanto em modo binário. A única diferença no funcionamento dessas funções em um modo ou outro está no tratamento do caracter '\n' e no reconhecimento do final de arquivo, conforme já foi discutido.

Como exemplo do uso de arquivos binários vamos criar um programa para realizar criptografia de dados. A *criptografia* é uma operação que transforma uma mensagem legível em outra ilegível, visando manter o sigilo de informações. Evidentemente, precisamos ter uma forma de recuperar a mensagem original, o que é denominado *decriptografia*.

---

<sup>51</sup> *Carriage-Return (ASCII 13) e Line-Feed (ASCII 10)*

**Exemplo 8.12.** Criptografia "ingênua" de dados.

```
#include <stdio.h>

void main(int argc, char *argv[]) {
    FILE *e, *s;
    char c;
    if( argc!=3 ) {
        printf("Uso: cript <origem> <destino>\n");
        exit(1);
    }
    if( (e=fopen(argv[1],"rb"))==NULL ) {
        printf("Arquivo não pode ser aberto\n");
        exit(1);
    }
    if( (s=fopen(argv[2],"wb"))==NULL ) {
        printf("Arquivo não pode ser criado\n");
        exit(1);
    }
    while(1) {
        c = fgetc(e);
        if( feof(e) ) break;
        fputc( ~c, s);
    }
    fcloseall();
}
```

A criptografia realizada é muito simples e, na prática, seria muito fácil quebrar o sigilo de uma mensagem assim criptografada. A transformação é realizada pelo operador de negação bit-a-bit ( $\sim$ ), que inverte todos os bits de seu operando. Por exemplo, suponha que a variável  $c$  contenha a letra 'S', cujo código ASCII em binário é 01010011. Então,  $\sim c$  resulta no valor 10101100, que corresponde ao caracter '4'. A operação inversa é trivial.  $\square$

**Exercício 8.4.** A operação ou-exclusivo bit-a-bit ( $\wedge$ ), tem as seguintes propriedades: (i)  $x \wedge 0 = x$ , (ii)  $x \wedge x = 0$  e (iii)  $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ . Usando essa operação podemos criar um método de criptografia com senha. Seja  $m$  um caracter da mensagem e  $s$  um caracter da senha. Para criptografar  $m$ , fazemos  $m \wedge s$  e obtemos o caracter criptografado  $c$ . Para ter  $m$  de volta, basta fazer  $c \wedge s$ . Como  $c = (m \wedge s)$ , pelas propriedades acima, segue que  $c \wedge s = (m \wedge s) \wedge s = m \wedge (s \wedge s) = m \wedge 0 = m$ . Usando esse método, crie um programa para criptografia que receba a senha e os nomes dos arquivos de origem e destino via argumentos da linha de comando. [Dica: utilize as letras da senha ciclicamente, de modo que os caracteres da mensagem não sejam criptografados sempre com a mesma letra]

### 8.4.2. E/S FORMATADA

Em modo texto, dados formatados são gravados pela função *fprintf()* e lidos pela função *fscanf()*. Essas funções diferem de *printf()* e *scanf()* pelo fato de exigirem como primeiro argumento um ponteiro de arquivo.

Para exemplificar o uso de E/S formatada, vamos criar um programa para emitir uma listagem de pontuação dos candidatos de um concurso.

**Exemplo 8.13.** Pontuação dos candidatos.

```
#include <stdio.h>

void main(void) {
    FILE *e;
    int i, insc, ptos;
    char gab[6], resp[6];
    if( (e=fopen("cand.dat","r"))==NULL ) {
        printf("Arquivo não pôde ser aberto\n");
        exit(1);
    }
    printf("\nGabarito? ");
    gets(gab);
    while( 1 ) {
        fscanf(e, "%d %s", &insc, resp);
        if( feof(e) ) break;
        for(ptos=i=0; i<5; i++)
            if( resp[i]==gab[i] )
                ptos++;
        printf("\n%d %d",insc,ptos);
    }
    fclose(e);
}
```

Suponha que o arquivo *cand.dat* contenha as inscrições e respostas a seguir:

```
125 acbde
493 cdeaa
384 deabc
981 cadce
394 bbced
```

Então, fornecido o gabarito abcde, o programa produz a seguinte saída:

```
125 3
493 0
384 0
981 1
394 2
```



**Exercício 8.5.** Codifique um programa para criar o arquivo de candidatos utilizado no exemplo 8.13. O programa deve solicitar ao usuário o número de inscrição e as respostas dadas a cada questão pelo candidato.

#### 8.4.3. E/S BINÁRIA

Em modo binário, dados são gravados pela função *fwrite()* e lidos pela função *fread()*. Essas funções recebem quatro argumentos. O primeiro deles, no caso de *fwrite()*, é um ponteiro para a área de memória onde encontram-se os dados que serão gravados; no caso de *fread()*, esse ponteiro aponta o local da memória que receberá os dados lidos do disco. O segundo, é o número de *bytes* ocupados pelo item de dado a ser transferido, em geral, indicamos esse valor usando o operador *sizeof*. O terceiro argumento é o número de itens transferidos e, finalmente, o quarto argumento é o ponteiro de arquivo.

Vamos exemplificar o uso dessas funções desenvolvendo um pequeno cadastro de funcionários.

**Exemplo 8.12.** Criando um cadastro de funcionários.

```
#include <stdio.h>

typedef struct {
    char nome[31];
    float salario;
} Func;

void main(void) {
    FILE *s;
    Func f;

    if( (s=fopen("func.dat","wb"))==NULL ) {
        printf("Arquivo não pode ser criado\n");
        exit(1);
    }

    printf("Digite ponto para finalizar o cadastramento:\n");

    while(1) {
        printf("\nNome? ");
        gets(f.nome);
        if( !strcmp(f.nome, ".") ) break;
        printf("\nSalario? ");
        scanf("%f",&f.salario);
        fwrite(&f,sizeof(Func),1,s);
    }

    fclose(s);
}
```



**Exemplo 8.13.** Listando o cadastro de funcionários em vídeo.

```
#include <stdio.h>

typedef struct {
    char nome[31];
    float salario;
} Func;

void main(void) {
    FILE *e;
    Func f;

    if( (e=fopen("func.dat","rb"))==NULL ) {
        printf("Arquivo não pode ser aberto\n");
        exit(1);
    }

    printf("Nome\tSalario\n\n");
    while(1) {
        fread(&f,sizeof(Func),1,e);
        if( feof(e) ) break;
        printf("%s\t%.2f\n",f.nome,f.salario);
    }

    fclose(e);
}
```



**Exercício 8.6.** Crie um programa colocar o cadastro de funcionários em ordem alfabética. [*Dica:* utilize uma lista encadeada ordenada]

**Exercício 8.7.** Crie um programa para, dado nome de um funcionário, exibir o salário correspondente. Faça duas versões, uma usando busca linear e outra usando busca binária, ambas realizadas diretamente no arquivo. [*Dica:* use as funções *fseek()* e *ftell()* para realizar acessos aleatórios e para determinar o número de registros no arquivo. Consulte o sistema de ajuda do *Turbo C* para descobrir como usar essas funções.]

# TABELA ASCII

Cód	Car	Cód	Car	Cód	Car	Cód	Car	Cód	Car	Cód	Car	Cód	Car	Cód	Car
0		32		64	@	96	'	128	Ç	160	á	192	ˆ	224	Ó
1	☉	33	!	65	A	97	a	129	Ú	161	í	193	ˆ	225	ß
2	☉	34	"	66	B	98	b	130	é	162	ó	194	ˆ	226	Ô
3	♥	35	#	67	C	99	c	131	â	163	ú	195	ˆ	227	Ò
4	♦	36	\$	68	D	100	d	132	à	164	ñ	196	ˆ	228	ô
5	♣	37	%	69	E	101	e	133	à	165	Ñ	197	ˆ	229	Õ
6	♠	38	&	70	F	102	f	134	â	166	ª	198	ã	230	μ
7		39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ
8		40	(	72	H	104	h	136	ê	168	ˆ	200	ℓ	232	Ɔ
9		41	)	73	I	105	i	137	ë	169	®	201	ℝ	233	Ú
10		42	*	74	J	106	j	138	è	170	¬	202	ℙ	234	Û
11	♂	43	+	75	K	107	k	139	ï	171	½	203	ℙ	235	Ù
12	♀	44	,	76	L	108	l	140	î	172	¼	204	ℙ	236	Ý
13		45	-	77	M	109	m	141	ì	173	ı	205	=	237	Ÿ
14	♪	46	.	78	N	110	n	142	Ä	174	«	206	ℙ	238	ˆ
15	✱	47	/	79	O	111	o	143	Å	175	»	207	ˆ	239	ˆ
16	►	48	0	80	P	112	p	144	É	176	⋮	208	ð	240	-
17	◄	49	1	81	Q	113	q	145	æ	177	⋮	209	Ð	241	±
18	↑	50	2	82	R	114	r	146	Æ	178	⋮	210	Ê	242	ˆ
19	!!	51	3	83	S	115	s	147	ô	179		211	Ë	243	¾
20	¶	52	4	84	T	116	t	148	ö	180	†	212	È	244	¶
21	§	53	5	85	U	117	u	149	ò	181	Á	213		245	§
22	ˆ	54	6	86	V	118	v	150	û	182	Â	214	Î	246	÷
23	↑	55	7	87	W	119	w	151	ù	183	À	215	Ï	247	,
24	↑	56	8	88	X	120	x	152	ÿ	184	©	216	İ	248	°
25	↓	57	9	89	Y	121	y	153	ö	185	¶	217	Ɔ	249	ˆ
26	→	58	:	90	Z	122	z	154	Ü	186		218	Ɔ	250	.
27	←	59	;	91	[	123	{	155	Ø	187	¶	219	■	251	¹
28	¬	60	<	92	\	124		156	£	188	¶	220	■	252	³
29	↔	61	=	93	]	125	}	157	Ø	189	¢	221	ı	253	²
30	ˆ	62	>	94	^	126	~	158	×	190	¥	222	İ	254	
31	▼	63	?	95	_	127	Δ	159	f	191	ˆ	223	■	255	



## BIBLIOGRAFIA

H. SCHILDT, *C – The Complete Reference*, 3<sup>rd</sup> edition, McGraw-Hill, 1995.

K. JAMSA, *Microsoft C – Secrets, Shortcuts and Solutions*, Microsoft, 1989.

S. HOLZNER, *C Programming*, Brady, 1991.