

ALFREDO BOENTE

APRENDENDO A PROGRAMAR EM
LINGUAGEM



DO BÁSICO AO AVANÇADO

ABORDA INSTALAÇÃO DO TURBO C

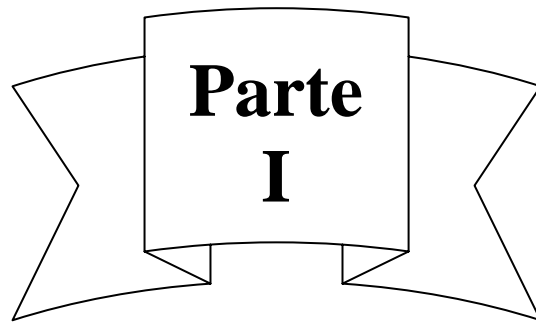
50 EXERCÍCIOS

CRIOGRAFIA

ROTINAS DE ROMBIOS

TABELA ASCII





**Parte
I**



**Conhecendo
A Linguagem C**

1

CONCEITOS BÁSICOS DA LINGUAGEM C

Antes mesmo de falarmos em linguagem C, gostaria de apresentar-lhes o Juan, um passarinho bem simpático que tem por objetivo chamar sua atenção para os momentos de maior importância deste livro.

Olá!
Meu nome é *Juan*. Espero
que vocês gostem deste livro
assim como eu.



A linguagem C, criada por Dennis M. Richie e Ken Thompson no laboratório Bell em 1972, é baseada na linguagem B de Thompson que era uma evolução da antiga linguagem BCPL. A linguagem B recebeu este nome por ser a primeira letra da linguagem BCPL e conseqüentemente, a linguagem C, com a segunda letra desta. Quando fui apresentado inicialmente à Linguagem C, me informaram que a linguagem sucessora da C seria a linguagem P e, por conseguinte, a linguagem L. Bem, depois desse tempo todo, acompanhei o surgimento do C++, C#, Oak, Java, entre outras, com características semelhantes às da linguagem C mas, somente isto. Creio que depois de tanto tempo, e por terem surgido linguagens de extremo poder computacional, as possíveis linguagens P e L ficaram apenas registradas como parte de um conto de estória.

O livro clássico que serve de literatura padrão para todo e qualquer programador C é *C The Programming Language*, escrito por Brian W. **Kernighan** e Dennis M. **Richie**, o qual é indicado como manual técnico do programador C. Equipara-se ao exemplo de que não existe falar de Engenharia de Software sem falar em **Roger Pressman**, grande nome desta cadeira. Bem, mas isto é assunto para outro livro, não é mesmo?

A linguagem C é uma linguagem de programação tão poderosa que se adequa a qualquer tipo de sistema. Só a título de curiosidade, saibam que o sistema operacional UNIX foi escrito em C, por exemplo.

Quanto ao uso de compiladores para a linguagem C, em geral, utiliza-se o TURBO C, da BORLAND, nada tendo contra a utilização de qualquer outro tipo de compilador pertencente a outros fabricantes, é claro.

Existem muitas outras virtudes da linguagem C que você poderá conhecer ao longo do seu aprendizado. Ser um programador C representa ser um profissional de programação de computadores altamente qualificado em detrimento ao saber adquirido por meio de uma linguagem de programação poderosíssima que abre horizontes mil, como a linguagem C.

A ESTRUTURA BÁSICA DE UM PROGRAMA EM LINGUAGEM C

Um programa desenvolvido em linguagem C consiste em uma ou várias “funções”. Os nomes programa e função se confundem em C pelo fato de que a linguagem C não utiliza “comandos” como é de costume em outras linguagens de programação já conhecidas, como por exemplo o PASCAL.

Então podemos afirmar que através de funções predefinidas teremos a possibilidade de criarmos outras funções em nossos programas de computador, ampliando assim a biblioteca padrão da linguagem C.

ELEMENTOS BÁSICOS

Veja a estrutura do menor programa possível na linguagem C:

```
main ( )  
{  
  
}
```

Este programa compõe-se de uma única função chamada main, que significa principal.

```
main ( )    ←   função principal de qualquer programa C
{           ←   inicia o corpo da função (BEGIN)
}           ←   termina a função (END)
```

Logo a função main () deve existir como parte integrante do seu programa pois ele marca o ponto de início da execução de um programa escrito em linguagem C. Quando você tiver que utilizar a função main () sem esperar nenhuma resposta dela (nenhum tipo de retorno) você poderá iniciá-la com o especificador void, da seguinte forma:

```
Void main( )
{
    _____
    _____   corpo da função
    _____
}
```



Exercícios

1. Quem são os criadores da Linguagem C?
2. Explique a origem da Linguagem C.
3. Quais os nomes dos autores do livro clássico C The Programming Language?
4. Para que servem as funções na Linguagem C?
5. Como representamos, na Linguagem C, o Begin e o End, da linguagem de programação Pascal?
6. Como é chamada a função principal de qualquer programa escrito em Linguagem C?
7. Qual é o famoso sistema operacional escrito em Linguagem C?
8. Descreva com suas palavras o que você realmente espera da Linguagem C.

2

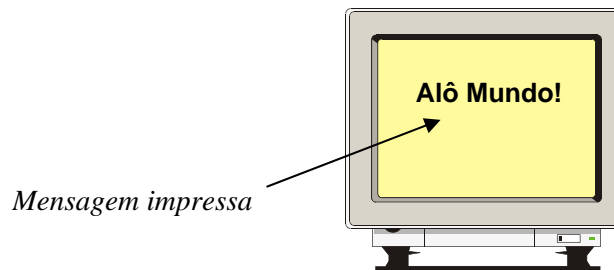
SAÍDA DE DADOS

Na verdade, quando falamos em saída de dados, queremos dizer realmente **saída de informações**, pois, segundo os conceitos básicos de processamento de dados, tudo aquilo que é inserido no computador, através de um INPUT, é dito dado e, conseqüentemente, tudo aquilo que sai, é dito informação.

Observe um exemplo em linguagem C do clássico programa Alô Mundo:

```
/* Representa uma linha de comentários */  
#include <stdio.h>  
  
/* Início do corpo principal do programa */  
main ( )  
{  
  
/* Função utilizada para saída de dados */  
    printf ("Alô Mundo!");  
}
```

Aparecerá então na tela do seu computador a mensagem “Alô Mundo!”, como você pode observar a seguir:



FUNÇÃO DE SAÍDA FORMATADA – printf()

O printf () é uma função utilizada para saída de informações. Sua utilização equipara-se ao comando WRITE da linguagem de programação Pascal. Observe:

```
#include "stdio.h"
main ( )
{
    printf ("Tenho %d anos de idade", 25);
}
```

No nosso exemplo, o código de formatação %d solicita à função printf() imprimir o segundo argumento em formato decimal, ou seja, o número vinte e cinco (25).

O printf () é uma função pertencente à biblioteca padrão de C e pode receber um número de variáveis de argumentos onde cada argumento deve ser separado por uma vírgula. A seguir, outro exemplo:

```
#include <stdio.h>
main ( )
{
    printf ("%s tem %d anos de idade", "Juan", 25);
}
```

No exemplo anterior, utilizamos o código de formatação %s para imprimir uma cadeia de caracteres. Além desse código de formatação a expressão de controle usa

o símbolo `\n`, que representa um código especial que informa à função `printf ()` que o restante da impressão deve ser feito em nova linha. Na verdade, ao representarmos `\n` significa imprimir em uma nova linha.

A linha `#include <stdio.h>` ou `#include "stdio.h"` informa ao compilador que ele deve incluir o arquivo (biblioteca padrão) `stdio.h`. Neste arquivo existem definições de funções de I/O (entrada e saída padrão). Standard Input Output é como devemos efetuar a leitura de `stdio.h`.

```
/* Exemplo – Usando caracter e string */
```

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

```
main ( )
```

```
{
```

```
    printf ("A letra %c ", 'j');
```

```
    printf ("pronuncia-se %s.", "jota");
```

```
}
```

Note que 'j' é um caracter e a palavra "jota" representa uma cadeia de caracteres que, por esse motivo, vem representado por aspas e não por plicas, como é o caso. Ainda, o código de formatação para representação de apenas um caracter é `%c`, conforme podemos observar no exemplo a seguir:

```
/* Exemplo prático */
```

```
#include "stdio.h"
```

```
main( )
```

```
{
```

```
    printf("Olá, eu estou aqui...");
```

```
}
```

Podemos utilizar a função `printf()` através do formatador `%s`, para entrada de dados do tipo `String`. Contudo, se for digitado um nome composto, por exemplo, ANA MARIA, a função interpreta que ANA é uma `String` e MARIA, outra. Bem, a respeito da função para entrada de dados, `scanf()`, veremos um pouco mais tarde. Para evitar esse tipo de problema, mais adiante será estudada a função `gets()`, específica para dados do tipo `String`.



Então, uma `String` para a função `printf()` é terminada quando o caracter espaço em branco é lido por ela. Tecnicamente dizemos que este caracter é nulo (NULL) também, representado pela linguagem de programação C como `'\0'`.

FORMATADORES DA FUNÇÃO

As funções de Entrada e Saída formatadas utilizam por padrão os chamados formadores de tipos de variáveis. É através dele que as funções conseguem expressar seus respectivos valores.

A tabela a seguir mostra os códigos para impressão formatada da função `printf()` e o que eles realmente expressam.

CÓDIGO <code>printf()</code>	FORMATO
<code>%c</code>	APENAS UM CARACTER
<code>%d</code>	DECIMAL INTEIRO
<code>%e</code>	NOTAÇÃO CIENTÍFICA
<code>%f</code>	PONTO FLUTUANTE – float ou double
<code>%g</code>	<code>%e</code> OU <code>%f</code> (O MAIS CURTO)
<code>%o</code>	OCTAL
<code>%s</code>	CADEIA DE CARACTERES – STRING
<code>%u</code>	DECIMAL SEM SINAL
<code>%x</code>	HEXADECIMAL

Exemplo:

```
#include <stdio.h>

void main( )
{
    int num = 12;

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

O especificador de função **void()** quando é utilizado para uma determinada função indica que esta função não retornará valor algum. Logo, em certas literaturas, ao invés de encontrar, por exemplo, a função **main()** escrita dessa forma, você poderá encontrá-la assim: **void main()**.

Na verdade isto quer dizer que ao utilizar a função **main()** sem o especificador **void**, o programador refere-se a esta somente como o corpo principal de um programa em linguagem C.

CARACTERES ESPECIAIS

A tabela a seguir mostra os códigos da linguagem C para caracteres que não podem ser inseridos diretamente pelo teclado.

CÓDIGOS ESPECIAIS	SIGNIFICADO
<code>\n</code>	NOVA LINHA
<code>\t</code>	TAB
<code>\b</code>	RETROCESSO
<code>\"</code>	ASPAS
<code>\\</code>	BARRA INVERSA
<code>\f</code>	SALTA PÁGINA DE FORMULÁRIO
<code>\0</code>	NULO
<code>\x</code>	MOSTRA CARACTER HEXADECIMAL

OUTRAS FUNÇÕES DE SAÍDA

Além da função `printf()`, a linguagem C também disponibiliza outras funções para saída de dados, a função `puts` – específica para strings – e a função `putchar` – específica para caracteres.

Saiba ainda que você tem a possibilidade de ampliar sua biblioteca padrão com até outras funções de saída. Mas isso nós só falaremos melhor mais tarde.

FUNÇÃO `puts()`

Representa a saída de uma única STRING por vez, seguida do caracter de nova linha, ou seja, a função muda de linha automaticamente sem que haja a necessidade do pressionamento da tecla de entrada de dados, ENTER.

Exemplos:

```
puts ("JUAN");  
puts ("Gabriel");  
puts ("Pires Boente");
```

FUNÇÃO `putchar ()`

Representa a saída de apenas um caracter na tela do computador e não acrescenta uma nova linha automaticamente como a função `puts ()`. As duas instruções seguintes são equivalentes:

```
putchar ('c');  
printf ("%c", 'c');
```

```
/* Exemplo Prático */
```

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

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

```
void main( )
```

```
{
```

```
clrscr( );
printf("A primeira letra\n");
puts("do alfabeto é...");
gotoxy(5,4);
putchar('a');
}
```



Foi utilizada a biblioteca <conio.h> que permite ao programador usar a função `clrscr()` para limpar a tela do computador e a função `gotoxy(COL, LIN)` para posicionar o cursor em um ponto de coluna e linha na tela do computador.

A biblioteca <conio.h> significa Console Input/Output.

O programa-exemplo a seguir mostra a exibição de uma certa mensagem redirecionada tanto para a tela do computador quanto para o spool de impressora. As bibliotecas e dispositivos utilizados serão explicados minuciosamente no item Tipos de Redirecionamento, a seguir, neste capítulo:

```
/* Imprime uma mensagem no Vídeo / Impressora */
#include <stdio.h>
#include <conio.h>
#include <dos.h>

main( )
{
    clrscr( );
    system("date");
    printf(stdprn, "MENSAGEM Teste...");
    printf(stdout, "Mensagem Teste...\n\n");
}
```

```
    getch( );  
}
```

Utilizamos a biblioteca **dos.h** cuja função é permitir ao programador a utilização de comandos ligados ao DOS, através da linha de comando de um programa em linguagem C. No nosso exemplo, foi utilizado o comando `date` (usado para exibição e alteração da data do sistema operacional) do DOS.

Como parâmetro especial de funções de I/O formatadas, neste caso, a função `printf()`, foram usados dois tipos de redirecionamentos, **stdout** e **stdprn**. Um redirecionamento também é aplicável para a função `scanf()`, pois esta é uma função de entrada de dados formatada. Observe a tabela a seguir:

TIPOS DE REDIRECIONAMENTOS

stdin	Standard Input Device – Utilizado para teclado
stdout	Standard Output Device – Utilizado para monitor de vídeo
stderr	Standard Error Device – Utilizado para a tela do computador
stsaux	Standard Auxiliary Device – Utilizado para porta serial
stdprn	Standar Printing Device – Utilizado para impressora paralela

O redirecionamento padrão, que já fica implícito na própria especificação da função `printf()`, é o `stdout`, para o monitor de vídeo do seu computador.



Exercícios

1. Para que utilizamos a função `printf()` em nossos programas de computador?
2. Qual a função dos formatadores na função `printf()`?
3. Qual o significado dos seguintes caracteres especiais:
 - a) `\n`
 - b) `\t`

- c) \0
 - d) \x
4. Qual a diferença da função `main()` e `void main()`?
 5. Qual a função de cada formatador especificado abaixo:
 - e) %d
 - f) %c
 - g) %s
 - h) %f
 - i) %u
 6. Qual a função do `puts()`?
 7. Diferencie o uso da função `printf()` e `puts()`.
 8. O que faz a função `putchar('Z')`?
 9. Como posso fazer para receber um mesmo valor (caracter), mostrando o que está sendo digitado ou não?
 10. Comente com suas palavras qual a principal vantagem existente na variação de funções para manipulação de dados do tipo caracter.

3

TIPOS DE DADOS

Os dados são representados pelas informações que serão processadas pelo computador. Estas informações são caracterizadas por dados numéricos, caracteres ou lógicos.

Os chamados dados numéricos podem ser inteiros, número positivo ou negativo, sem o uso de casas decimais ou reais, número positivo ou negativo, com o uso de casas decimais. Como exemplo tem-se: 56, 0, -8, entre outros.

Os dados caracteres podem ser representados por um único caracter ou por um conjunto de caracteres, o qual nomeamos de string. Como exemplo tem-se: “RENATA”, “Ryu Karlus”, “Rua Alfa, nº 24”, “171”, “B”, entre outros.

Os dados que são conhecidos como lógicos são também chamados de dados booleanos por apresentarem apenas dois valores possíveis:

Verdadeiro (true) ou **Falso** (false).

VARIÁVEIS

Uma variável nada mais é do que um endereço de memória que é reservado para armazenar dados do seu programa. Procure utilizar, sempre que possível, as chamadas variáveis significativas, pois seu nome significa, na íntegra, o que elas armazenam (referem-se).

Exemplos:

nome, idade, altura, endereço, data_nasc, salário, cargo etc.

TIPOS DE VARIÁVEIS EM C

Com exceção do tipo *void*, os demais tipos podem vir acompanhados por modificadores do tipo *short*, *long* etc. no momento de sua declaração.

TIPO	BIT	BYTES	ESCALA
char	8	1	-128 a 127
int	16	2	-32768 a 32767
float	32	4	3.4e-38 a 3.4e+38
double	64	8	1.7e-308 a 1.7e+308
void	0	0	sem valor

Cada tipo de dado é associado a uma determinada variável a fim de suprir a necessidade real do programa de computador a ser desenvolvido. Logo, você deve estar bastante atento a este simples porém valioso detalhe.

VARIÁVEL INTEIRA

No exemplo a seguir, iremos demonstrar como usar variáveis do tipo INTEIRA. Um pouco mais tarde teremos acesso a outros tipos de variáveis que serão declaradas como externas.

```
/* Exemplo Prático */
```

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

```
main ( )
```

```
{
```

```
    int num;
```

```
    num = 2;
```

```
    printf ("Este é o numero dois: %d", num);
```

```
}
```

Outro exemplo:

```
#include <stdio.h>
void main ( )
{
    int num1,num2=4;
    num1 = num2;
    printf(“%d \n %d”, num1, num2);
}
```

Utilizamos a palavra reservada `int` para representar o tipo de variável inteira cujo código de formatação utilizado é `%d`. Ainda, foi usado o símbolo de `=` que para a linguagem C é interpretado como atribuição (`:=` no PASCAL).

Em linguagem C, o símbolo `=` representa igualdade e `!=` representa diferença. Tais símbolos serão mostrados mais tarde. Podemos ainda utilizar os especificadores de variáveis `short`, `long` e `unsigned`.

O especificador/modificador de variável `short` representa um valor curto e o `long` um valor longo (o dobro). Observe que `short int` ocupa 2 bytes e `long int` ocupa o dobro, 4 bytes. Já o especificador `unsigned` representa apenas a não utilização de sinais; logo, `unsigned int` também ocupará 2 bytes na memória.

Podemos ainda formatar o tamanho mínimo para impressão do conteúdo de uma determinada variável dependendo somente dos valores que serão estipulados pelo próprio programador de acordo com sua real necessidade.

Veja o próximo exemplo:

```
#include “stdio.h”
main( )
{
    int ano=2002;

    printf(“Feliz %4d.”, ano);      /* A saída será Feliz 2002. */
    printf(“\nFeliz %6d.”, ano);  /* A saída será Feliz 2002. */
}
```

```
        printf("\nFeliz %8d.", ano);    /* A saída será Feliz      2002. */  
    }
```

Caso você queira completar com zeros à esquerda de um valor numérico do tipo inteiro, é possível. Esse recurso também pode ser aplicado à variável inteira com o especificador **unsigned**. Veja nosso próximo exemplo:

```
#include <stdio.h>  
#include <conio.h>  
main( )  
{  
    int a=89;  
    unsigned int b=95;  
  
        printf("%02d", a);    /* A saída será 89 */  
        printf("%04u", b);    /* A saída será 0095 */  
        printf("%06d", a);    /* A saída será 000089 */  
        printf("%08u", b);    /* A saída será 00000095 */  
}
```

Quando estivermos nomeando variáveis para nossos programas, devemos tomar bastante cuidado com os nomes criados para que não venhamos a usar as chamadas **PALAVRAS RESERVADAS** da linguagem C.

Observe algumas dessas palavras reservadas:

auto	extern	sizeof
break	float	static
case	for	struct
char	goto	switch
const	if	typedef
continue	int	union
default	long	unsigned
do	register	void
double	return	volatile
else	short	while
enum	signed	

VARIÁVEL REAL

Como já foi comentado anteriormente, as variáveis reais são aquelas que apresentam o uso de casas decimais (valores fracionários). Em C, podemos utilizar duas categorias de variáveis reais: as variáveis reais de simples precisão, ou precisão simples, e as variáveis reais de dupla precisão, ou precisão dupla.

REAL DE PRECISÃO SIMPLES

Na linguagem C, utiliza-se float para representar uma variável do tipo REAL de precisão simples (ocupa 4 bytes).

```
/* Real de precisão simples */
```

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

```
main ( )
```

```
{
```

```
    float n1, n2;
```

```
    n1=6;
```

```
    n2=5.5;
```

```
    printf ("A soma de %f com %f é %f", n1, n2, n1+n2);  
}
```

REAL DE PRECISÃO DUPLA

Caso você queira usar uma variável real de precisão dupla, ou seja, aquela que ocupa 8 bytes ao invés de 4, utiliza-se, em linguagem C, a palavra reservada `double` no lugar de `float`.

```
/* Real de precisão dupla */  
#include "stdio.h"  
  
main( )  
{  
    double result;  
    int num;  
  
    num = 59;  
    printf ("O resultado é %f", 3.1415 * num );  
}
```

Podemos, ainda, atribuir o recurso de formatar o tamanho mínimo dos campos na saída de variáveis do tipo real de simples ou dupla precisão, com arredondamento do número de casas decimais.

```
/* Exemplo Prático */  
#include <stdio.h>  
  
main( )  
{  
    float salario=1250;  
    double pi=3,1415;
```

```
printf("Salário = %4.2f.", salário);          /* A saída será 1250.00 */  
printf("PI = %1.2f", pi);                   /* A saída será 3.14 */  
}
```

VARIÁVEL CHARACTER

No exemplo a seguir será usada a palavra reservada `char` para representar o tipo de variável caracter (ocupa 1 byte).

/ Exemplo Prático */*

```
#include "stdio.h"
```

```
void main ( )  
{  
    char letra;  
    letra='a';  
    printf ("%c e a primeira letra do alfabeto", letra);  
}
```

VARIÁVEL CADEIA DE CARACTERES

É importante saber que na linguagem C não existe o tipo de variável `STRING`, encontrado, por exemplo, na linguagem de programação Pascal. Caso você queira representar uma cadeia de caracteres (`STRING`), que ocupa `n` bytes na memória, use o seguinte formato:

```
char <nome da variável>[<tamanho>];
```

Por exemplo:

```
char nome[40];  
char telefone[13];  
char endereço[20];
```

```
/* Entrada de dados com String */
#include <stdio.h>
#include <conio.h>

main( )
{
    char nome[30],
        endereço[20],
        cidade[15],
        uf[2],
        tel[13];

    clrscr( );
    puts("Entre com seu nome:");
    gets(nome);
    puts("Seu endereço:");
    gets(endereco);
    puts("Cidade:");
    gets(cidade);
    puts("UF:");
    gets(uf);
    puts("Telefone:");
    gets(tel);
    clrscr( );
    gotoxy(10,5);
    printf("%s %s",nome,tel);
}
```




IMPORTANTE: Logo, é verdadeiro afirmar que uma variável “STRING” é vista como um vetor de caracteres pela linguagem C.

VARIÁVEL PONTEIRO

Uma variável ponteiro ou apontadora é aquela que, ao invés de armazenar um certo conteúdo de dado, guarda em seu espaço de memória o endereço de memória de outra variável que normalmente apresenta um dado a ser manipulado pelo programa.

Na declaração de uma variável ponteiro ou apontadora, devemos utilizar, como prefixo da mesma, um asterístico (*) para que o compilador interprete que esta variável é, realmente, um ponteiro.

Exemplo:

```
/* Utilizando Ponteiro ou Apontadores */
#include <stdio.h>

main( )
{
    int cont,
        * ponteiro; /* Representação de uma variável ponteiro */

    ponteiro = &cont; /* Referencia-se ao endereço da variável cont */
    *ponteiro = 3; /* Armazena indiretamente na variável cont o nº 3 */

    printf("Ponteiro = %d \n Cont = %d", ponteiro, cont);
    getch( ); /* Pausa temporária até que uma tecla seja pressionada */
}
```

Na verdade, quando utilizamos ponteiros ou apontadores, utilizamos a simulação da técnica de gerenciamento de memória denominada, em arquitetura de computadores e sistemas operacionais, como endereçamento indireto.

Outro Exemplo:

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

main( )
{
    int a, *p;

    p = &a;
    *p = 9;

    clrscr( );

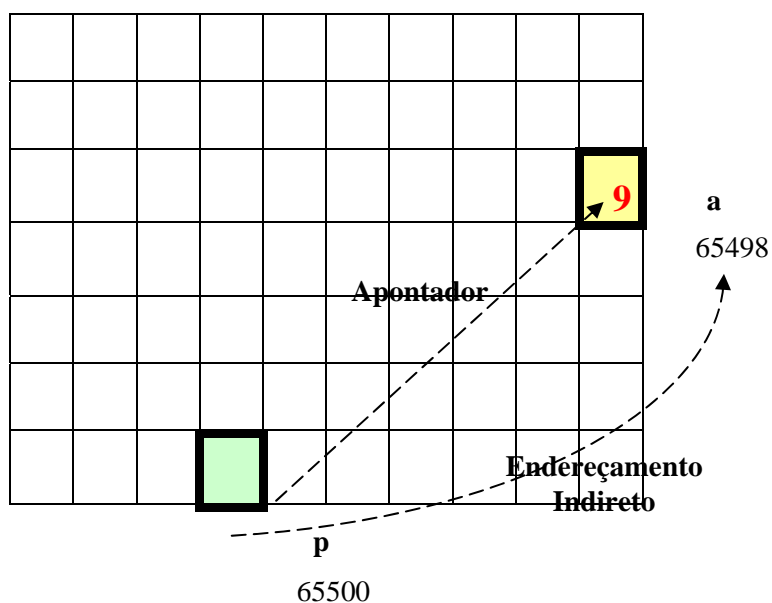
    printf("Endereço de a = %u\n", &a);
    printf("Conteúdo de a = %d\n", a);

    printf("Conteúdo de p = %u\n", p);
    printf("Valor apontado por p = %d\n", *p);
    printf("Endereço de p = %u\n", &p);
    getch( );
}
```

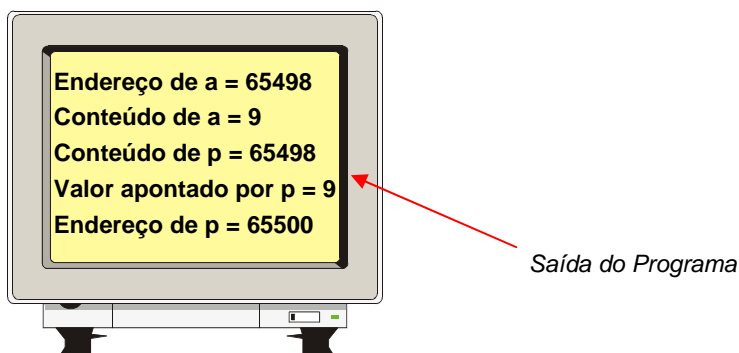
Neste exemplo conseguimos observar que ao trabalharmos com variáveis do tipo ponteiro ou apontadoras podemos ter acesso a três informações úteis sobre elas:

- 1ª. O conteúdo da variável ponteiro;
- 2ª. O valor que está sendo apontado por ela;
- 3ª. O endereço de memória onde está alocada essa variável.

Suponha que as variáveis estejam alocadas aos seguintes endereços de memória:



De acordo com a figura anterior, aparecerá como resultado das saídas provenientes a função `printf()` escritas no programa:



Pois $p = \&a$ indica que o ponteiro p armazenará o endereço de memória da variável a , em questão. Na verdade, quando p faz isso, ele passa a ter o chamado endereçamento indireto da outra variável, aqui denominada como a .

Na linha de comando $*p = 9$ indica que o ponteiro p está servindo como “ponte” para armazenar no conteúdo da variável apontado por ele, a variável a , o número 9. Logo, o ponteiro p aponta para o conteúdo armazenado no endereço de memória da variável a . Maiores detalhes sobre o uso das variáveis apontadoras serão abordados no capítulo 10.

Também foi utilizado o especificador de formato `%u`, que indica a utilização de uma variável dita unsigned, ou seja, sem sinal.

VARIÁVEL SEM SINAL

Uma variável sem sinal é caracterizada em C quando utilizamos o modificador `unsigned` na frente da declaração de uma variável. Por exemplo, `unsigned int num`. Neste, significa dizer que a variável `num` será uma variável do tipo inteira sem sinal.

Então, se for atribuído o valor `-3` para essa variável seu resultado será `3`, pois o sinal de negativo simplesmente será ocultado. Não podemos esquecer de citar que uma variável sem sinal é representada pelo formatador `%u` ao invés de `%d`, no nosso exemplo.

```
/* Utilizando variáveis sem sinal */
#include "stdio.h"

main( )
{
    unsigned int num1;
    int num2;

    printf("Entre com dois números inteiros:");
    scanf("%u %d", &num1, &num2);
}
```

VARIÁVEIS LOCAIS X VARIÁVEIS GLOBAIS

Uma variável local é aquela declarada dentro do corpo de uma certa função e somente pode ser utilizada por aquela função e nenhuma outra mais. Já uma variável global, que poderá ser utilizada por todas as funções existentes em seu programa, é declarada fora, antes do início do corpo da função principal do programa, `main()`. Num programa, podem ser apresentadas tanto variáveis locais quanto variáveis globais.

Observe o exemplo abaixo:

```
/* Variável Local x Variável Global */
#include <stdio.h>
#include <conio.h>
    int a;   /* Variáveis Globais */
    float b;

main( )
{
    int c;   /* Variáveis Locais */
    float d;

    printf("Entre com dois números inteiros:");
    scanf("%d %d", &a, &c);
    printf("\nEntre com dois números reais:");
    scanf("%f %f", &b, &d);
    clrscr( );
    printf("%d %d %2.1f %2.1f", a, c, b, d);
    getch( );
}
```

CONSTANTES

Uma constante representa um valor fixo, constante, durante todo o processamento de um certo programa. Em linguagem C, utilizamos a cláusula de pré-processamento `#define` para declararmos uma constante. Para diferenciarmos as constantes das variáveis, em linguagem C, escrevemos estas constantes com letras maiúsculas.

Exemplo 1:

```
/* Usando constantes em seus programas */
#include <stdio.h>
#include <conio.h>

/* Definição da constante DOIS com o valor numérico 2 */
#define DOIS 2

main( )
{
    float num1, num2, result;

    printf("Entre com dois números:");
    scanf("%f %f", &num1, &num2);
    result = (num1 + num2) / DOIS;

    clrscr( );
    gotoxy(10,10);
    printf("Resultado = %2.1f", result);
    getch( );
}
```

Aqui foi criada uma constante chamada DOIS para representar o numeral 2. Na verdade, a criação de constantes não é feita para ser usada de forma ao acaso, pois tudo aquilo que você cria dentro do seu programa, afinal de contas, ocupa espaço físico na memória do computador.

Exemplo 2:

```
#include <stdio.h>
#define PI 3.14

main( )
{
    float s = 5;

    s += PI;
    printf("s = %2.2f", s);
}
```

No exemplo anterior foi criada uma constante PI para representar o valor da letra grega π . Logo, toda vez que quisermos representar o valor de π utilizaremos a constante PI.

No próximo exemplo será criada a constante ANO para representar o valor 2002 toda vez que o quisermos referenciar no programa. Então, observe, atentamente, o programa-exemplo a seguir:

Exemplo 3:

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

#define ANO 2002

main( )
{
```

```

int anonasc = 1968;

clrscr( );
printf("Idade = %d\n", ANO - anonasc);
getch( );
}

```



Essa vale a pena
prestar um pouco
mais de **ATENÇÃO!**

Para diferenciar as variáveis das constantes, procuraremos, sempre que possível, utilizar letras minúsculas para representar nossas variáveis enquanto utilizaremos letras maiúsculas para representar nossas constantes.

Variável	int numero;
Constante	#define MAX 30

Nos programas citados anteriormente não citamos nenhuma constante do tipo literal. Para termos essa visão acompanhe atentamente as linhas de código do programa apresentado a seguir:

```

#include <stdio.h>
#include <conio.h>
#define EU "Renata Miranda Pires Boente"
main( )
{
    clrscr( );

```



```
printf("Comunico a quem a turma X que todos foram aprovados com
      louvor\n");
printf("Desejo sucesso profissional a todos os formandos de 2002\n\n");
printf("Assinado: %s", EU);
getch( );
}
```

Neste exemplo, foi definido EU como constante cujo valor atribuído é "Renata Miranda Pires Boente". Logo, toda vez que no programa for referenciada a constante EU, na verdade, será referenciado como conteúdo "Renata Miranda Pires Boente".



Exercícios

1. Para que utilizamos uma variável?
2. Cite três nomes de variáveis válidas.
3. Diferencie Variável Local de Variável Global.
4. Como declarar uma variável do tipo ponteiro ou apontadora?
5. Como declarar uma constante num programa em linguagem C?
6. Como podemos utilizar uma variável sem sinal?
7. Quais são os tipos primitivos de variáveis?
8. Diferencie variável real de precisão simples de real de precisão dupla.
9. Quais são as três informações expressas por uma variável do tipo apontadora ou ponteiro?
10. Uma variável String na linguagem C é interpretada por um conjunto de caracteres. Verdadeiro ou Falso?
11. Quais os respectivos tamanhos em bytes da variável real de precisão simples e da variável real de precisão dupla?
12. O que você entende por endereçamento indireto?

4

ENTRADA DE DADOS

Em alguns exemplos, já abordados anteriormente, forçosamente tive que utilizar entrada de dados para representarmos algumas aplicações. Neste momento, iremos abordar as possíveis maneiras de fornecermos dados a um certo programa em linguagem C.

Devemos, na realidade, saber que a linguagem C apresenta dois tipos de entrada de dados: formatada e não-formatada.

FUNÇÃO DE ENTRADA FORMATADA – scanf()

É utilizada para permitir ao usuário realizar uma entrada de dados, geralmente através do teclado. Assim como a função printf () requer a utilização dos códigos de formatação de tipos de variáveis. Observe bem o nosso próximo exemplo:

```
#include "stdio.h"

main ( )
{

    int a, b, soma;
    puts ("Entre com dois numeros inteiros:");
    scanf ("%d %d", &a, &b);
    soma = a + b;
    printf ("Soma = %d \n", soma);
}
```

Na função `scanf()` utiliza-se o `&` (e comercial) antes de cada variável para indicar o endereço de memória no qual o conteúdo da variável estará armazenado. Este é de uso obrigatório quando tratamos variáveis do tipo numérica (inteira ou real).

```
/* Outro exemplo */
```

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

```
main ( )
```

```
{
```

```
    char nome[30];
```

```
    printf ("Digite o seu nome:");
```

```
    scanf ("%s", nome);
```

```
    printf ("Como vai voce %s\n", nome);
```

```
}
```

Observe que não foi utilizado o operador `&` antes da variável `nome`. Isto ocorreu porque em C o nome de um vetor contém o endereço do primeiro byte deste. Assim, não há uma necessidade real de fazê-lo, embora você observe em diversas literaturas a utilização do `&` para esse tipo de variável.

```
scanf ("%s", &nome[0]);
```

Os mesmos formatadores (`%d`, `%f`, `%s`, `%c` etc.) utilizados na função `printf()`, e já estudados, serão utilizados também para a função `scanf()`, por se tratar de entrada e saída de dados formatados.

OUTRAS FUNÇÕES DE ENTRADA

Como já foi comentado anteriormente, o `scanf()` não é a única função utilizada para realizar as entradas de dados de um certo programa. Iremos estudar a entrada de dados não-formatadas através das funções `gets()`, `getchar()`, `getche()` e `getch()`.

FUNÇÃO gets()

Essa função processa tudo que foi digitado até que a tecla ENTER seja pressionada. O caracter ENTER não é acrescentado à string mas sim identificada como término da mesma.

```
/* Exemplo Prático */
```

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

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

```
main ( )
```

```
{
```

```
    char nome[35];
```

```
    clrscr( );
```

```
    printf ("Digite seu nome:");
```

```
    gets(nome);
```

```
    clrscr( );
```

```
    printf ("Como vai %s\n", nome);
```

```
}
```

FUNÇÃO getch()

Em algumas situações não é conveniente o uso do scanf () ou gets (). Essas situações surgem porque em ambas as funções é preciso pressionar a tecla ENTER para sinalizar o fim da leitura. Quando queremos efetuar uma ação parecida para apenas um caracter, devemos utilizar a função getch().

A biblioteca C dispõe de funções que lêem um caracter no instante em que ele é digitado, sem a necessidade do pressionamento da tecla ENTER, o que veremos mais adiante.

```
/* Aproveito a oportunidade para mostrar
```

```
    a forma de representar a utilização de
```

```
comentários com múltiplas linha    */
#include <stdio.h>

main ( )
{
    char ch;

    printf ("Digite uma tecla:");
    ch = getchar ( );

    printf ("\n A tecla que você pressionou
            foi %c.", ch);
}
```

FUNÇÃO getche()

A função `getche()` edita um caracter do teclado e permite que ele seja mostrado na tela do computador. É dispensável o pressionamento da tecla ENTER por parte do usuário. Isso já ocorre de forma automática.

```
#include <stdio.h>
#include <conio.h>
    char ch;

main ( )
{
    printf ("Digite uma tecla:");
    ch = getche( );

    clrscr( );
```

```
    printf (“\n A tecla que você pressionou foi %c.”, ch);  
}
```

Note que a variável **ch** foi declarada como global por estar fisicamente fora do corpo principal do programa.

FUNÇÃO getch()

A função `getch()` permite que o usuário forneça um caracter através do teclado. Este caracter não será mostrado na tela do computador. É também dispensado o pressionamento da tecla ENTER por parte do usuário, pois a passagem para a próxima linha já ocorre automaticamente, como acontece com a função `getche()`. Também, esta função atende à necessidade como recurso de parada temporária da execução do programa, como acontece na linguagem Pascal através do comando `ReadKey`.

```
/* Programa exemplo */  
  
#include <stdio.h>  
#include <conio.h>  
  
main ( )  
{  
    char ch;  
  
    clrscr( );    /* Limpa a tela do computador */  
    printf (“Digite uma tecla.”);  
    ch = getch ( );  
    clrscr( );  
    printf (“\n A tecla que você pressionou foi %c.”, ch);  
    getch( );  
}
```

Logo, podemos afirmar que, ao utilizarmos a função `getchar`, temos a obrigatoriedade de pressionarmos a tecla de entrada de dados para concluirmos a operação, enquanto nas funções `getche` e `getch` não temos a necessidade de pressionar a tecla `enter` (esta operação já é implícita da própria função).



Lembre-se que a escolha da melhor função a ser utilizada no seu programa irá depender exclusivamente da sua decisão, pois quem sabe a real necessidade do seu programa é você mesmo.



Exercícios

1. Por que a função `scanf()` é considerada uma entrada de dados formatada?
2. Podemos usar os mesmos formataadores para a função `scanf()` que são utilizados para a função `printf()`?
3. Qual a diferença entre entrada de dados formatada e entrada de dados não-formatada?
4. Para que serve a função `getchar()`?
5. Qual a diferença entre a função `getche()` e a `getch()`?
6. Para que serve a função `gets()`?
7. Qual a diferença básica na utilização da função `scanf()` e na utilização da função `gets()`?
8. Explique detalhadamente o que irá fazer cada um dos “comandos” abaixo citados:
 - a) `variavel = getchar()`;
 - b) `variavel = getche()`;
 - c) `variavel = getch()`;

5

OPERADORES

Os operadores são utilizados com o objetivo de auxiliar na formação de uma certa expressão. Existem diversos tipos diferentes de operadores. Fazemos então um estudo mais detalhado sobre eles.

OPERADORES ARITMÉTICOS

SÍMBOLOS	OPERADORES
*	Multiplicação
/	Divisão
%	Módulo
+	Adição
-	Subtração

O operador de módulo (resto da divisão inteira) não pode ser aplicado a variáveis do tipo float nem double.

A precedência matemática quanto à utilização de sinais é mantida da mesma forma que na linguagem de programação Pascal, ou seja, (, *, /, +, -. Somente são apresentados de forma diferente os operadores compostos, assim como os operadores de incremento e decremento não estão disponíveis para a linguagem de programação Pascal.

OPERADORES DE INCREMENTO E DECREMENTO

Parece ser bem complicado, à primeira vista, mas não é nenhum bicho de sete cabeças. Incrementar uma variável significa na íntegra que estamos adicionando um valor a ela. Decrementar uma variável é justamente ao contrário.

Veja:

```
i = i + 1;      i ++;      ++ i;
```

Ambas as expressões significam a mesma coisa; incrementar um à variável i. Da mesma forma que as expressões abaixo têm a mesma função.

```
i = i - 1;      i --;      -- i;
```

Ou seja, ambas as expressões decrementam em um a variável i.

PÓS-INCREMENTO E PRÉ-INCREMENTO

```
#include <stdio.h>
main ( )
{
    int a, b;
    a = 2;
    b = a ++;
    printf ("%d %d", a, b);
}
```

```
#include <stdio.h>
main ( )
{
    int a, b;
    a = 2;
    b = ++ a;
    printf ("%d %d", a, b);
}
```

No primeiro caso serão impressos os valores 3 para a variável a e 2 para a variável b. Já no segundo caso, será impresso o valor 3 tanto para a variável a quanto para a variável b.

PÓS-DECREMENTO E PRÉ-DECREMENTO

```
#include <stdio.h>
main ( )
{
    int a, b;
    a = 2;
    b = a --;
    printf ("%d %d", a, b);
}
```

```
#include <stdio.h>
main ( )
{
    int a, b;
    a = 2;
    b = -- a;
    printf ("%d %d", a, b);
}
```

No primeiro caso, serão impressos os valores 1 para a variável a e 2 para a variável b. Já no segundo caso, será impresso o valor 1 tanto para a variável a quanto para a variável b.



Utilizando o recurso do teste chinês, determine quais serão os valores impressos para as variáveis do programa a seguir, no final do processamento.

```
/* Programa Atento pra não errar */  
#include <stdio.h>  
  
main( )  
{  
    int a=1,  
        b=3,  
        c=4,  
        d, e;  
  
    a++;  
    d = --b;  
    c += a;  
    e = a + b * c;  
    --e;  
  
    printf(“%d %d %d %d %d”, a, b, c, d, e);  
}
```

Assim, os valores das variáveis serão os seguintes:

a = 2	b = 2	c = 6
d = 2	e = 13	

OPERADORES DE BITS

Existem na linguagem C os chamados operadores de bits que objetivam, de forma geral, um tratamento lógico. Eles não podem ser aplicados às variáveis do tipo float e double. São eles:

&	E bit-a-bit
 	OU inclusivo bit-a-bit
^	OU exclusivo bit-a-bit
<<	Deslocamento a esquerda
>>	Deslocamento a direita
~	Complemento unário

Geralmente, o operador & é utilizado para mascarar um dado conjunto de bits. Já o operador | é utilizado para fazer ligações de bits. O operador ^ é utilizado com o objetivo de “desligar”, separar, bits ligados através do operador ou inclusivo.

Os operadores de deslocamento << e >> realizam um deslocamento à esquerda e à direita em uma quantidade determinada de bits, expressa em uma certa operação. Logo, $k \ll 3$, desloca a variável k à esquerda de 3 bits, preenchendo esses bits em branco (vagos) com zeros.

O operador unário ~ fornece o chamado complemento de um de um certo número inteiro, isto é, ele realiza a inversão de cada bit 1 em 0 e também ao contrário.

OPERADORES LÓGICOS

Esses operadores são também conhecidos como conectivos lógicos de operação, pois objetivam conectar expressões lógicas que, geralmente, são apresentadas através de comandos de decisão. São eles:

OPERADOR	FUNÇÃO
&&	E lógico
	Ou lógico
!	Não lógico

Modelos:

```
if ( (estado_civil=='S') && (idade>17))  
    printf("To dentro...");
```

```
if ((uf=='R') || (uf=='S') || (uf=='M') || (uf=='E'))  
    printf("Região Sudeste");
```

```
if (!(sexo=='F'))  
    printf("MASCULINO");
```

```
if((ano<1990) && ((idade=20) || (idade=30))  
    printf("Mensagem Enviada...");  
else  
    printf("Mensagem interrompida...\n");
```

OPERADORES RELACIONAIS

SÍMBOLOS	SIGNIFICADO
>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual
==	Igualdade
!=	Diferença
=	Atribuição simples

Existem ainda os chamados operadores de atribuição composta. São aqueles que apresentam mais de uma atribuição ao conteúdo de uma dada variável.



ATRIBUIÇÃO SIMPLES

Trata-se da atribuição simplificada de um certo valor à uma determinada variável através do operador = (igual).

/ Exemplo de Atribuição Simples */*

```
#include <stdio.h>

void main( )
{
    int a, b=4;
    a=b;
    printf(“%d %d”, a, b);
}
```

ATRIBUIÇÃO COMPOSTA

É caracterizada uma atribuição composta quando utilizamos o recurso da linguagem C de colocarmos, em uma certa expressão, dois ou mais operadores.

Exemplo:

a = a + b;	a += b;
a = a - b;	a -= b;
a = a * b;	a *= b;
a = a / b;	a /= b;
a = a % b;	a %= b;

/ Exemplo Prático */*

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

```
main( )
```

```
{
```

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

```
    a=b;
```

```
    b+=a;
```

```
    printf(“%d %d”, a, b);
```

```
}
```

OPERADORES ESPECIAIS DE ENDEREÇO

A linguagem C suporta dois operadores que lidam com endereços: o operador devolve endereço (&) onde a variável estiver armazenada, e o operador indireto (*) que se referencia ao conteúdo apontado de uma certa variável.

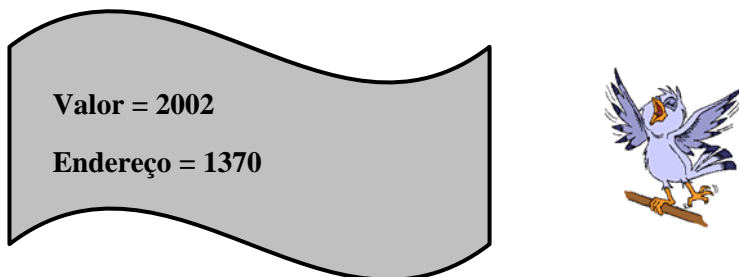
A expressão &variável retorna o endereço do primeiro byte onde a variável está guardada. Assim: *&variávelx*.

A expressão *ptr devolve o conteúdo da posição de memória apontada por ptr. Esse operador será visto com mais detalhes quando estudarmos ponteiros ou apontadores.

```
/* Exemplo do uso de Operadores de Endereço */  
#include <stdio.h>  
  
main ( )  
{  
    int x=2002;  
    printf ("valor = %d \n endereço = %u", x, &x);  
}
```

Um endereço de memória é tratado como um inteiro sem sinal. A saída desse programa varia conforme a máquina e o endereço de memória onde o programa é carregado.

Na realidade, a alocação disponível da memória onde o programa será carregado depende também da quantidade de aplicativos que você já tem aberto e alocado à memória do seu computador. Uma possível saída é:



OPERADOR CONDICIONAL TERNÁRIO

Funciona muito bem para situações de decisões do tipo IF... THEN... ELSE, ou seja, quando em uma determinada condição tem-se que obter duas alternativas possíveis, uma verdadeira e outra falsa. O próximo capítulo trata detalhadamente das estruturas condicionais.

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

```
main ( )
{
    int a,
        b,
        max;

    clrscr( );
    printf ("Digite dois numeros:");
    scanf ("%d %d", &a, &b);
    max = ( a > b ) ? a : b;
    printf ("O maior deles e %d\n", max);
}
```

Observe a seguir que ainda podemos fazer uma representação bem melhor desse comando:

```
#include "stdio.h"
#include "conio.h"

main ( )
{
    int a,
        b;

    clrscr ( );
    printf ("Digite dois numeros:");
    scanf ("%d %d", &a, &b);
    printf ("O maior deles e %d\n", (a > b) ? a : b);
}
```


A forma pela qual você irá trabalhar com o operador condicional ternário, na verdade, não importa muito pois, ao utilizá-lo, você estará eliminando, por exemplo, o uso da função condicional `if()`, que iremos abordar no próximo capítulo.

```
If( a > b )  
printf(“O maior deles e %d\n”, a);  
else  
printf(“O maior deles e %d\n”, b);
```

Você poderá ainda ter uma estrutura ternária encadeada a outra estrutura ternária, assim como também pode acontecer com a estrutura condicional ninho de `if`'s. Veja a seguir:

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

```
void main ( )  
{  
    int a,  
        b,  
        val;  
  
    clrscr( );  
    printf (“Digite dois numeros:”);  
    scanf (“%d %d”, &a, &b);  
    val = ( a == b ) ? a+b : ( a > b ) ? a : b;  
    printf (“O maior deles e %d\n”, val);  
}
```

De repente você perguntaria: “Quando devo utilizar essa estrutura?” Na verdade, não existe uma regra que determine onde e como você deva utilizar o operador condicional ternário ou a função `if()`.

Logo, para a solução de um determinado problema que envolva um tipo de teste lógico condicional cabe especificamente a você decidir “quem” irá servi-lo no momento.



Exercícios

1. Para que servem os operadores?
2. Qual o símbolo que representa o operador de módulo da linguagem C?
3. Qual a diferença básica na utilização do operador de pré-incremento e pós-incremento?
4. O que significa dizer quando falamos que uma variável está sendo decrementada?
5. Qual a função do operador condicional ternário?
6. O que significa atribuição composta? Cite um exemplo prático.
7. O operador de módulo pode ser aplicado à uma variável do tipo real?
8. Explique com suas palavras qual o significado real do comando apresentado:
valor = (a == b) ? a+b : (a > b) ? a : b

6

ESTRUTURAS CONDICIONAIS

Essas estruturas permitem que o programa execute diferentes tipos de procedimentos baseados em uma determinada decisão. Basicamente, existem dois tipos de estruturas condicionais: alternativa simples e alternativa composta.

ALTERNATIVA SIMPLES – if ()

É uma estrutura que, através de uma determinada condição, retorna um valor possível. Equipara-se à estrutura IF...THEN... do Pascal.

```
#include "stdio.h"  
#include "conio.h"
```

```
main ( )  
{  
    int a;  
  
    clrscr( );  
    printf ("Entre com o valor de A:");  
    scanf ("%d", &a);  
  
    if ( a > 0 )  
        printf ("A e maior que ZERO\n");  
}
```

ALTERNATIVA COMPOSTA – if... else

Essa estrutura, diferente da primeira, permite ao usuário retornar dois valores possíveis. O primeiro verdadeiro, se a condição estipulada for satisfeita e o segundo falso, caso a condição não seja devidamente atendida.

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

main ( )
{
    float salario;

    clrscr ( );
    printf (“Entre com seu salario:”);
    scanf (“%f”, &salario);

    if ( salario > 1500 )
        printf (“Voce ganha bem\n”);
    else
        printf (“Voce precisa ganhar mais um pouquinho\n”);
}
```

ENCADEAMENTO DE if’s

Trata-se de um recurso que permite ao usuário utilizar uma estrutura if dentro de outra obtendo, assim, diversas respostas possíveis.

```
/* Uso do ninho de if’s */
#include “stdio.h”

void main ( )
```

```
{
    int num;

    printf ("Entre com um numero:");
    scanf ("%d", &num);

    if ( num = 0 )
        printf ("Numero ZERO");
    else
        if ( num < 0 )
            printf ("Numero Negativo");
        else
            printf ("Numero Positivo");
}
```

Aqui podemos realmente constatar que a função `if()` tem ordem matemática N-1 que significa dizer...

Para cada N respostas que eu precise obter utilizarei N-1 função `if()`.

MÚLTIPLA ESCOLHA – `switch... case...`

A utilização do `switch/case` oferece inúmeras vantagens em relação à utilização da estrutura ninho de `if's`. Um exemplo prático é a facilidade de escrita de uma estrutura composta por múltiplas escolhas, que requer diversas alternativas, a partir de um certo programa de computador.

Observe o comparativo entre o comando ninho de `if's` e o comando `switch`:

```
/* Exemplo Explicativo */
#include <stdio.h>
```

```
void main ( )
{
    char opcao;

    puts ("Entre com uma letra:");
    opcao = getch( );

    if ( opcao == 'A' )
    {
        /* entrada de dados */
        .
        .
        /* para a alternativa */
    }
    else
        if ( opcao == 'B' )
        {
            /* entrada de dados */
            .
            .
            /* para a alternativa */
        }
    else
        if ( opcao == 'C' )
        {
            /* entrada de dados */
            .
            .
        }
}
```

```
        .
        /* para a alternativa */
    }
else
    if ( opcao == 'D' );
    {
        /* entrada de dados */
        .
        .
        /* para a alternativa */
    }
else
    puts ("Opcao Invalida");
}
```

No exemplo mostrado utilizamos o chamado bloco de comandos representado por { e } (funciona como o begin e end; do PASCAL).

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

```
main ( )
```

```
{
```

```
    char opcao;
```

```
    puts ("Entre com uma letra:");
```

```
    opcao = getch( );
```

```
    switch ( opcao )
```

```
    {
```

```
        case 'A' :
```

```
        printf ("Letra A\n");
case 'B' :
        printf ("Letra B\n");
case 'C' :
        printf ("Letra C\n");
case 'D' :
        printf ("Letra D\n");
default :
        printf ("Não e A, B, C nem D\n");
}
}
```

Observação: Na estrutura condicional de múltipla escolha ou estudo de casos, é opcional a utilização da cláusula default para representar uma alternativa que significa “EM NENHUM DOS CASOS ANTERIORES” (refere-se, comparativamente, ao ELSE do comando CASE da linguagem de programação Pascal).



Também, é importante saber que a execução do comando switch segue os seguintes passos:

1. A expressão é avaliada.
2. Se o resultado da expressão for igual a uma constante, então a execução começará a partir do comando associado a essa constante e prossegue com a execução de todos os comandos até o fim do switch, ou até que se encontre uma instrução de parada denominada break.
3. Se o resultado da expressão não for igual a nenhuma das constantes e já estiver sido incluída no comando switch a opção default, o comando associado ao default será executado. Caso contrário, isto é, se a opção default não estiver presente, o processamento continuará a partir do comando seguinte ao switch.

Usando o switch, para fazer o programa anterior, com todas as suas respectivas cláusulas, tem-se:


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

main ( )
{
    char
        opcao;

    clrscr( );
    puts (“Entre com uma letra:”);
    opcao = getch( );
    switch ( opcao )
    {
        case 'A' :
            printf (“Letra A\n”);
            break;
        case 'B' :
            printf (“Letra B\n”);
            break;
        case 'C' :
            printf (“Letra C\n”);
            break;
        case 'D' :
            printf (“Letra D\n”);
            break;
        default :
            printf (“Não e A, B, C nem D\n”);
    }
}
```



Importante: Pode haver uma ou mais instruções seguindo cada case. Essas instruções não são consideradas bloco de comandos logo, não aparecem entre chaves.

A expressão em switch (<expressão>) deve ter um valor compatível com um inteiro, isto é, podem ser usadas expressões do tipo char e int com todas as suas variações. Você não pode usar reais (float e double), ponteiros, strings ou estruturas de dados.

O comando break provoca a saída imediata do switch. Se não existir um comando break seguindo as instruções associadas a um case, o programa prosseguirá executando todas as instruções associadas aos cases a seguir.

CONNECTIVOS DE OPERAÇÃO

Os chamados conectivos de operação são utilizados para permitir que possamos utilizar condições múltiplas ligadas a mesma estrutura condicional. Eles podem ser utilizados da seguinte forma:

CONJUNÇÃO

Representado pelo símbolo &&. Expressa que uma alternativa deve ser satisfeita assim como a outra que acompanha a estrutura do comando utilizado. Na verdade, ambas as alternativas devem ser verdadeiras. Observe no exemplo:

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

main( )
{

    int num1, num2;

    clrscr( );
    printf("Digite dois números:");
```

```
scanf("%d %d", &num1, &num2);

if((num1 == 0) && (num2 == 0))
    printf("Você forneceu apenas o numeral zero...");
else
    printf("%d", a + b);
}
```

DISJUNÇÃO

Representado pelo símbolo ||, expressa que uma alternativa deve ser satisfeita ou a outra que acompanha a estrutura do comando utilizado. Na verdade, basta que uma delas seja verdadeira. Observe no exemplo:

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

main( )
{

    int num1, num2;

    clrscr( );
    printf("Digite dois números:");
    scanf("%d %d", &num1, &num2);

    if((num1 < 0) || (num2 > 0))
        printf("%d é negativo e %d é positivo", num1, num2);
    else
        printf("Sem comentários...\n");
}
```

NEGAÇÃO

Representado pelo símbolo !, expressa exatamente o contrário do valor relacionado na expressão. Observe no exemplo:

```
/* Exemplo Prático */
#include <stdio.h>
#include <conio.h>

main( )
{

    int num;

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

    if(!(num1 != 0))
        printf("%d é igual a zero...", num);
    else
        printf("%d é diferente de zero...",num);
}
```

Atividades



Atividade 1

1. Faça um programa em linguagem C que permita cadastrar dois números inteiros distintos através do teclado. Ao final do processamento, imprima qual o maior e o menor desses números.
2. Escreva um programa em linguagem C que permita ao usuário ler três números distintos pelo teclado, listando, ao final do programa, qual o maior, menor e o mediano deles.
3. Faça um programa em linguagem C que permita ao usuário ler uma letra qualquer através do teclado. No final do processamento, o programa deverá informar se a letra digitada é uma vogal ou uma consoante.
4. Escreva um programa em linguagem C que leia um número, informando ao final do processamento se esse número é primo ou não.



Exercícios

1. Qual a diferença do uso da múltipla escolha para o ninho de if's?
2. O que significa dizer que a função if() apresenta ordem matemática N-1?
3. Explique como funciona a chamada alternativa simples. Exemplifique.
4. Explique como funciona a chamada alternativa composta. Exemplifique.
5. Qual a função do "comando" break?
6. Para que utilizamos, em nossos programas de computador, as chamadas estruturas condicionais?
7. O que quer dizer a seguinte linha de comando:

```
if( a > b )  
    printf("%d", a);
```

```
else
```

```
    printf("%d", b);
```

8. Qual a função dos chamados conectivos de operação?
9. Qual o conectivo de operação que expressa o contrário do que estamos querendo referir?
10. O que faz a seguinte linha de comando:

```
    if( !(sexo == 'M')
```

```
        printf("Feminino...");
```

7

ESTRUTURAS DE ITERAÇÃO

Essas estruturas são utilizadas para que uma parte de seu programa possa ser repetida n vezes sem a necessidade de reescrevê-lo. Também são conhecidas como LOOP ou laços.

Iremos estudar as três estruturas possíveis conhecidas em C: FOR (para/variando), WHILE (enquanto/faça) e DO ... WHILE (repita/até). Vamos analisá-las nessa ordem.

LOOP FOR

É encontrado na maioria das linguagens de programação, incluindo C. No entanto, como vamos ver, a versão C é mais flexível e dispõe de muito mais recursos do que a implementação das outras linguagens.

A idéia básica do comando for é que você execute um conjunto de comandos, um número fixo de vezes, enquanto uma variável de controle é incrementada ou decrementada a cada passagem pelo laço.

Vejamos o exemplo a seguir:

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

```
main ( )
{
    int i;
```

```
clrscr( );  
for ( i = 1; i <= 10; i ++)  
    printf ("%d\n", i);  
}
```

Observe que dentro dos parênteses tem três expressões separadas por ponto-e-vírgula. A primeira expressão é, normalmente, a inicialização da variável de controle do loop for. A expressão dois (central) é um teste que, enquanto o resultado for verdadeiro, reflete em continuação do laço. A terceira expressão (última) é, normalmente, o incremento ou decremento da variável de controle do loop for. Analise bem o próximo exemplo do laço for.

```
/* Exemplo Prático */
```

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

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

No exemplo mostrado é suprimida a representação da inicialização da variável contador dentro do próprio comando, pois o contador *i* já havia sido inicializado anteriormente, no momento de sua declaração.

Observe a seguir outro exemplo da estrutura for tendo como resultado um outro comando for resultando assim em um for dentro de outro for.

```
#include "stdio.h"
```

```
#include "conio.h"
```



```
main ( )
{
    int i, j;

    clrscr( );
    for ( i = 1; i < 10; i ++ )
        for ( j = 1; j < 10; j ++ )
            printf (“\n%d x %d = %d”, i, j, i * j);
}
```

Veja que o laço externo executa 10 vezes, enquanto o laço interno executa 10 vezes para cada passagem do laço externo, totalizando assim, $10 \times 10 = 100$ vezes.

LOOP WHILE

É o mais genérico dos três e pode ser usado para substituir os outros dois; em outras palavras, o laço while supre todas as necessidades. Já os outros dois são usados por uma questão de comodidade. Vamos analisar o exemplo a seguir:

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

main ( )
{
    int x, y;

    x = y = 0;
    while ( y < 10 )
        x += ++y;
    clrscr( );
    printf (“\nx = %d\n y = %d\n”, x, y);
}
```

```
}
```

LOOP DO ... WHILE

O comando do ... while é semelhante ao comando while. A diferença está no momento da avaliação da expressão, o que ocorre sempre após a execução do comando. Isto faz com que o comando do laço do ... while sempre execute pelo menos uma vez antes de realizar tal teste. Observe no exemplo abaixo:

```
#include <stdio.h>

main ( )
{
    int i = 1;

    do
        printf ("%d\n", i);
    while ( ++i <= 10 );
}
```

COMANDOS BREAK, CONTINUE E GOTO

O comando break, quando utilizado em um bloco de comandos, associado a um for, while ou do ... while, faz com que o laço seja imediatamente interrompido, transferindo o processamento para o primeiro comando seguinte do laço. Observe o programa-exemplo a seguir:

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

main ( )
{
    char ch;
```

```
int i;

for ( i = 0; i < 10; i ++ )
{
    ch = getch( );
    if ( ch == '\x1b' ) /* character escape ESC */
        break;
    printf ( "\n%c", ch );
}
puts ( "\nAcabou" );
getch( );
}
```

O comando continue funciona de forma semelhante ao comando break. A diferença reside em que, ao invés de interromper a execução do laço, como o comando break, o comando continue pula as instruções que tiverem abaixo e força a próxima iteração do laço. Observe o programa-exemplo a seguir:

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

main ( )
{
    char
        ch;
    int
        i;

    for ( i = 0; i < 10; i ++ )
    {
```

```
    ch = getch( );
    if ( ch == '\x1b' )
        continue;
    printf (“\n%c”, ch);
}
clrscr( );
puts (“\n Acabou”);
getch( );
}
```

Já o comando goto provoca o desvio da execução do programa para algum outro ponto dentro do código fonte. Como, hoje em dia, todos os programas seguem as técnicas de programação estruturadas, o uso do goto não é recomendável.

Veja um exemplo de simulação do uso do comando goto em um programa escrito em linguagem C:

```
#include <stdio.h>
/* Simulação do uso do comando goto */

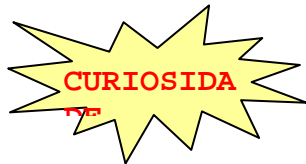
main ( )
{
    .
    .
    .
    while( )
    {
        .
        .
        .
        if (erro )
            goto ERRO;
    }
}
```

```
        .  
        .  
        .  
    }  
    .  
    .  
    .  
}  
ERRO:  
/* representa o rótulo do desvio goto */
```

```
    .  
    .  
    .  
    /* rotina de tratamento de erro */
```

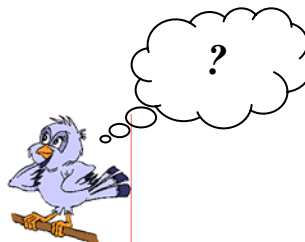
A utilização do goto já foi abolida, há bastante tempo, do dicionário dos bons programadores por causar uma desestruturação no programa que é criado.

Na verdade, quando um programador utiliza goto toda sua programação estruturada desce “privada” abaixo. Em outras palavras: modularização não combina com o uso do goto.



Analise o programa a seguir que mostra um jogo de adivinhação:

```
/* Jogo da Adivinhação */  
#include <stdio.h>  
#include <conio.h>  
#include <stdlib.h>  
#include <time.h>
```



```
main( )
{
    int num,
        secreto;
    char resp = 's';

    while(resp=='s')
    {
        srand(time(NULL));
        secreto = rand( )/100;

        clrscr( );
        printf("Qual é o nº secreto?");
        scanf("%d", &num);
        if(secreto == num)
        {
            printf("Acertou!!!");
            printf("\n o numero e %d", secreto);
        }
        else
            if(secreto < num)
                printf("Errado!! N° muito alto...");
            else
                printf("Errado!! N° muito baixo...");
        getch( );
    }
}
```

```
    printf("Deseja jogar novamente (s/n):");  
    resp = getche( );  
}  
}
```

Neste programa foi utilizada a biblioteca `time.h`, que associa funções de tempo ao seu programa. Também, utilizamos a função `rand()` que retorna um número inteiro aleatório, sorteado internamente na memória do computador, pelo próprio temporizador da linguagem C.



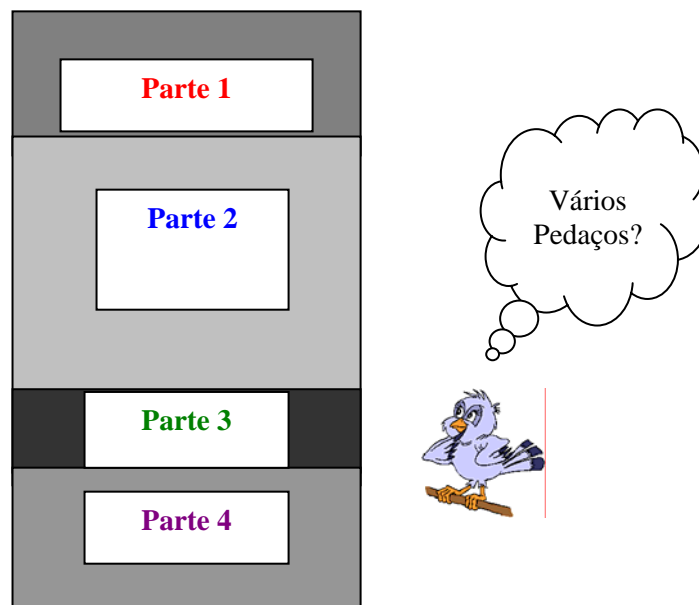
Exercícios

1. Qual o uso prático das chamadas estruturas de iteração?
2. Qual a diferença básica em utilizar a estrutura de repetição `repita...até` ou `enquanto...faça`?
3. Qual a seqüência lógica dos parâmetros atribuídos ao “comando” `for`?
4. Exemplifique o uso da estrutura `para...variando` com um contador incrementado.
5. Exemplifique o uso da estrutura `para...variando` com um contador decrementado.
6. Escreva um programa em linguagem C que escreva todos os números pares compreendidos na seguinte seqüência: 10 até 20. Para tal, utilize a estrutura de repetição...
 - a) `Para...variando`
 - b) `Repita...até`
 - c) `Enquanto...faça`
7. Qual a diferença do comando `break` e `continue`?
8. Por que, hoje em dia, não se utiliza mais com frequência o comando `goto`?
9. Qual a vantagem de elaborarmos um programa cujo teste seja feito no início?
10. Qual a vantagem de elaborarmos um programa cujo teste seja feito no final?

8

FUNÇÕES

As funções na linguagem C funcionam como procedimentos (PROCEDURES) para representarem as sub-rotinas necessárias em um determinado programa. É através do uso de funções que aplicamos a técnica de programação estruturada denominada **MODULARIZAÇÃO**.



Veamos, então, um exemplo prático do uso de funções descrito logo a seguir:

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



```
/* Corpo principal do programa */
main ( )
{

    clrscr( );
    linha ( ); /* realiza a chamada da função */
    printf (“\xDB UM PROGRAMA EM C \xDB\n”);
    linha ( );
    getch( );
}
```

```
/* Declaração da função */
linha ( )
{
    int j;

    for ( j =1; j <= 20; j++ )
        printf (“\xDB”);
    printf (“\n”);
}
```

Exemplo 2:

```
/* Programa Principal */
#include <stdio.h>

main ( )
{
    char ch;
```

```
printf (“Digite 'a' e depois 'b': “);
ch = minusculo ( );
switch ( ch )
{
    case 'a' :
        printf (“\n Voce pressionou 'a.'”);
        break;
    case 'b' :
        printf (“\n Voce pressionou 'b.'”);
        break;
    default :
        printf (“\n Voce não obedeceu o que foi solicitado...”);
}
}

/* Converte para minusculo caso seja digitado uma letra em maiusculo */
minusculo ( )
{
    char ch;
    ch = getche ( );
    if ( ch >= 'A' && ch <= 'Z' )
        ch += 'a' - 'A';
    return ( ch );
}
```

No exemplo anterior, foi utilizado o “comando” return que é usado para retornar um determinado valor a uma dada expressão. Na realidade, ele tem duas funções: primeiro, você pode usar o return() para devolver um valor e retornar, imediatamente, para a próxima instrução do código de chamada. Segundo, usando-o sem os

parênteses, para resultar em uma saída imediata da função na qual ele se encontra; isto é, return fará com que a execução do programa volte para o código de chamada assim que o computador encontrar esse comando, o que ocorre, em geral, antes da última instrução da função.

Exemplo 3:

```
#include <stdio.h>

main ( )
{
    int mins1, mins2;

    printf ("Digite a primeira hora (hora:min): ");
    mins1 = minutos( );
    printf ("Digite a Segunda hora (hora:min): ");
    mins2 = minutos( );
    printf ("A diferenca e %d minutos.", mins2 – mins2);
}

/* função minutos */
minutos( )
{
    int hora, min;

    scanf("%d:%d", &hora, &min);
    return(hora * 60 + min);
}
```

Exemplo 4:

```
#include <stdio.h>

void main ( )
{
    printf ("Luiza\t");
    bar(27);
    printf ("Chris\t");
    bar(41);
    printf ("Regina\t");
    bar(34);
    printf ("Cindy\t");
    bar(22);
    printf ("Harold\t");
    bar(15);
}

/* função gráfico de barras horizontal */
bar(pontos)
    int pontos; /* variável global da função */
{
    int j;      /* variável local da função */

    for ( j=1; j <= pontos; j++ );
        printf ("\xCD");
    printf ("\n");
}
```

Observe que neste último exemplo foram passados determinados números através das funções criadas no programa. Quando isso ocorre dizemos ter uma passagem de parâmetros que pode ser interpretada de duas maneiras distintas: por valor e por referência.

PASSAGEM DE PARÂMETRO POR VALOR E POR REFERÊNCIA

Trata-se da substituição dos chamados parâmetros formais pelos parâmetros reais durante a execução de uma certa sub-rotina. Essa substituição pode ocorrer de duas formas. Por valor, quando o parâmetro passado não tem seu valor alterado durante um certo processamento. Por referência, ocorre quando existe uma alteração do valor do parâmetro real quando o parâmetro formal estiver sendo manipulado por um dado processamento.

```
/* calcula a área da esfera */
#define PI 3.14159

main ( )
{
    float area( );
    float raio;

    printf ("Digite o raio da esfera:");
    scanf ("%f", &raio);
    printf ("A area da esfera e %.2f", area(raio));
}

/* função de cálculo */
float area( r )
    float r;
{
```

```
    return( 4 * PI * r * r );  
}
```

Observe que foi utilizada a diretiva `#define` para definir uma constante simbólica que denominamos `PI`. Logo, a diretiva de pré-processamento `#define` é utilizada em C para definição de constantes de um determinado programa. Também foi utilizado um fixador de número de casas decimais no formatador de variáveis reais `float`, `%f`; então, `%.2f` significa dizer que o valor expresso em reais terá um número de casas decimais fixado em duas.

```
/* Tratamento de Strings */
```

```
#include "stdio.h"
```

```
#include "conio.h"
```

```
#include "string.h"
```

```
char string1[80];      /* Declaração de variável como Global */
```

```
void main( )
```

```
{
```

```
    char string2[80], caract1, caract2;
```

```
    clrscr( );
```

```
    printf("Entre com a string origem:");
```

```
    scanf("%s", &string1);
```

```
    printf("\nEntre com a string resultante:");
```

```
    scanf("%s", &string2);
```

```
    printf("\nCaracter a ser trocado (velho):");
```

```
    caract1=getche( );
```

```
    printf("\nCaracter para mudar (novo):");
```

```
    caract2=getche( );
```

```
    printf("\n\nNº de ocorrências = %d", substitui(string1, string2, caract1, caract2));
```

```
    getch( );
```

```
}
```

```
/* Declaração da função substitui */
int substitui(char s1[80], char s2[80], char c1, char c2)
{
    int i, n=0;
    if(strcmp(s1, s2)==0)
        return(0);
    else{
        for(i=0; i<sizeof(string1); i++)
            if(s1[i]==c1)
            {
                s1[i]=c2;
                n++;
            }
        return(n);
    }
}
```

No programa-exemplo foi utilizada a função `strcmp()` conhecida como `STRING COMPARE` pois, compara o conteúdo do primeiro parâmetro com o segundo. Caso sejam iguais, para o C, tal referência é identificada pela igualdade `ZERO`.

Veja as possíveis variações na utilização da função `strcmp()`:

VARIAÇÕES	SIGNIFICADOS
<code>strcmp(STRING1, STRING2) == 0</code>	STRING1 é igual STRING2
<code>strcmp(STRING1, STRING2) != 0</code>	STRING1 é diferente de STRING2
<code>strcmp(STRING1, STRING2) > 0</code>	STRING1 é maior que STRING2
<code>strcmp(STRING1, STRING2) < 0</code>	STRING1 é menor que STRING2

Existe também a função para tratamento de string, cujo objetivo é realizar cópia de um conteúdo string para outro, `strcpy()` – `STRING COPY`. Veja como realizar tal operação:

```
strcpy(String1, String2);  
strcpy(endereço, "Rua Alfa 32");
```

```
/* Exemplo Prático */  
#include <stdio.h>  
#include <conio.h>  
#include <string.h>  
  
main( )  
{  
    char string1[80], string2[80], velha[80], nova[80];  
    int idade1, idade2;  
  
    clrscr( );  
    printf("Nome da primeira pessoa:");  
    scanf("%s", &string1);  
    printf("\nIdade da primeira pessoa:");  
    scanf("%d", &idade1);  
    printf("\nNome da segunda pessoa:");  
    scanf("%s", &string2);  
    printf("\nIdade da Segunda pessoa:");  
    scanf("%d", &idade2);  
  
    if(idade1==idade2)  
        printf("\n\nAs pessoas têm a mesma idade...");  
    else{
```



```
    if(idade1>idade2)
    {
        strcpy(maior, string1);
        strcpy(menor, string2);
    }
    else{
        strcpy(maior, string2);
        strcpy(menor, string1);
    }
}
```

```
clrscr( );
printf("O nome da pessoa mais velha é %s ", maior);
printf(" e o nome da pessoa mais nova é %s.", menor);
getch( );
}
```

Já que estamos falando em diretivas de pré-processamento, eis a diretiva `#include`. Ela é utilizada para realizar a inclusão de um determinado programa-fonte em outro qualquer. Analise o exemplo a seguir.

Suponhamos que você tenha escrito várias fórmulas matemáticas para calcular áreas de diversas figuras geométricas. Você poderá colocar essas fórmulas em macros em um programa separado.

No instante em que você precisa reescrevê-las para a utilização em seu programa, use a diretiva `#include`.

```
#define PI 3.14159
#define AREA_CIRCULO(raio) (PI * raio * raio)
#define AREA_RETANG(base, altura) (base * altura)
```

```
#define AREA_TRIANG( base, altura ) ( base * altura / 2 )
#define AREA_ELIPSE( raio1, raio2 ) ( PI * raio1 * raio2 )
#define AREA_TRAPEZ( alt, lado1, lado2 ) ( alt * ( lado1 + lado2 ) / 2 )
```

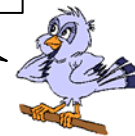
Basta gravar o programa digitado acima como `areas.h`. Recebe esta extensão (`.h`) porque o mesmo irá fazer parte de sua biblioteca padrão.

Quando você for escrever seu programa, simplesmente inclua a seguinte linha de comando: `#include <areas.h>` ou `#include "areas.h"`.

Vamos analisar o seguinte programa:

```
#include "pascal.h"
program
  begin
    write("Isto e linguagem C mesmo ??");
  end
```

Será que estamos realmente falando de
linguagem C?



Parece até Pascal, não é mesmo?!

Bem, o segredo está no arquivo `pascal.h` descrito logo a seguir:

```
#define program  main( )
#define begin  {
#define write( x )  printf( x )
#define end  }
```

Na verdade, o que aconteceu foi uma redefinição de alguns “comandos” da linguagem C mascarando-os de tal forma que ficassem parecidos com os comandos da linguagem de programação Pascal.

Gostou do que você viu, a título de curiosidade? Então, vamos observar, em seguida, como seria o processo inicial da escrita de uma linguagem de programação nos padrões da língua portuguesa (como se fôssemos escrever programas nos modelos de algoritmos).

Suponha algo deste tipo:

```
#include "algoritmo.h"
algoritmo
    inicio
        texto nome;
        escreva("Exemplo de algoritmo:");
        escreva("Entre com seu nome:");
        leia(nome);
        escreva("Ola ", nome);
    fim
```

O segredo está no arquivo algoritmo.h a seguir, conforme já foi apontado por exemplo anterior, na simulação do Pascal, através do arquivo pascal.h:

```
#define algoritmo main( )
#define inicio {
#define texto x string x
#define escreva( x ) printf( x )
#define leia( x ) scanf( x )
#define fim }
```

Então, afirmo que a linguagem C não é simplesmente mais uma linguagem de programação, com base no poder que ela detém. Não esqueça que o sistema operacional UNIX, por exemplo, foi escrito em C.

FUNÇÃO RECURSIVA

Uma função é dita recursiva quando existe dentro de uma certa função uma chamada para ela mesma por diversas vezes. Logo, a técnica de recursividade cria consecutivos “espelhos” para refletir n vezes a chamada da função que está sendo referenciada.

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

main ( )
{
    int num;
    char resp='s';
    long fac( );

    while(resp=='s')
    {
        printf ("\n Digite um numero:");
        scanf ("%d", &num);
        printf ("\nO fatorial de %d e %ld", num, fac(num));
        printf ("Continua (s/n):");
        resp=getche( );
    }
}

/* calcula fatorial – função recursiva */
long fac(n)
    int n;
{
```

```
long resposta;  
  
if (n == 0)  
    return(1);  
resposta = fac( n - 1 ) * n;  
return( resposta );  
}
```

Oba, entendi!

Parece difícil mas, na verdade, é bem fácil de aprender.



Não podemos deixar de mencionar que, através da função recursiva, é criada, na memória do computador, uma pilha que é finita e, conseqüentemente, caso o número de chamadas para a função ultrapassar o limite máximo disponível, a memória alocada para armazenar os valores da pilha irá se esgotar.

Um estudo mais aprofundado a respeito do uso de estrutura de dados, pilhas, para ser mais específico, será abordado no capítulo 13.



Exercícios

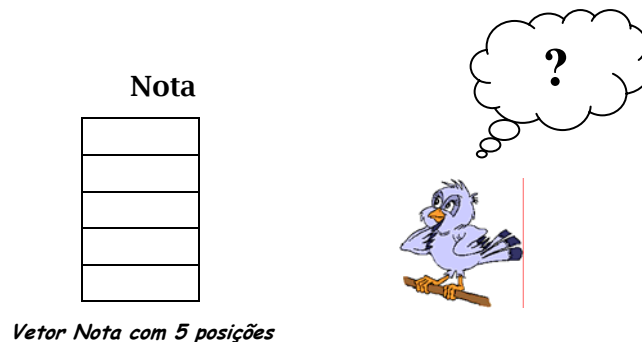
1. O que vem a ser a técnica de Recursividade?
2. Diga o que faz os comandos a seguir:
 - a. `strcmp(s1, s2) == 0`
 - b. `strcpy(nome1, nome2)`
3. Para que serve a passagem de parâmetros?
4. Diferencie passagem de parâmetro por valor e passagem de parâmetros por referência.
5. Defina modularização.

6. Na linguagem C, qual a diferença existente entre procedimentos e funções, caso realmente exista?
7. O que fazem as linhas de comando abaixo:
 - a) `#define Mostra (x) printf(x)`
 - b) `#include "minharotina.h"`
8. Diga o que faz a instrução `return(valor)`.

9

MANIPULAÇÃO COM VETORES

Primeiramente vamos tirar uma dúvida cruel. O que é vetor? Bem, vetor representa um endereço de memória onde serão armazenados diversos dados, de acordo com o dimensionamento dado a ele, na própria definição (criação) da variável vetor. Isso é possível na maioria das linguagens de programação. Um vetor representa então conjuntos indexados de elementos de um mesmo tipo.



Essa forma de estruturação já foi vista em outras linguagens de programação, como, por exemplo na linguagem de programação Pascal, deste modo:

Var

```
TAB : array [ 0..99 ] of Integer;
```

Define uma tabela de 100 elementos inteiros. Veja como é tratada a definição de um vetor do mesmo tipo:

```
int
    TAB[ 100 ];
```

Não podemos esquecer que em C o primeiro elemento é referenciado como TAB[0] e o último como TAB[99].

UNIDIMENSIONAIS

Vejam os então um simples exemplo de aplicação de vetores. Seja o seguinte programa que soma as posições correspondentes de dois vetores: vetor1 e vetor2, cujos elementos são fornecidos pelo usuário.

```
/* programa exemplo de vetor */
#define DIM 5

#include <stdio.h>

/* Corpo principal do programa */
main ( )
{
    int i, vetor1[DIM], vetor2[DIM];

    for ( i=0; i<DIM; i++ )
    {
        printf ("vetor1[%d]=", i);
        scanf ("%d", &vetor1[ i ]);
    }
    for ( i=0; i<DIM; i++ )
    {
        printf ("vetor2[%d]=", i);
        scanf ("%d", &vetor2[ i ]);
    }
}
```



```
for ( i=0; i<DIM; i++ )
    printf (“vetor1[%d] + vetor2[%d] = %d\n”, i, i, vetor1[ i ] + vetor2[ i ]);
}
```

Podemos ainda definir vetores como externos e estáticos cuja inicialização seria uma lista de valores entre chaves e separados por vírgulas.

Veja o exemplo abaixo:

```
static int
    vetor[5] = {0, 1, 2, 3, 4};
```

ou

```
static int
    vetor[ ] = {0, 1, 2, 3, 4};
```

Note que usamos a palavra reservada da linguagem C `static` para especificar que o vetor é um elemento estático. Não esqueçam que tais vetores são chamados de vetores unidimensionais.

COM MAIS DE UMA DIMENSÃO

Observe a seguir a utilização de um vetor com mais de uma dimensão.

```
#define NumeroDeProvas 5
#define MaximoDeAlunos 50
```

```
float
    boletim[ MaximoDeAlunos ] [ NumeroDeProvas ];
```

Como podemos referenciar a terceira nota do 15º aluno da turma?

```
Boletim [15] [2]
```

Veja agora um exemplo da utilização de vetor bidimensional, matriz.

```
/* exemplo de MATRIZ */
#include "stdio.h"

main ( )
{
    static int
        a[3][4] = { {-14, -36, -62, 78},
                    {-77, 14, -92, 17},
                    {67, -51, 18, -60} },
        b[4][2] = { {7, 34},
                    {-23, 69},
                    {32, -1} };

    int
        i, j, k,
        c[3][2];

    for ( i=0; i<3; i++ )
        for ( j=0; j<2; j++ )
        {
            for( c[ i ][ j ] = 0, k = 0; k<4; k++ )
                c[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ];
            printf ("c[%d][%d] = %d\n", i, j, c[ i ][ j ]);
        }
}
```

Exemplo 2:

```
#include <stdio.h>
#define LIN 2
#define COL 2

main ( )
{
    int
        mat[LIN][COL],
        i, j;

    for ( i=1; i<3; i++ )
        for ( j=1; j<3; j++ )
        {
            printf (“\nEntre com o elemento[%d][%d]”, i, j);
            scanf (“%d”, &mat[ i ][ j ]);
        }
    for ( i=1; i<3; i++ )
        for ( j=1; j<3; j++ )
            if ( i == j )
                printf (“\n%d”, mat[ i ][ j ]);
}
```

Imagine que, baseado no exemplo anterior, você também precisasse especificar que os elementos da matriz a serem impressos também deveriam ser números pares. Observe a seguir o uso prático do operador de módulo, já visto no capítulo 5 deste livro:

```
#include <stdio.h>
#define LIN 5
```

```
#define COL 5

void main ( )
{
    int
        mat[LIN][COL],
        i, j;

    for ( i=1; i<=5; i++ )
        for ( j=1; j<=5; j++ )
        {
            printf (“\nEntre com o elemento[%d][%d]”, i, j);
            scanf (“%d”, &mat[ i ][ j ]);
        }

    for ( i=1; i<=5; i++ )
        for ( j=1; j<=5; j++ )
            if (( i == j ) && ((mat[ i ][ j ] % 2) == 0))
                printf (“\n%d”, mat[ i ][ j ]);
}
```

ALGORITMOS DE ORDENAÇÃO

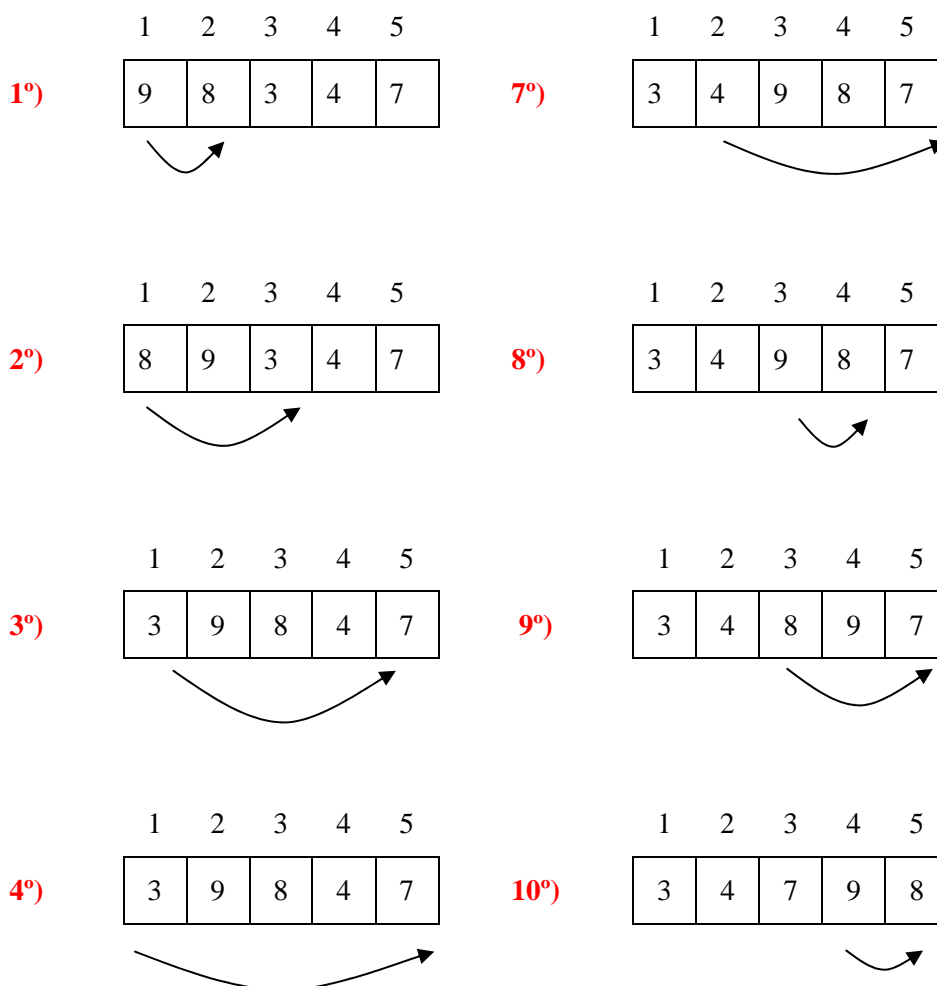
Ordenar ou simplesmente classificar uma tabela consiste em fazer com que seus elementos sejam armazenados de acordo com um critério de classificação determinado.

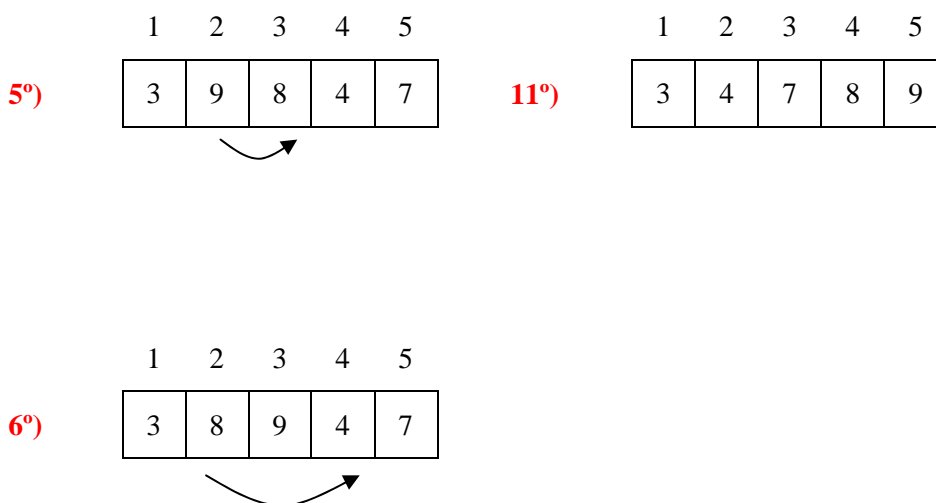
Os ditos critérios de classificação podem variar muito, dependendo do tipo dos elementos que estejam sendo ordenados e, também, do propósito final do programa.

Nos casos mais simples, como os que serão examinados aqui, os elementos da tabela são números inteiros, e os critérios de ordenação se resumem à ordem crescente (ascendente – do menor para o maior) ou decrescente (descendente – do maior para o menor) dos valores expressos na tabela.

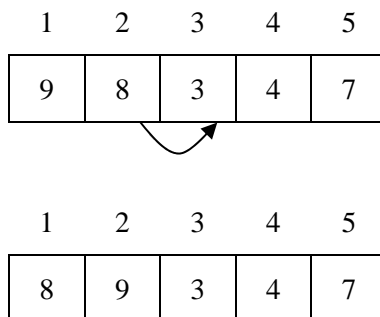
Podemos até trabalhar com tabelas não-ordenadas, contudo, não é aconselhável pois dificultaria um processo de busca dinâmica, como é o caso da pesquisa binária, por exemplo.

A seguir, veremos a ordenação de forma crescente de um vetor numérico do tipo inteiro de 5 elementos.





A estrutura em destaque mostra como ficará o vetor após as comparações e devidas trocas. Quando você consegue acompanhar passo a passo o algoritmo desenvolvido, torna-se muito mais fácil seu processo de aprendizagem de classificação. Trocando em miúdos, a primeira troca. Ou melhor, passando para algoritmo.



Digamos que o vetor chame-se `Numeros`, para realizar esta troca eu irei precisar de uma variável auxiliar, pois, se eu disser que a posição 1 recebe o conteúdo da posição 2, então, tanto a posição 1 quanto a posição 2 irão conter o mesmo valor como mostrado a seguir:

NUMEROS

1	2	3	4	5
8	8	3	4	7

E agora, onde vou buscar o 9 que estava na posição 2?

Por isso precisamos de uma variável auxiliar, vamos chamá-la de Aux, não é sugestivo?

NUMEROS

1	2	3	4	5
9	8	3	4	7

Primeiro, eu guardo a conteúdo da posição 1 na variável auxiliar;

{AUX ← NUMEROS[1]}

AUX

9

Depois, coloco o conteúdo da posição 2 na posição;

{NUMEROS[1] ← NUMEROS[2]}



NUMEROS

1	2	3	4	5
8	8	3	4	7

E por último, coloco na posição 2, o valor da posição 1 que guardei na variável auxiliar. {NUMEROS[2] ← AUX}

NUMEROS

1	2	3	4	5
8	9	3	4	7

Analisando o que ocorre nas trocas, chegamos à seguinte conclusão:

Pegamos a posição	E verificamos com as demais para saber se devemos ou não fazer a troca
1	2, 3, 4, 5
2	3, 4, 5
3	4, 5
4	5

Analisando o quadro acima, podemos chegar à seguinte conclusão:

Quando I for	J será
1	2, 3, 4, 5
2	3, 4, 5
3	4, 5
4	5



Observe que a primeira posição a ser verificada será sempre a primeira subsequente, ou seja, se a posição em questão é I , a primeira subsequente é $I + 1$.

Agora, você precisa saber que existem dois métodos ordenação ou classificação a serem aplicados em seus programas: **Quick-Sort** e **Bubble-Sort**.

Logo a seguir, vamos analisar cada um deles detalhadamente para que possamos então fixar melhor toda essa nova idéia.

MÉTODO DE SELEÇÃO – Quick Sort

Ordem crescente: $\text{tab}[i] \geq \text{tab}[j]$ se $i > j$

Ordem decrescente: $\text{tab}[i] \leq \text{tab}[j]$ se $i < j$

```
/* Exemplo1 – Ordenação Crescente */
#include <stdio.h>

#define N 5

main ( )
{
    int
        ind1, ind2,
        tab[N];

    puts("Entre com os valores da tabela:");

    for ( ind1=0; ind1< N; ind1++ )
        scanf ("%d", &tab[ind1]);
    for ( ind1=0; ind1 < N-1; ind1++ )
    {
        int aux, indmin;

        for ( indmin=ind1, ind2=ind1+1; ind2 < N; ind2++ )
            if ( tab[indmin] > tab[ind2] )
                indmin = ind2;
```

```
    aux = tab[indmin];
    tab[indmin] = tab[ind1];
    tab[ind1] = aux;
}
puts("\nO vetor ordenado e:");

for ( ind1=0; ind1 < N; ind1++ )
    printf ("%d\n", tab[ind1] );
}

/* Exemplo2 – Ordenação Decrescente */
#include <stdio.h>

#define N 10

main ( )
{
    int
        ind1, ind2,
        tab[N];
    puts("Entre com os valores da tabela:");

    for ( ind1=0; ind1 < N; ind1++ )
        scanf ("%d", &tab[ind1]);

    for ( ind1=0; ind1 < N-1; ind1++ )
```



```
{
    int aux, indmin;

    for ( indmin=ind1, ind2=ind1+1; ind2 < N; ind2++ )
        if ( tab[indmin] < tab[ind2] )
            indmin = ind2;
    aux = tab[indmin];
    tab[indmin] = tab[ind1];
    tab[ind1] = aux;
}
puts("\nO vetor ordenado e:");

for ( ind1=0; ind1 < N; ind1++ )
    printf ("%d\n", tab[ind1] );
}
```

A seguir será abordado o método da bolha, tecnicamente denominado “Bubble Sort”. Observe e faça um comparativo entre eles:

MÉTODO DA BOLHA – Bubble Sort

O nome método da bolha deriva-se da maneira com que os maiores valores “afundam” em direção ao fim do vetor, enquanto os menores valores “borbulham” em direção à “superfície” ou topo da tabela.

No programa a seguir usa-se a variável troquei para indicar se durante um passo foi executada alguma troca.

```
/* Exemplo1 – Ordenação Crescente */
#include <stdio.h>
```

```
#define N 10
#define FALSE 0
#define TRUE 1

main ( )
{
    int ind1, ind2,
        tab[N],
        aux,
        troquei;

    puts("Entre com os valores da tabela");

    for ( ind1=0; ind1 < N; ind1++ )
        scanf("%d", &tab[ind1] );
    ind2 = N - 1;

    do
    {
        troquei = FALSE;

        for ( ind1 = 0 ; ind1 < ind2; ind1++ )
            if ( tab[ind1] > tab[ind1 + 1] )
            {
                aux = tab[ind1];
                tab[ind1] = tab[ind1 + 1];
                tab[ind1 + 1] = aux;
            }
    }
}
```



```
        troquei = TRUE;
    }
    ind2--;
} while( troquei );

puts("\nO vetor ordenado e:");

for ( ind1 = 0; ind1 < N; ind1++ )
    printf("%d\n", tab[ind1] );
}

/* Exemplo2 – Ordenação Decrescente */
#include <stdio.h>

#define N 10
#define FALSE 0
#define TRUE 1

main ( )
{
    int ind1, ind2,
        tab[N],
        aux,
        troquei;
    puts("Entre com os valores da tabela");
```

```
for ( ind1=0; ind1 < N; ind1++ )
    scanf(“%d”, &tab[ind1] );
ind2 = N - 1;

do
{
    troquei = FALSE;

    for ( ind1 = 0 ; ind1 < ind2; ind1++ )
        if ( tab[ind1] < tab[ind1 + 1] )
            {
                aux = tab[ind1];
                tab[ind1] = tab[ind1 + 1];
                tab[ind1 + 1] = aux;
                troquei = TRUE;
            }
        ind2--;
    } while( troquei );

puts(“\nO vetor ordenado e:”);

for ( ind1 = 0; ind1 < N; ind1++ )
    printf(“%d\n”, tab[ind1] );
}
```

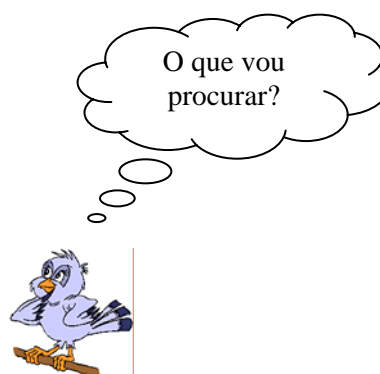
Não podemos esquecer de que quando temos a necessidade de classificar dados de uma certa tabela, não basta trocar apenas o campo-chave utilizado como referência para o processo de ordenação. Devemos, então, trocar de posição todos os campos

pertencentes aos registros da tabela, senão haverá diversos registros com informações misturadas.

ALGORITMOS DE BUSCA

Os algoritmos de busca existem com o objetivo de realizar uma busca dentro de uma determinada tabela, objetivando assim localizar um determinado registro.

Irei abordar e esclarecer a diferença básica existente entre os dois métodos de busca mais utilizados nos programas tradicionais: a busca seqüencial ou pesquisa seqüencial e a busca binária ou pesquisa binária.

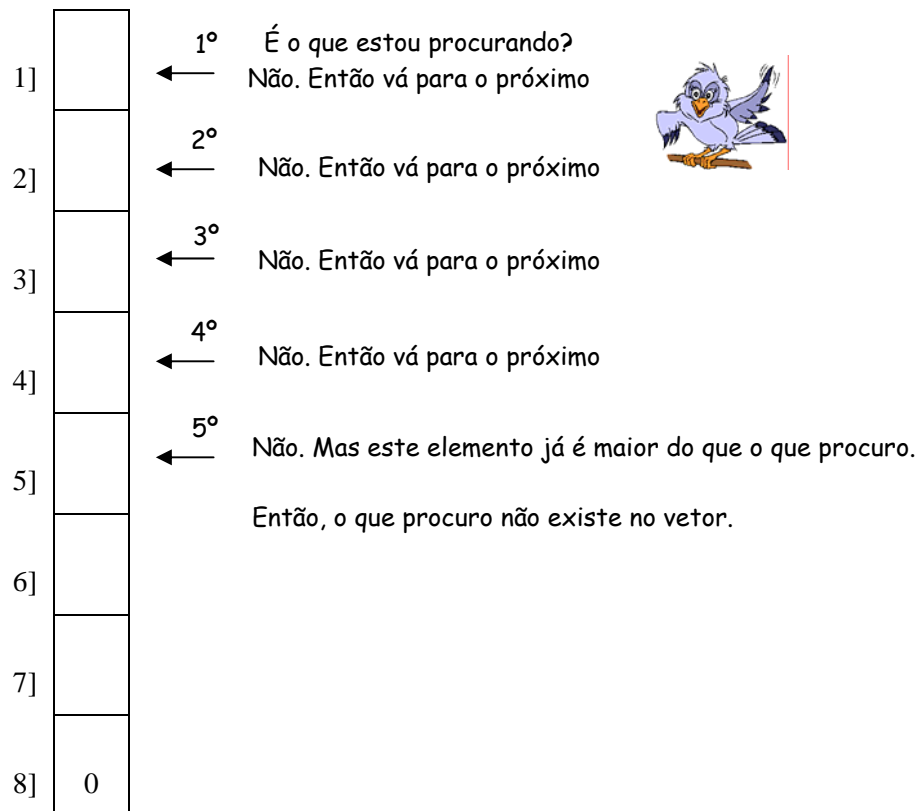


PESQUISA SEQÜENCIAL

Neste método, o processo de busca pesquisa a tabela seqüencialmente, desde o início até seu fim. Cada elemento da tabela é comparado com a chave. Se eles forem iguais, o índice do elemento é retornado e a busca termina. Esse tipo de busca pode ser realizada em tabelas ordenadas ou não. A princípio, a maioria dos programadores de computadores realizam suas operações de busca em tabelas ordenadas.

Vamos então verificar, no exemplo a seguir, a eficiência desse método de busca:

VET

Vamos Procurar pelo número 6

Observe a seguir o código-fonte em linguagem C e como funciona na realidade todo esse processo de busca:

```
/* Exemplo – Pesquisa Sequencial */
```

```
#include "stdio.h"
```

```
#include "conio.h"
```

```
#define N 10
```

```
main ( )
```

```
{
```



```
int
    ind, chave, tab[N];

clrscr( );
puts("Entre com os valores da tabela:");

for ( ind = 0; ind < N; ind++ )
    scanf ("%d", &tab[ind]);

clrscr( );
printf ("Entre com o valor a ser pesquisado:");
scanf ("%d", &chave);

for ( ind = 0; (ind < N) && (tab[ind] != chave); ind++ )
    if (ind < N)
        printf ("A chave foi encontrada na posicao %d\n", ind);
    else
        printf ("Chave nao encontrada\n");
getch( );
}
```

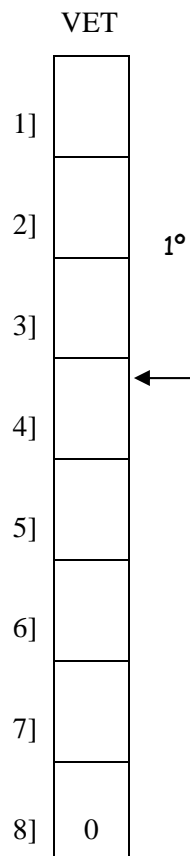
PESQUISA BINÁRIA

Tem por característica reduzir sempre o espaço da tabela a ser procurado pela metade, objetivando, assim, um menor tempo de procura de um determinado elemento dentro da tabela. *Aconselha-se que esse tipo de pesquisa deva ser feito apenas em tabelas ordenadas.*



Vamos Procurar pelo número 7 (Valor existente)

1º Passo



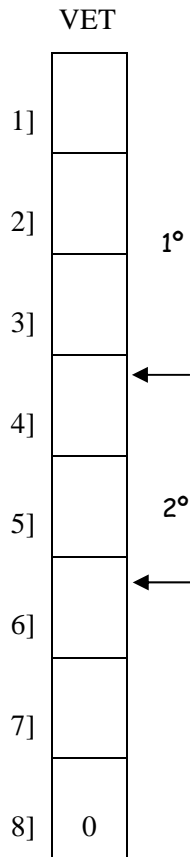
Posição Inicial Atual = 1 e Posição Final Atual = 8

Primeiro devo apontar para o elemento central do vetor. Para calcular esta posição somo a posição inicial com a final e divido por 2, ou seja, $\text{Quociente}(1 + 8, 2)$.

Com isso concluímos que devemos apontar para o elemento de posição 4.

Em seguida, perguntamos se o conteúdo da posição para qual estamos apontando é o que estamos procurando. Como 5 não é o que estamos procurando, a resposta é "NÃO".

Concluímos então que devemos continuar a pesquisa.

2º Passo

A seguir, verifico que o número que estou procurando é maior que o número que estou apontando, logo, concluo que o que procuro só pode estar da posição posterior a que estou apontando até o final do vetor.

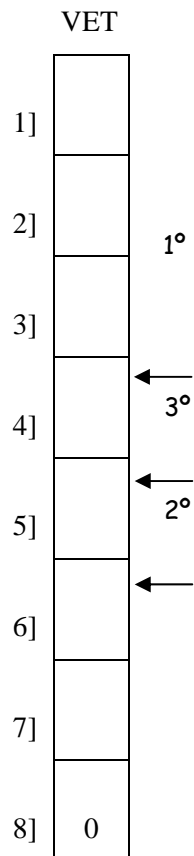
Posição Inicial Atual = 5 e Posição Final Atual = 8

Então pego a minha nova posição inicial, que é a posição que estou apontando mais 1, somo com a posição final que é 8 e divido por 2 para encontrar a minha nova posição central, ou seja, $\text{Quociente}(5 + 8, 2)$.

A seguir aponto para o elemento 6.

Aí faço a célebre pergunta: “É o que estou procurando?”, a resposta é “**NÃO**”.

Então continuaremos a pesquisa.

3º Passo

A seguir, verifico que o número que estou procurando é menor que o número que estou apontando, logo, concluo que o que procuro só pode estar da posição anterior a que estou apontando até a minha atual posição inicial do vetor.

Posição Inicial Atual = 5 e Posição Final Atual = 5

Então pego a minha nova posição final, que é a posição que estou apontando menos 1, somo com a posição inicial que é 5 e divido por 2 para encontrar a minha nova posição central, ou seja, $Quociente(5 + 5, 2)$.

A seguir, aponto para o elemento 5.

Aí faço a célebre pergunta: “É o que estou procurando?”, a resposta é “**SIM**”, a seguir paro a pesquisa.

Entendeu? Então, para facilitar ainda mais o entendimento desse processo, vamos simular a seguir a busca de um determinado valor armazenado no vetor VET.

Vamos procurar pelo número 6 (Valor inexistente).

1º Passo



VET

1]	
2]	
3]	
4]	
5]	
6]	
7]	
8]	0

1º

←

Posição Inicial Atual = 1 e Posição Final Atual = 8

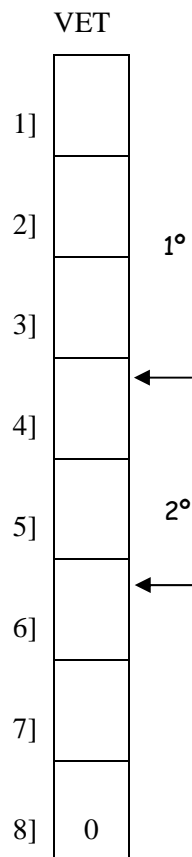
Primeiro devo apontar para o elemento central do vetor, para calcular esta posição como a posição inicial com a final e dividido por 2, ou seja, $\text{Quociente}(1 + 8, 2)$.

Com isto concluímos que devemos apontar para o elemento de posição 4.

Em seguida, perguntamos se o conteúdo da posição para qual estamos apontando é o que estamos procurando. Como 5 não é o que estamos procurando, a resposta é "NÃO".

Concluímos então que devemos continuar a pesquisa.

A posição inicial corresponde à posição dita topo da tabela, enquanto a posição final corresponde à posição de fim da tabela.

2º Passo

A seguir, verifico que o número que estou procurando é maior que o número que estou apontando, logo, concluo que o que procuro só pode estar da posição posterior a que estou apontando até o final do vetor.

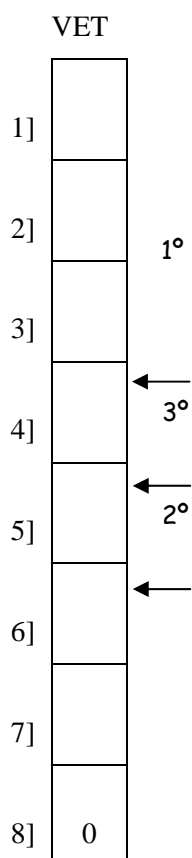
Posição Inicial Atual = 5 e Posição Final Atual = 8

Então pego a minha nova posição inicial, que é a posição que estou apontando, mais 1, somo com a posição final que é 8 e divido por 2 para encontrar a minha nova posição central, ou seja, $\text{Quociente}(5 + 8, 2)$.

A seguir, aponto para o elemento 6.

Aí faço a célebre pergunta: “É o que estou procurando?”, a resposta é “NÃO”.

Então continuaremos a pesquisa.

3º Passo

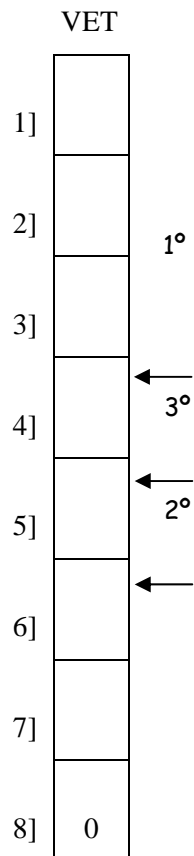
A seguir, verifico que o número que estou procurando é menor que o número que estou apontando, logo, concluo que o que procuro só pode estar da posição anterior a que estou apontando até a minha atual posição inicial do vetor.

Posição Inicial Atual = 5 e Posição Final Atual = 5

Então pego a minha nova posição final que é a posição que estou apontando menos 1, somo com a posição inicial que é 5 e divido por 2 para encontrar a minha nova posição central, ou seja, $Quociente(5 + 5, 2)$.

A seguir aponto para o elemento 5.

Aí faço a celebre pergunta: "É o que estou procurando?", a resposta é "NÃO".

4º Passo

A seguir, verifico que o número que estou procurando é menor que o número que estou apontando, logo, concluo que o que procuro só pode estar da posição anterior a que estou apontando até a minha atual posição inicial do vetor.

Posição Inicial Atual = 5 e Posição Final Atual = 4

Então ele volta a apontar para uma posição que já havíamos visto anteriormente que não poderia estar.

Concluo, então, que toda vez que a posição inicial for maior que a posição final é sinal que o que estamos procurando não existe no vetor.

Segundo a teoria apresentada anteriormente, observe atentamente o próximo exemplo de programa escrito em linguagem C para busca ou pesquisa binária:

```
/* Exemplo – Pesquisa Binária */
```

```
#include "stdio.h"
```

```
#include "conio.h"
```

```
#define N 100
```

```
void main ( )
```

```
{
```



```
int
    ind, chave, tab[N],
    inicio = 0, fim = N - 1,
    meio = (inicio + fim) / 2;

clrscr( );
puts("Entre com os valores da tabela em ordem crescente:");

for ( ind = 0; ind < N; ind++ )
    scanf("%d", &tab[ind]);
printf ("Entre com o valor a ser procurado:");
scanf("%d", &chave);

while ((inicio <= fim) && (tab[meio] != chave))
{
    if (tab[meio] < chave)
        inicio = meio + 1;
    else
        fim = meio - 1;

    meio = (inicio + fim) / 2;

    if (tab[meio] == chave)
        printf ("Chave encontrada na posicao %d\n", meio);
    else
        printf ("Chave nao encontrada\n");
}
```

```
    getch( );  
}
```

Quando você executar esse programa, você será capaz de observar que este só realiza a pesquisa apenas uma vez. É claro, caso você queira realizar a pesquisa mais de uma vez, você deverá criar uma estrutura de repetição vinculada a toda estrutura apresentada do programa com questionamentos do tipo:

DESEJA CONTINUAR?
OUTRA PESQUISA??
NOVAMENTE???

Então, você fica livre para decidir qual o melhor tipo de mecanismo a ser utilizado pelo seu programa. Só não esqueça que a característica primordial da busca binária é realizar uma divisão sucessiva na tabela, reduzindo sempre a mesma em 50% do seu tamanho real. Isto ocorre de forma dinâmica, tornando-a assim bem mais rápida que a busca seqüencial.

Segue um exemplo prático de manipulação de vetores utilizando como base um possível cadastramento de alunos de uma determinada turma de uma certa instituição de ensino, onde serão impressos todos os nomes de alunos que pertencerem a uma turma especial (média aritmética maior ou igual a oito).

```
/* Exemplo prático para fixação */
```

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

```
#define MAX 40
```

```
main( )  
{  
    int matr[MAX], a, b, aux1, troquei;
```

```
char aluno[MAX][30], aux2[30];
float av1[MAX], av2[MAX], med[MAX], aux3;

/* Entrada de Dados */
for (a=0; a<MAX; a++)
{
    clrscr( );
    printf("Matrícula:");
    scanf("%d", &matr[a]);
    printf("\nNome do aluno:");
    scanf("%s", &aluno[a]);
    printf("\n1ª nota:");
    scanf("%f", &av1[a]);
    printf("\n2ª nota:");
    scanf("%f", &av2[a]);
    med[a] = (av1[a] + av2[a]) / 2;
}

/* Ordenação Decrescente */
b = MAX - 1;
do {
    troquei = 0;

    for (a=0, a<b; a++)
    {
        if (matr[a] < matr[a+1])
        {
```

```
    aux1 = matr[a];
    matr[a] = matr[a+1];
    matr[a+1] = aux1;

    strcpy(aux2, aluno[a]);
    strcpy(aluno[a], aluno[a+1]);
    strcpy(aluno[a+1], aux2);

    aux3 = av1[a];
    av1[a] = av1[a+1];
    av1[a+1] = aux3;

    aux3 = av2[a];
    av2[a] = av2[a+1];
    av2[a+1] = aux3;

    aux3 = med[a];
    med[a] = med[a+1];
    med[a+1] = aux3;

    troquei = 1;
}
b--;
} while(troquei);

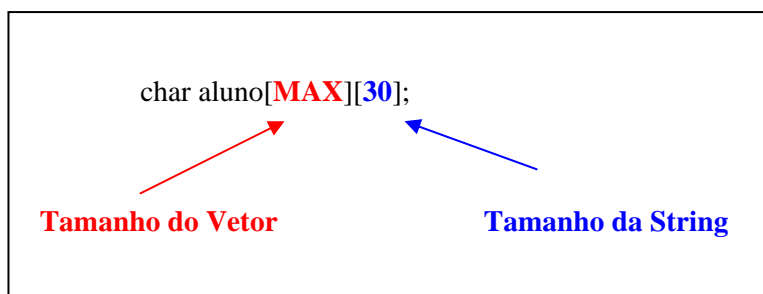
clrscr( );
```

```
    a = 0;

    while( a < MAX )
    {
        if (med[a] > 8 )
            printf("Aluno: %s – Média: %2.1f\n", aluno[a], med[a]);
        a ++;
    }

    getch( );
}
```

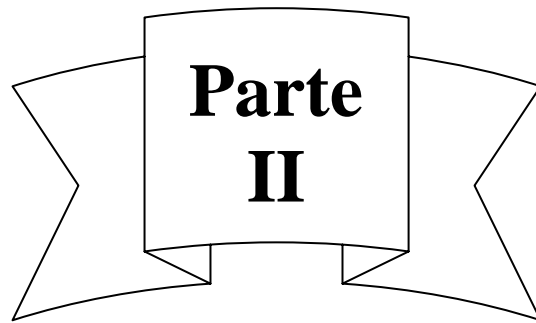
Note bem que ao referenciar uma variável como string, na origem de sua declaração, precisamos inicialmente definir o tamanho do vetor e logo em seguida a quantidade de caracteres que esta variável assumirá. Observe bem a representação a seguir:





Exercícios

1. O que é vetor?
2. Como posso declarar um vetor através da linguagem C?
3. Qual a diferença de vetor unidimensional para um vetor que apresente mais de uma dimensão?
4. O que realmente é uma matriz?
5. Declare uma matriz quadrada de ordem 4 que apresentará elementos do tipo inteiro.
6. Como posso determinar se minha ordenação será crescente ou decrescente?
7. Qual a diferença de Bubble Sort para Quick Sort?
8. Por que é necessário o uso de variáveis auxiliares durante o processo de troca em uma ordenação?
9. Escreva um algoritmo que expresse o princípio da pesquisa seqüencial.
10. Escreva um algoritmo que expresse o princípio da busca binária.



**Parte
II**



**Tópicos Avançados em
Linguagem C**

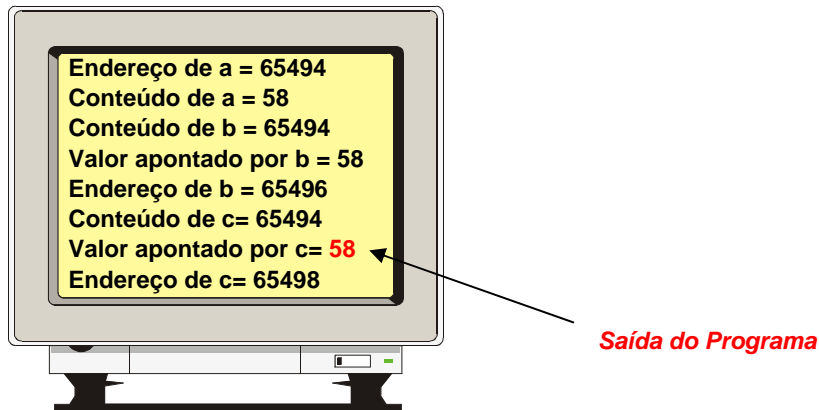
Observe o exemplo abaixo:

```
/* Exemplo – Ponteiro */
#include <stdio.h>
#include <conio.h>

void main ( )
{
    int
        a,
        *b, *c; /* representação de uma variável ponteiro */

    b = &a;
    *b = 58;
    c = b;

    clrscr( );
    printf (“Endereco de a: %u\n”, &a);
    printf (“Conteudo de a: %d\n”, a);
    printf (“Conteudo de b: %u\n”, b);
    printf (“Valor apontado por b: %d\n”, *b);
    printf (“Endereco de b: %u\n”, &b);
    printf (“Conteudo de c: %u\n”, c);
    printf (“Valor apontado por c: %d\n”, *c);
    printf (“Endereco de c: %u\n”, &c);
    getch( );
}
```



ALOCAÇÃO DINÂMICA

Vamos estudar uma variação mais avançada do programa anterior. Através dessa próxima versão trataremos alocação de endereços de memória que tecnicamente é denominada alocação dinâmica.

```
#include <stdio.h>
#include <alloc.h>
main ( )
{
    int
        *ptr;

    ptr = ( int * ) malloc( sizeof( int ) );
    *ptr = 3;
    printf ( "%d\n", *ptr );
}
```

Neste programa não declaramos a variável `i`. Em vez disso, atribuímos à `ptr` um valor retornado por uma função chamada `malloc()`, a qual é declarada em `alloc.h`.

Para que possamos entender a instrução `ptr = (int *) malloc(sizeof(int))` vamos dividi-la em partes:

- a) O operador `sizeof()` devolve o tamanho, em bytes, do tipo ou da expressão entre parênteses.
- b) A função `malloc()` tem o objetivo de retornar um ponteiro para uma área livre de memória a ser identificada por ela.

Assim, a instrução `ptr = (int *) malloc(sizeof(int))` cria dinamicamente uma variável inteira referenciada por `*ptr`. Podemos trabalhar com várias variáveis do tipo ponteiro na memória do computador.

A determinação referente à quantidade necessária para utilização em um determinado programa depende exclusivamente a que o programa se propõe.

ARITMÉTICA COM PONTEIROS

Inicialmente, vejamos o seguinte exemplo:

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

```
main ( )
```

```
{
```

```
    int vetor[3],
```

```
        *p1, *p2;
```

```
    p1 = vetor;
```

```
    p2 = &vetor[2];
```

```
    *p1 = 0;
```

```
    *(p1 + 1) = 1;
```

```
    *( p1 + 2) = 2;
```

```
    if (p2 > p1)
```

```
        printf ("Posicoes: %d\n", p2 - p1);
```

```
}
```

Observe nesse exemplo que podemos usar com variáveis ponteiros os mesmos operadores aritméticos que usamos com variáveis comuns. Isto porque, embora seja uma variável dita ponteiro ou apontadora, simplesmente, não deixa de ser uma variável, a qual tratamos com os já conhecidos operadores.

A única diferença é que em determinados momentos estaremos trabalhando com endereços de memória e não somente conteúdos, que é o que acontece com as variáveis comuns.

```
/* Outro Exemplo */
```

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

```
#define MAX 5
```

```
void main ( )
```

```
{
```

```
    int d;
```

```
    int entra = 0;
```

```
    char nome[40];
```

```
    static char *list[MAX] =
```

```
        { "Katarina",
```

```
          "Nigel",
```

```
          "Gustavo",
```

```
          "Francisco",
```

```
          "Airton"    };
```

```
    printf ("Digite seu nome:"0;
```

```
    gets(nome);
```

```
for ( d = 0; d < MAX; d++ )
    if (strcmp(list[d], nome) == 0)
        entra = 1;
if ( entra == 1 )
    printf (“Voce foi identificado em nosso cadastro”);
else
    printf (“Guardas! Prendam este sujeito!”);
}
```

ARGUMENTOS DA FUNÇÃO `main()`

A linguagem C só permite ao programador utilizar como argumentos da função `main()`, o `argc`, que representa um número inteiro referente à quantidade de parâmetros que poderão ser passados através da função `main()`, e o `argv`, que representa um vetor vazio de ponteiro de caracteres referentes aos possíveis argumentos que serão passados pela linha de comandos.

/ Exemplo Prático */*

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

```
main(argc, argv)
    int argc;
    char *argv[ ];
{
    int contador=0;
    for(;contador<argc;contador++)
        printf(“%s”, argv[contador]);
}
```

Considere que o nome do programa seja **TESTE** e através da linha de comandos foram passados os seguintes valores como argumentos:



```
C:\>TESTE Omega Gama Delta <ENTER>
```

Logo o valor de *argc* será igual a 3 pois, foram passados três argumentos: Omega, Gama e Delta.

Assim, teremos como valores de *argv*:

`argv[0] = Omega`

`argv[1] = Gama`

`argv[2] = Delta`



Exercícios

1. O que você entende por alocação dinâmica?
2. Defina variáveis apontadoras.
3. Quais são os possíveis argumentos da função `main()`?
4. Diferencie `argv` de `argc`.
5. Na aritmética com ponteiros posso efetuar operações com tipos de variáveis diferentes (que não tenham o mesmo tipo)? Justifique sua resposta.
6. Numa alocação dinâmica posso usar vetor de diferentes tipos? Justifique sua resposta.
7. Escreva um programa em linguagem C que permita ao usuário armazenar em uma variável ponteiro valores de forma indireta, ou seja, o conteúdo armazenado deverá ser feito a partir de outra variável, também ponteiro do mesmo tipo.
8. Explique com suas palavras para que serve a função `malloc()`.

11

ESTRUTURAS

Uma estrutura em C é criada através da palavra reservada `struct`. Criar uma estrutura de dados nada mais é do que definir um tipo de dado não existente como padrão (PRIMITIVO) para uma linguagem de programação.

ESTRUTURA SIMPLES

Trata-se de manipular de forma simplificada uma estrutura sem que haja nenhum tipo de encadeamento, o que ocorre nas estruturas compostas (que veremos mais adiante).

/ Exemplo de Estrutura Simples em C */*

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

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

```
struct livro {
```

```
    char titulo[30];
```

```
    int regnum;
```

```
};
```

```
struct livro livro1 = { "Helena", 102};
```

```
struct livro livro2 = { "Iracema", 321};
```



```
main ( )
{
    clrscr( );
    printf (“\n Lista de livros:\n”);
    printf (“ Titulo: %s\n”, livro1.titulo);
    printf (“ Numero do registro: %03d\n”, livro1.regnum);
    printf (“ Titulo: %s\n”, livro2.titulo);
    printf (“ Numero do registro: %03d\n”, livro2.regnum);
    getch( );
}
```

O tratamento de uma estrutura em C funciona da mesma forma que na linguagem Pascal. Para mencionar uma variável da estrutura criada, basta colocar o nome da variável estrutura, um ponto e o campo o qual vai ser trabalhado (livro1.titulo).

ESTRUTURA COMPOSTA

Trata-se do uso de mais de uma estrutura encadeada a outras estruturas. Os itens (campos) dessas estruturas são trabalhados de forma semelhante aos da estrutura simples.

```
/* Exemplo Prático Estrutura Composta em C */
```

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

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

```
struct cliente {
    int codcli;
    char nome[30],
    char tel[15];
    struct dtnasc {
        int dia,
```

```
        mes,  
        ano;  
    };  
};  
  
void main ( )  
{  
    clrscr( );  
    printf (“\n Código do Cliente:”);  
    gets(cliente.codcli);  
    printf (“\n Nome:”);  
    gets(cliente.nome);  
    printf (“\n Telefone:”);  
    gets(cliente.tel);  
    printf (“\n Data de nascimento:”);  
    scanf(“%d %d %d”, &cliente.dtnasc.dia, &cliente.dtnasc.mes,  
        &cliente.dtnasc.ano);  
  
    clrscr( );  
    printf(“Pressione uma tecla para continuar...”);  
    getch( );  
}
```

Podemos ter uma estrutura declarada e com seu nome redefinido através da função da linguagem C **typedef**. Ela permite criar novos nomes de tipos de dados definidos. Veja os exemplos a seguir:

Exemplo 1:

```
typedef int INTEIRO;
```



Indica que INTEIRO, a partir desse momento, serve como sinônimo para a palavra reservada int. Logo, podemos representar da seguinte forma:

```
INTEIRO a, b;  
INTEIRO *fp;  
INTEIRO mat[50];
```

Exemplo 2:

```
struct end {  
    char nome[30];  
    char rua[20];  
    int num;  
    char bairro[15];  
    char cidade[20];  
    char uf[2];  
    char cep[10];  
};  
  
typedef struct end ENDERECO;  
ENDERECO registro;
```

No próximo capítulo iremos estudar as chamadas uniões (unions), realizando assim um comparativo das mesmas com as já vistas estruturas (structs). Será que é muito diferente? Não se trata da mesma coisa??

Então, estrutura é igual a união???



Exercícios

1. O que é e para que serve uma estrutura?
2. Diferencie estrutura simples de estrutura composta.
3. Escreva uma estrutura para compor um registro de alunos de uma turma.
4. Escreva uma estrutura para compor um registro de funcionários de uma certa empresa.
5. Qual a função do typedef?
6. Redefina o nome atribuído a uma certa estrutura denominada Jogadores cuja função é armazenar dados de jogadores de um certo time de futebol de salão.
7. Escreva uma estrutura que seja caracterizada como homogênea.
8. Escreva uma estrutura de estrutura (estrutura composta) que seja caracterizada como homogênea.

12

UNIÕES

Trata-se também de uma Estrutura, ou seja, tanto a União quanto a Estrutura são usadas para armazenar elementos de tipos diferentes em uma mesma variável. A diferença básica em utilizar Estruturas ou Uniões está no fato de que ao declararmos uma certa estrutura é alocado a ela espaço suficiente para armazenar todos os elementos que fazem parte dela de uma só vez, enquanto na união, é alocado um espaço por vez para cada elemento dela.

UNIÃO SIMPLES

É o simples fato de utilizarmos uma única união para concentrar um conjunto de elementos de tipos diferentes. Funciona da mesma forma que a estrutura simples. Observe o próximo exemplo:

```
/* Exemplo Prático */
```

```
#include "stdio.h"
```

```
#include "conio.h"
```

```
union
```

```
{
```

```
    int num1, num2;
```

```
    float num3, num4;
```

```
}    exemp;
```

```
void main( )
{
    exemp.num1 = 2;
    exemp.num2 = 3;
    printf("num1 = %d num2 = %d\n", exemp.num1, exemp.num2);
    exemp.num3 = 5.5;
    exemp.num4 = 9.8;
    printf("num3 = %2.1f num4 = %2.1f\n", exemp.num3, exemp.num4);
    getch( );
}
```

UNIÃO COMPOSTA

Da mesma forma que as estruturas podem associar membros de outras estruturas, as uniões também podem. Contudo, também, podem ser referenciados membros de estruturas e uniões e vice-versa. Observe nosso próximo exemplo:

```
/* Exemplo Prático */
```

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

```
struct alfa
```

```
{
    int n1, n2;
```

```
};
```

```
struct gama
```

```
{
    float n3, n4;
```

```
};
```

```
union
```

```
{
    struct alfa
        a;
    struct gama
        g;
} omega;

void main( )
{
    omega.a.n1 = 2;
    omega.a.n2 = 3;
    omega.g.n3 = 1.5;
    omega.g.n4 = 2.5;

    printf("n1 = %d \n n2 = %d \n", omega.a.n1, omega.a.n2);
    printf("n3 = %2.1f \n n4 = %2.1f\n", omega.g.n3, omega.g.n4);
}
```

Assim, procuraremos utilizar uniões ou estruturas quando quisermos realmente especificar uma estrutura de dados heterogênea qualquer.





Exercícios

1. Podemos ter em um programa de computador elaborado em linguagem C uma estrutura junto com uma união? Justifique sua resposta.
2. O que é e para que serve uma união?
3. Diferencie união simples de união composta.
4. Crie uma união chamada passageiros que tenha por objetivo juntar duas estruturas definidas como classe A e classe B.
5. Diferencie estrutura de união.
6. Escreva uma união para criar uma estrutura composta de clientes de uma certa papelaria. Sabe-se que estes clientes podem ser classificados como Pessoa Física ou Pessoa Jurídica.
7. Posso criar uma união e redefini-la utilizando para isto o typedef?
8. Redefina o nome de uma união chamada Receitas para NotasDaMamãe.
9. Crie uma união realmente homogênea.
10. Escreva uma união de união (união composta).

13

ESTRUTURA DE DADOS

Segundo o professor da COPPE/UFRJ, Jayme Luiz Szwarcfiter, em seu livro *Estruturas de Dados e seus Algoritmos*:

“As estruturas diferem uma das outras pela disposição ou manipulação de seus dados. A disposição de dados em uma estrutura obedece a condições preestabelecidas e caracteriza a estrutura.

O estudo de estrutura de dados não pode ser desvinculado de seus aspectos algoritmos. A escolha certa da estrutura adequada a cada caso depende diretamente do conhecimento de algoritmos para manipular a estrutura de maneira diferente.”

As estruturas de dados que aqui serão representadas através de programas escritos em linguagem C, que objetivam expressar o entendimento concreto do que realmente vem a ser Fila, Pilha, Lista e Árvore, são aquelas de maior difusão desse estudo. Também, será mostrado um exemplo de aplicação prática que manipula listas encadeadas.

USANDO FILAS

Filas nada mais são que estruturas lineares de informação que são acessadas na ordem FIFO (primeiro que entra é o primeiro a sair). O primeiro item colocado na Fila é o primeiro item a ser recuperado, e assim por diante. Uma característica especial da fila é não permitir o acesso randômico a seus dados.

As filas são usadas em muitas situações de programação, tal qual simulações, distribuição de eventos, armazenamento de entrada e saída etc. Considere o seguinte programa-exemplo em linguagem C, para representar o uso de filas, que utiliza as funções *qstore*() – para realizar armazenamento de dados/elementos em uma certa fila, verificando também se a estrutura já está preenchida (cheia), e *qretrieve*() – para realização de leituras de elementos/itens guardados em uma determinada fila.

```
#define MAX_EVENT 100
char *p(MAX_EVENT);
int spos;
int rpos;
qstore(q)
char *q;
{
    if(spos == MAX_EVENT)
    {
        printf("Estrutura Cheia\n");
        return;
    }
    p(spos)=q;
    spos++;
}

qretrieve( )
{
    if(rpos == spos)
        printf("Nao ha eventos para executar\n");
    rpos++;
    return p(rpos-1);
}
```



Essas funções requerem duas variáveis globais: **spos**, que contém a posição da próxima posição livre de armazenamento, e **rpos**, que contém o índice do próximo item a recuperar.

USANDO PILHAS

Uma Pilha é o contrário de uma Fila porque usa o acesso LIFO (o último a entrar é o primeiro a sair). Imagine uma pilha de provas para serem corrigidas por um certo professor. A prova da base da pilha será a última a ser corrigida pelo professor e a prova do topo será a primeira a ser corrigida.

```
/* Exemplo Prático */
```

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

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

```
#include <ctype.h>
```

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

```
int *p,*tos,*bos;
```

```
void push(int i)
```

```
{
```

```
    if (p > bos)
```

```
    {
```

```
        printf("Pilha cheia !\n");
```

```
        return;
```

```
    }
```

```
    *p = i;
```

```
    p++;
```

```
}
```

```
int pop( )
{
    p--;

    if (p < tos)
    {
        printf("Underflow da pilha\n");

        return(0);
    }
    return(*p);
}

void mostraresult(int result)
{
    printf("Resultado e Novo Elemento da Pilha    : %10.2f\n",result);
}

void main( )
{
    int a,b;
    char s[80];

    p = (int*) malloc(100);

    tos = p;
```

```
    bos = p + (100/sizeof(int)) - sizeof(int);

    clrscr( );

    printf("Calculadora com 4 operações básicas (Digite s para Sair)\n");

do
{
    printf("Elemento da Pilha ou Operação (+,-,*,/) : ");
    gets(s);

    *s = toupper(*s);

    switch(*s)
    {
        case '+':

            a = pop();
            b = pop();

            mostrarResult(a + b);
            push(a + b);

            break;

        case '-':
            a = pop();
```

```
        b = pop();

        mostraresult(a - b);
        push(a - b);

        break;

    case '*':

        a = pop();
        b = pop();

        mostraresult(a * b);
        push(a * b);

        break;

    case '/':

        a = pop( );
        b = pop( );

        if (a == 0)
        {
            printf("Infinito !\n");
            break;
        }
```

```
        mostrarResult(b / a);
        push(b / a);

        break;

    default :

        push(atoi(s));
    }
} while(*s != 'S');
}
```

Dessa forma, para implementar uma pilha, você precisa de duas funções: **push()** – que coloca um valor na pilha, e **pop()** – que recupera o valor do topo da pilha.

USANDO LISTAS ENCADEADAS

Filas e Pilhas dividem características comuns. Primeiro, ambas têm regras específicas para referenciar estruturas de dados. Segundo, operações de restauração são por natureza *destrutivas*, isto é, acessar um item em uma pilha ou fila requer sua retirada e, a menos que seja salvo em algum outro lugar, é destruído. Essa visão é associada aos elementos da tabela como forma de manipulação de dados de um vetor.



É verdadeiro que tanto as pilhas como as filas requerem, pelo menos conceitualmente, uma região contínua de memória para operar.

Diferente das pilhas e filas, uma lista encadeada pode acessar seu buffer de modo randômico, porque cada informação carrega consigo um enlace ao próprio item da corrente. Uma lista encadeada requer uma estrutura de dados complexa, ao contrário das pilhas e filas que podem operar tanto com dados simples como complexos. Observe o exemplo a seguir:

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

struct prs{
    char titulo[30];
    char autor[30];
    int regnum;
    double preco;
    struct prs *ptrprox;
};

struct prs *ptrprim, *ptrnovo, *ptratual;

void novonome( )
{
    char numstr[81],prestr[10];

    ptrnovo = (struct prs*) malloc(sizeof(struct prs));

    if (ptrprim == (struct prs*) NULL)
        ptrprim = ptratual = ptrnovo;

    else
    {
        ptratual = ptrprim;
```



```
while (ptratual->ptrprox != (struct prs*) NULL)
    ptratual = ptratual->ptrprox;

ptratual->ptrprox = ptrnovo; // recebe o endereço da nova posição
ptratual = ptrnovo;

}

printf("\n");
printf("\nDigite o título: ");
gets(ptratual->titulo);

printf("\nDigite o autor : ");
gets(ptratual->autor);

printf("\nDigite número de registro: ");
gets(numstr);

ptratual->regnum = atoi(numstr);

printf("\nDigite o preço: ");
gets(prcstr);

ptratual->preco = atof(prcstr);
ptratual->ptrprox = (struct prs*) NULL;
}
```

```
void listatudo( )
{
    if (ptrprim == (struct prs*) NULL)
    {
        printf("\nLista vazia\n");
        return;
    }

    ptratual = ptrprim;

    do
    {
        printf("\n");
        printf("\nTítulo: %s \n",ptratual->titulo);
        printf("\nAutor: %s\n",ptratual->autor);
        printf("\nNúmero do Reg: %03d\n",ptratual->regnum);
        printf("\nPreço: %4.2f\n",ptratual->preco);

        ptratual = ptratual->ptrprox;

    } while(ptratual != (struct prs*) NULL);
}

void main( )
{
    char ptrprim = (char) (struct prs *) NULL;
```

```
char op,ch;

do
{
    clrscr( );

    printf("\nOpções : (N)ovo livro, (L)istar livros, (S)air “);

    op = getch( );
    op = toupper(op);

    switch(op)
    {

        case 'N' :

            novonome( );

            break;

        case 'L' :

            listatudo( );
            printf("\nPressione qualquer tecla !\n");
            ch = getch( );

            break;
```

```
        case 'S' :  
  
            break;  
  
        default :  
  
            puts(“\nDigite somente opções válidas”);  
  
            printf(“\nPressione qualquer tecla !\n”);  
            ch = getch( );  
  
            break;  
    }  
} while (op != 'S');  
}
```

Aqui foram utilizadas as funções `atoi()`, que convertem uma string em valor numérico inteiro, e também a função `atof()`, que converte uma string em valor numérico real.

USANDO ÁRVORES BINÁRIAS

Embora existam muitos tipos de árvores, as árvores binárias são especiais porque, quando são ordenadas, elas se aplicam a buscas velozes, inserções e deleções. Cada item de uma árvore binária consiste em informação com um elo no ramo esquerdo e um no ramo direito.

A tecnologia especial necessária para discutir árvores é um caso clássico de metáforas misturadas. A raiz é o primeiro item da árvore. Cada item (dado) é chamado de nó (ou algumas vezes folhas) da árvore e qualquer parte da árvore de sub-árvore. A altura da árvore é igual ao número de camadas (de profundidade) que as raízes atingem.

As árvores binárias são uma forma especial de lista encadeada. Pode-se inserir, deletar e acessar itens em qualquer ordem.

A ordenação de uma árvore depende exclusivamente de como ela será referenciada. O procedimento de acesso a cada nó é chamado de *árvore transversal*. Observe exemplo:

```
/* Exemplo de Árvore Binária */
```

```
struct tree
```

```
{  
    char info;  
    struct tree *left;  
    struct tree *right;  
} t;
```



```
struct tree *root; /* primeiro nó da árvore */
```

```
void main ( )
```

```
{  
    char s[80];  
    struct tree *stree( );
```

```
    root = 0; /* inicializa a árvore*/
```

```
do
```

```
{  
    printf ("Entre com uma letra:");  
    gets(s);  
    if ( !root )  
        root = stree( root, root, *s );
```

```
        else
            stree( root, root, *s );
    } while( *s );

    printf_tree( root, 0 );
}

struct tree *stree(root, r, info);
struct tree *root;
struct tree *r;
char info;
{
    if ( r == 0 )
    {
        r = malloc(sizeof( t )); /*primeiro nó da sub-árvore */
        if ( r == 0 )
        {
            printf (“Estouro de memoria\n”);
            exit ( 0 );
        }

        r -> left = 0;
        r -> right = 0;
        r -> info = info;

        if ( info < root -> info )
            root -> left = r;
```

```
    else
        root -> right = r;
    return r;
}

if ( info < r -> info )
    stree(r, r -> left, info);
else
    if ( info > r -> info )
        stree(r, r -> right, info);
}

print_tree( r, l )
    struct tree *r;
    int l;
{
    int i;

    if ( r == 0 )
        return;
    print_tree( r -> left, l + 1 );

    for ( i = 0; i < l; ++i )
        printf ( "  ");

    printf ( "%c\n", r -> info);
    print_tree(r -> right, l + 1);
}
```



Importante: Embora uma árvore não precise ser ordenada, na maioria das manipulações que fazemos com ela o ideal seria que a mesma estivesse em determinados casos, pois trata-se de um requerimento obrigatório.



Exercícios

1. Quais as estruturas de dados fundamentais estudadas neste capítulo?
2. O que você entende por pilha?
3. Diferencie lista de fila.
4. O que vem a ser LIFO?
5. Como funciona o processo de interpretação de uma lista encadeada e como isso é tratado na linguagem C?
6. Defina árvore binária. Faça um breve comentário a respeito de sua aplicação.
7. O que vem a ser FIFO?
8. Qual a estrutura que define uma região contínua de memória para operar?
9. Marque Verdadeiro ou Falso:
 - () As árvores não precisam ser ordenadas para serem manipuladas.
 - () Pilhas são mais eficientes que as Filas por apresentar a técnica de FIFO.
 - () Na verdade quando usamos listas encadeadas realizamos uma alocação dinâmica na memória do computador.
 - () A função `qrestore()` pertence ao estudo das listas.
 - () As funções `pop()` e `push()` são integrantes das árvores binárias.
10. O termo ‘o último a entrar é o primeiro a sair’ é referente a que estrutura de dados?

14

TRABALHANDO COM CARACTERES GRÁFICOS

Na linguagem C procura-se representar um caracter gráfico basicamente de duas formas. A primeira, digitando de forma direta, entre aspas, o caracter desejado. Uma outra maneira é representar o caracter através da representação deste em hexadecimal (valor este associado ao caracter pretendido de acordo com padrões estabelecidos na tabela ASCII – veja no apêndice B).

```
/* Mostra o desenho de um carro e uma caminhonete */
```

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

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

```
void main ( )
```

```
{
```

```
clrscr ( );
```

```
printf (“\n\n\n”);
```

```
printf (“\n \xDC\xDC\xDB\xDB\xDB\xDB\xDC\xDC”);
```

```
printf (“\n \xDFO\xDF\xDF\xDF\xDFO\xDF”);
```

```
printf (“\n \n \n”);
```

```
printf (“\n \xDC\xDC\xDB \xDB\xDB\xDB\xDB\xDB\xDB”);
```

```
printf (“\n \xDFO\xDF\xDF\xDF\xDF\xDFOO\xDF”);
```

```
printf (“\n \n \n”);
```

```
    getchar( ); /* Faz uma pausa até que uma tecla seja pressionada */  
}
```

Outro exemplo gráfico:

```
/* Mostra o desenho de uma caixa quadrada – moldura*/  
#include "stdio.h"  
#include "conio.h"  
  
void main ( )  
{  
    clrscr ( );  
    printf ("\xC9\xCD\xCD\xCD\xCD\xBB\n");  
    printf ("\xBA \xBA\n");  
    printf ("\xC8\xCD\xCD\xCD\xCD\xBC\n");  
    getchar( ); /* Realiza uma pausa temporária */  
}
```

Mais um exemplo:

```
#include <stdio.h>  
#include <conio.h>  
  
void main( )  
{  
    espacos(30);  
    linha( );  
    espacos(30);  
    printf("\xDB UM PROGRAMA EM C \xDB\n");  
}
```

```
    espacos(30);
    linha( );
}

linha( )
{
    int j;
    for(j=1; j<=20; j++)
        printf(“\xDB”);
    printf(“\n”);
}

espacos( x )
    int x;
{
    int i;
    for(i=0; i<x; i++)
        printf(“ ”);
    getch( );
}
```

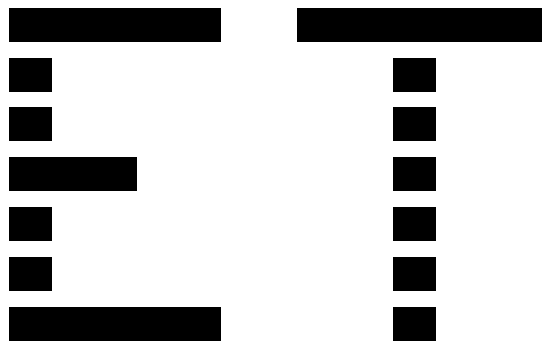
Como já tínhamos visto no início do estudo da linguagem C, o caracter especial `\X` permite ao programador representar um caracter (símbolo) em hexadecimal. Logo, se esse símbolo for um caracter gráfico, o mesmo será então representado.



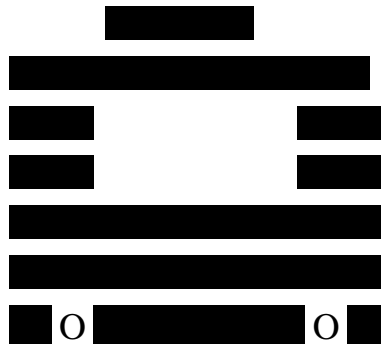


Exercícios

1. Escreva um programa em linguagem C que, manipulando os chamados caracteres gráficos, tenha por objetivo gerar a seguinte figura:



2. Gere o seguinte desenho através de um programa escrito em linguagem C que realize a manipulação de caracteres gráficos.



3. Como podemos representar em linguagem C um caracter da tabela ASCII em hexadecimal?

15

OPERAÇÕES COM ARQUIVOS

Agora vamos aprender como realizar leituras e gravações em disco com a linguagem C. É importante saber que C trabalha com duas formas de manipulação de arquivos (ARQUIVOS TEXTOS e ARQUIVOS BINÁRIOS).

Vamos abordar inicialmente os arquivos textos e, logo em seguida, os arquivos binários.

ARQUIVO TEXTO

Um arquivo aberto de modo texto é interpretado pela linguagem C como uma seqüência de caracteres agrupados em linha. As linhas são separadas por um único caracter chamado caracter de nova linha ou LF. Posso afirmar que esse tipo de arquivo é utilizado principalmente quando desejamos armazenar uma carta redigida, pelo processo de gravação.

Na linguagem C, um arquivo texto é tecnicamente denominado arquivo bufferizado.

Observe como se comporta um arquivo texto, na linguagem C, e quais são as funções (comandos) utilizadas para realizar tal operação:

```
/* Escreve um caracter por vez no arquivo texto */
```

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

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

```
void main( )
```

```
{
```

```

FILE *fp;
char ch;

fp=fopen("arqtext.txt", "w");

while((ch=getche( )) != '\x1b') /* \x1b representa a tecla ESC */
    putc(ch, fp);
fclose(fp);

printf("Gravação efetuada com sucesso!");
getch( );
}

```

Algumas funções e palavras reservadas utilizadas na criação de arquivos:

FILE – Determina que uma variável ponteiro é associada ao arquivo físico externo (cria um ponteiro para a estrutura arquivo – uso obrigatório na linguagem C).

fopen – Realiza a abertura de um arquivo em linguagem C.

putc – Grava um caracter por vez num arquivo texto.

fclose – Usado para fechar um arquivo aberto na memória do computador pela função fopen().

A seguir é apresentada uma lista completa com as opções possíveis de abertura de arquivo texto para a função fopen():

“r”	Abre um arquivo texto para leitura. Nesse tipo de leitura o arquivo deverá estar presente fisicamente no disco.
“w”	Abre um arquivo texto para gravação de dados. Se o arquivo referenciado já existir ele será sobreposto, caso contrário, será criado um novo.
“a”	Abre um arquivo texto para gravação. Os dados inseridos serão adicionados no final do arquivo, mantendo assim, os dados já gravados anteriormente.

“r+”	Abre um arquivo texto para leitura e gravação. O arquivo referenciado deve existir para ser atualizado.
“w+”	Abre um arquivo texto para leitura e gravação. Caso o arquivo já exista o mesmo será sobreposto. Senão, o mesmo será criado.
“a+”	Abre um arquivo texto para atualização e para adicionar dados no final do arquivo existente ou um novo arquivo será criado.

Observe a seguir a codificação de um arquivo texto criado para efetuar a leitura de dados já gravados (lê o arquivo texto anterior criado):

```
/* Lê um caracter por vez do arquivo texto */
#include <stdio.h>
#include <conio.h>

void main( )
{

    FILE *fp;
    int ch;

    fp=fopen("arqtext.txt", "r");
    clrscr( );

    while((ch=getc(fp)) != EOF)
        printf("%c", ch);
    fclose(fp);
}
```

Algumas funções e palavras reservadas utilizadas na criação de arquivos:

EOF – Determina o final do arquivo (End Of File).

getc – Realiza a leitura de um caracter por vez de um arquivo texto já criado.

Outro exemplo:

```
/* Um arquivo texto que grava uma string digitada */
#include <stdio.h>
#include <conio.h>

main( )
{

    FILE *fp;
    char string[81];

    clrscr( );
    fp=fopen("arqtext.txt", "w");
    while(strlen(gets(string))>0)
    {
        fputs(string, fp);
        fputs("\n", fp);
    }
    fclose(fp);
}
```

Algumas funções utilizadas:

strlen() – Retorna um número inteiro relacionado à quantidade de caracteres contidos numa certa string.

fputs – Realiza a gravação de uma linha (string) digitada através do teclado.

```
/* Leitura de uma linha inteira (string) de um arquivo texto */
#include <stdio.h>
#include <conio.h>
```



```
main( )
{

    FILE *fp;
    char string[81];

    clrscr( );
    fp=fopen("arqtext.txt", "r")
    while(fgets(string, 80, fp) != NULL)
        printf("%s", string);
    fclose(fp);
}
```

Foi utilizada a função **fgets()** que, por padrão, assume três argumentos em sua sintaxe de utilização: o primeiro é um ponteiro para o buffer onde será armazenada a linha lida; o segundo indica a quantidade máxima de caracteres a serem lidos e o terceiro argumento é um ponteiro para a estrutura arquivo criada (FILE).



ARQUIVO BINÁRIO

A manipulação feita com arquivos binários é conhecida como arquivos não bufferizados ou baixo nível pois obriga o programador a criar e a manter o buffer de dados para as manipulações de leitura e gravação.

LENDO ARQUIVOS

```
#include "fcntl.h" /* necessário para trabalhar com oflags */
#define TAMBUFF 512
```

```
char buff[TAMBUFF];
```

```
main ( argc, argv ) /*São os únicos parâmetros possíveis da função main */

    /* Declaradas como Globais */
    int argc;
    char *argv[ ];
{
    int i, bytes, j;

    if ( argc != 2)
    {
        printf ("Formato: C:\>lenb arq.xxx");
        exit( );
    }

    if (( i = open(argv[1], O_RDONLY | O_BINARY )) < 0)
    {
        clrscr( )
        printf ("Não posso abrir %s.", argv[1]);
        exit( );
    }

    while((bytes = read(i, buff, TAMBUFF)) > 0)
        for ( j = 0; j < bytes; j++ )
            putchar(buff[ j ]);
    close( i );
    getch( );
}
```

No nosso exemplo, para a função `main`, foram utilizados como parâmetros **argc** (que deve ser um inteiro) e **argv** (que deve ser um vetor ponteiro de caracteres vazios) para referenciar possíveis passagens de parâmetros através da função `main`(). Também utilizamos a função `exit`(), que funciona de forma semelhante à função `HALT` da linguagem de programação Pascal. Sua função é interromper a execução do programa na memória do computador.

Abaixo segue uma lista completa com as opções possíveis de abertura de arquivo binário para a função `fopen`():

“rb”	Abre um arquivo binário para leitura. Nesse tipo de leitura o arquivo deverá estar presente fisicamente no disco.
“wb”	Abre um arquivo binário para gravação de dados. Se o arquivo referenciado já existir ele será sobreposto, caso contrário, será criado um novo.
“ab”	Abre um arquivo binário para gravação. Os dados inseridos serão adicionados no final do arquivo, mantendo assim, os dados já gravados anteriormente.
“rb+”	Abre um arquivo binário para leitura e gravação. O arquivo referenciado deve existir para ser atualizado.
“wb+”	Abre um arquivo binário para leitura e gravação. Caso o arquivo já exista o mesmo será sobreposto. Senão, o mesmo será criado.
“ab+”	Abre um arquivo binário para atualização e para adicionar dados no final do arquivo existente ou um novo arquivo será criado.

Utilizamos a função `open`() para abrir um determinado arquivo que deve ser especificado no próprio programa, e `close`() para fechá-lo.

Na abertura de um arquivo precisamos referenciar os flags de baixo nível que iremos trabalhar para que o compilador C possa referenciar de forma correta um arquivo não-bufferizado (binário).

Os flags na linguagem C, assim como em algumas outras linguagens de programação, servem como “bandeirinhas” de controle para algumas específicas manipulações a serem feitas na própria linguagem. Neste caso, o controle de flags é voltado para a forma de interpretação de uma operação com os diferentes tipos de arquivos existentes na linguagem C.

A seguir, veja a relação de flags aplicáveis num programa em C:

OFLAG	SIGNIFICADO
O_APPEND	Coloca o ponteiro de arquivo no fim dele.
O_CREAT	Cria um novo arquivo para gravação (não faz efeito se já existe)
O_RDONLY	Abre um arquivo somente para leitura.
O_RDWR	Abre um arquivo para leitura e gravação.
O_TRUNC	Abre e trunca um arquivo existente para tamanho ZERO.
O_WRONLY	Abre um arquivo somente para gravação.
O_BINARY	Abre um arquivo em modo binário.
O_TEXT	Abre um arquivo em modo texto.

A função **read()** foi utilizada para ler informações contidas no arquivo.

Outro Exemplo:

```
#include <fcntl.h> /* necessário para trabalhar com oflags */
#include <stdio.h> /* necessário para trabalhar com NULL */
#include <string.h> /* necessário para trabalhar com memchr( ) */
```

```
#define TAMBUFF 1024
char buff[TAMBUFF];
```

```
main(argc, argv)
int argc;
char *argv[ ];
{
int i, bytes;

if(argc != 3)
```

```
{
    printf ("Formato: C>Procura fonte.xxx frase");
    exit( );
}

if(( i = open(argv[1], O_RDONLY)) < 0)
{
    printf ("Não posso abrir %s.", argv[1]);
    exit( );
}

while((bytes = read(i, buff, TAMBUFF)) > 0)
    procur(argv[2], bytes);
close( i );
printf ("frase não encontrada");
}

/* procura frase no buffer */
procur(frase, tambuf)
    char *frase;
    int tambuf;
{
    char *ptr, *p;

    ptr = buff;

    while((ptr = memchr(ptr, frase[0], tambuf)) != NULL)
```

```
if(memcmp(ptr, frase, strlen(frase)) == 0)
{
    printf ("Primeira ocorrencia da frase: \n");

    for ( p = ptr - 100; p < ptr + 100; p++ )
        putchar(*p);
    exit( );
}
else
    ptr++;
}
```

Utilizamos a função **memchr**() que serve para procurar no buffer um determinado caracter especificado. Também usamos a função **memcmp**() que realiza comparações em conteúdos de buffers de memória.

Mais um exemplo:

```
/* Gravando arquivos em baixo-nível */
#include <fcntl.h>
#include <sys\stst.h> /* usada para permissões */

#define TAMBUFF 4096
    char buff[TAMBUFF];

main(argc, argv)
    int argc;
    char *argv[ ];
{
    int origem, dest, bytes;
```

```
    if (argc != 3)
    {
        printf ("Formato: C>copy2 origem.xxx dest.xxx");
        exit( );
    }

    if ((origem = open(argv[1], ORDWR | O_BINARY)) < 0)
    {
        printf ("Não posso abrir o arquivo %s.", argv[1]);
        exit( );
    }

    if((dest = open(argv[2], O_CREAT | O_WRONLY | O_BINARY, S_IWRITE))
        < 0)
    {
        printf ("Não posso abrir o arquivo %s.", argv[2]);
        exit( );
    }
    while((bytes = read(origem, buff, TAMBUFF)) > 0)
        write(dest, buff, bytes);
    close(origem);
    close(dest);
    clrscr( );
    printf("*** F I M ***");
    getch( );
}
```

Já que manipulamos os chamados argumentos de permissão, existem três possibilidades possíveis: `S_IWRITE`, `S_IREAD` e `S_IREAD | S_IWRITE`. Observe a tabela a seguir:

PERMISSÃO	SIGNIFICADO
<code>S_IWRITE</code>	Permissão para gravação.
<code>S_IREAD</code>	Permissão para leitura.
<code>S_IREAD S_IWRITE</code>	Permissão para leitura e gravação.

Independentemente da implementação a ser feita através de um programa escrito em linguagem C, a forma de tratamento é a mesma para qualquer tipo de argumento de permissão.



Exercícios

1. Quais os tipos de arquivos existentes na linguagem C?
2. Na criação de um arquivo texto, quais os comandos utilizados para gravar um caracter por vez e uma string por vez?
3. Quanto à criação de um arquivo binário, qual o comando utilizado para gravar um registro?
4. Escreva em linguagem C um programa que permita ao usuário gerar e ler um arquivo texto para registrar os acontecimentos de sua vida pessoal como se fosse um diário. Utilize como base o seguinte menu de opções:

MEU DIÁRIO

[E] screver no diário

[L] er o que está escrito no diário

[F] echar diário

5. Qual a diferença das permissões `S_IWRITE` e `S_IREAD`?
6. O que significa o flag `O_RDONLY`?
7. Diferencie o modo de abertura de arquivo “`r`” de “`rb`”.
8. O que significa o modo de abertura de arquivo binário “`wb`”?

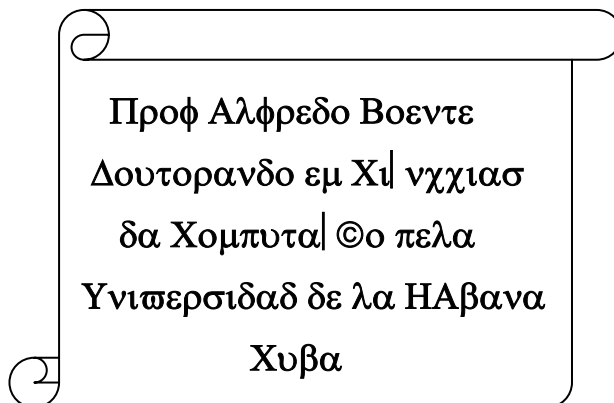
-
9. Para que serve a função `fgets()`?
 10. Qual o comando utilizado para fechar um determinado arquivo nomeado internamente como `ftpr`?
 11. Qual o significado da linha das seguintes linhas de comandos:
 - a) `while((letra=getche()) != '\x1b')`
 - b) `while((ch=getc(fp)) != EOF)`
 - c) `while(fgets(string, 80, fp) != NULL)`
 12. O que significa EOF? Para que é utilizado em um determinado programa escrito em linguagem C?

16

CRIPTOGRAFIA

Embora ninguém saiba realmente quando iniciou o processo de escrita secreta, é afirmado por [Schildt, 89] que um dos exemplos mais antigos é o tablete cuneiforme feito por volta de 1500 a.C.

Esse tablete cuneiforme contém uma fórmula codificada para fazer cobertura vitrificada em cerâmica. Os gregos e espartanos usavam códigos em 475 a.C., e a classe alta romana freqüentemente usava cifras simples durante o reino de Julio César.



Durante a Idade Média, o interesse pela criptografia (bem como muitas outras áreas intelectuais) diminuiu, exceto entre os monges, que as usavam ocasionalmente. Com a vinda do renascimento italiano, a arte da criptografia novamente floresceu.



Na época de Luis XIV da França, um código baseado em 587 chaves randomicamente selecionadas era usado em mensagens governamentais. Por volta de 1800, dois fatores ajudaram no desenvolvimento da criptografia. Primeiro eram as histórias de Edgar Allan Poe, tais como *THE GOLD BUG*, que apresentava mensagens em código e excitava a imaginação dos leitores. O segundo foi a invenção do telégrafo e do código Morse. O código Morse foi a primeira representação binária (pontos e traços) do alfabeto que teve grande aplicação.

Ao chegar a primeira guerra mundial, várias nações construíram “máquinas de codificação” mecânicas que permitiam facilmente codificar e decodificar textos usando cifragens sofisticadas e complexas.

Aqui a história da criptografia dá uma leve virada para a história da decifragem de códigos. Antes da utilização de dispositivos mecânicos para codificar e decodificar mensagens, cifragens complexas eram raramente usadas por causa do tempo e do trabalho necessário para codificar e decodificar. Assim, a maioria dos códigos eram quebrados num período relativamente curto de tempo.

Logo, é possível, a partir de qualquer linguagem de computador, criar um processo de cifragem de códigos. O programa em linguagem C a seguir permite que você seja capaz de codificar qualquer texto usando qualquer descolamento após especificar com que letra começará o alfabeto.

Segue um programa exemplo da utilização prática do processo de criptografia:

```
/* Criptografia através da linguagem C */  
#include "stdio.h"  
#include "conio.h"  
#include "ctype.h"
```

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

    if ( argc != 5)
    {
        clrscr( );
        printf("Uso: Entrada, Saída, Codificação/Decodificação offset\n");
        exit( );
    }

    if (! Isalpha ( *argv[4] ))
    {
        clrscr( );
        printf("O offset tem que ser um caracter alfabético\n");
        exit( );
    }

    if (toupper(*argv[3] == 'C')
        code(argv[1], argv[2], *argv[4]);
    else
        decode(argv[1], argv[2], *argv[4]);
}

code(input, output, start)
```

```
char *input, *output;
char start;
{

    int ch;
    FILE *fp1, *fp2;

    If ((fp1=fopen(input,"r"))==0
    {
        clrscr( );
        printf("Arquivo inacessível para entrada\n");
        exit( );
    }

    if((fp2=fopen(output,"w"))==0
    {
        clrscr( );
        printf("Arquivo inacessível para saída\n");
        exit( );
    }
    start=tolower(start);
    start=start - 'a';
    do{
        ch=getc(fp1);
        if (ch == EOF )
            break;
```

```
    if (isalpha(ch))
    {
        ch += start;

        if (ch > 'z')
            ch -= 26;
    }
    putc(ch, fp2);
}while(1);
fclose(fp1);
fclose(fp2);
}
```

```
decode(input, output, start)
```

```
char *input, *output;
char start;
{

    int ch;
    FILE *fp1, *fp2;

    If ((fp1=fopen(input, "r"))==0
    {
        clrscr( );
        printf("Arquivo inacessível para entrada\n");
        exit( );
    }
```

```
if((fp2=fopen(output,"w"))==0)
{
    clrscr( );
    printf("Arquivo inacessível para saída\n");
    exit( );
}

start = start - 'a';
do{
    ch=getc(fp1);
    if (ch==EOF)
        break;
    if (isalpha(ch))
    {
        ch -= start;

        if (ch('a'))
            ch += 26;
    }
    putc(ch, fp2);
}while(1);
fclose(fp1);
fclose(fp2);
}
```

Para este programa utilizamos as seguintes funções da linguagem C:

toupper() ⇒ Converte um caracter minúsculo em Maiúsculo.

tolower() ⇒ Converte um caracter Maiúsculo em minúsculo.

isalpha() ⇒ É uma macro em C que tem por objetivo classificar um caracter de acordo com os códigos estabelecidos na tabela ASCII.

Existem muitos outros métodos utilizados através da linguagem C para codificar e/ou decodificar uma mensagem criptografada. Para obter maiores informações e um maior aprofundamento no assunto e outras variações de programas exemplo em linguagem C, você poderá fazer uma revisão de literatura em [*Schildt, 89*].

17

ROTINAS DE ROM-BIOS



Agora vamos aprender como funcionam as chamadas rotinas de ROM-BIOS, que nada mais são do que interrupções em hexadecimal cuja função é realizar um tipo de tratamento físico, na “arquitetura de um computador”.

As rotinas de ROM-BIOS que são encontradas neste livro foram elaboradas por **Silvio Lago** e resgatadas, na íntegra, do livro *Treinamento em Linguagem C*, curso completo – módulo 2, das páginas de 237 até 246, de *Victorine Viviane Mizrahi*.

Interrupção 05 Hexa (Imprime Tela)

Função 05:	Imprime tela na impressora
Registradores de Entrada:	AH = 05
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 00:	Define modo de vídeo
Registradores de Entrada:	AH = 00, AL = modo de vídeo
	Modos de Vídeo
	00 : 40x25 texto 16 cinza
	01 : 40x25 texto 18/8 cor
	02 : 80x25 texto 16 cinza

03 : 80x25 texto 16/8 cor
04 : 320x200 gráf, 4 cores
05 : 320x200 gráf, cinza
06 : 640x200 gráf, p&b
07 : 80x25 texto p&b
08 : 160x200 gráf, 16 cores
09 : 320x200 gráf, 16 cores
0A : 640x200 gráf, 4 cores

Registadores de Saída: nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 01: Define o tamanho do cursor
Registadores de Entrada: AH = 01
CH = linha varredura inicial
CL = linha varredura final
Adaptador graf/colorido (0 a 7)
Adaptador monocromático (0 a 13)
Registadores de Saída: nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 02: Posiciona cursor
Registadores de Entrada: AH = 02
BH = número da página de vídeo
DH = linha
DL = coluna

Registradores de Saída: nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 06: Sobe a tela

Registradores de Entrada:

AH = 06

AL = linha a subir

BH = atributo preenchedor

CH = linha superior

CL = coluna esquerda

DH = linha inferior

DL = coluna direita

Registradores de Saída:

nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 07: desce a tela

Registradores de Entrada:

AH = 07

AL = linha a descer

BH = atributo preenchedor

CH = linha superior

CL = coluna esquerda

DH = linha inferior

DL = coluna direita

Registradores de Saída:

nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 08:	Lê caracteres e atributos
Registradores de Entrada:	AH = 08 BH = número de página de vídeo
Registradores de Saída:	AH = atributo AL = character

Interrupção 10 Hexa (Serviços de Tela)

Função 09:	Escreve caracteres e atributos
Registradores de Entrada:	AH = 09 AL = character BH = número da página BL = atributo CX = número de character a repetir
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 0A:	Escreve character
Registradores de Entrada:	AH = 0A AL = character BH = número da página BL = cor do modo gráfico CX = contador de caracteres
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 0B:	Define paleta de cor
Registradores de Entrada:	AH = 0B BH = identificação da paleta BL = cor a ser usada com a identificação da paleta
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 0C:	Escreve ponto na tela
Registradores de Entrada:	AH = 0C AL = cor CX = coluna do ponto DL = linha do ponto
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 0D:	Lê ponto da tela
Registradores de Entrada:	AH = 0D CX = coluna do ponto DL = linha do ponto
Registradores de Saída:	AL = cor lida

Interrupção 10 Hexa (Serviços de Tela)

Função 0E:	Imprime caracteres como TTY
Registradores de Entrada:	AH = 0E AL = caracter BH = número de página BL = cor no modo gráfico
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 0F:	Obtém modo de vídeo corrente
Registradores de Entrada:	AH = 0F
Registradores de Saída:	AH = largura em caracteres AL = modo de vídeo BH = número de página

Interrupção 10 Hexa (Serviços de Tela)

Função 13 (para AT):	Escreve cadeia de caracteres Não move o cursor
Registradores de Entrada:	AH = 13 AL = 00 BL = atributo BH = número de página de vídeo DX = posição inicial do cursor CX = tamanho da cadeia ES:BP = ponteiro para o início da cadeia
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 13 (para AT):	Escreve cadeia de caracteres Move o cursor para o final da cadeia
Registradores de Entrada:	AH = 13 AL = 01 BL = atributo BH = número da página de vídeo DX = posição inicial do cursor CX = tamanho da cadeia ES:BP = ponteiro para o início da cadeia
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 13 (para AT):	Escreve cadeia de atributos e caracteres Não move o cursor
Registradores de Entrada:	AH = 13 AL = 02 BH = número da página de vídeo DX = posição inicial do cursor CX = tamanho da cadeia ES:BP = ponteiro para o início da cadeia
Registradores de Saída:	nenhum

Interrupção 10 Hexa (Serviços de Tela)

Função 13 (para AT):	Escreve cadeia de atributos e caracteres Move o cursor para o final da cadeia
Registradores de Entrada:	AH = 13 AL = 03 BH = número da página de vídeo DX = posição inicial do cursor CX = tamanho da cadeia ES:BP = ponteiro para o início da cadeia
Registradores de Saída:	nenhum

Interrupção 11 (Lista de Equipamento)

Função:	Lista periféricos conectados
Registradores de Entrada:	nenhum
Registradores de Saída:	AX = lista de equipamentos 0 : drive de disquete 1 : coprocessador matemático 2-3 : RAM da placa de sistema em bloco de 16k 4-5 : modo de vídeo inicial 00 : não usado 01 : 40x25 cor 10 : 80x25 cor 11 : 80x25 p&b 6-7: número de drivers de disquetes menos 1 8 : 0 : DMA presente

1 : DMA ausente
 9-10-11: número de cartões RS-232
 no sistema
 12 : E/S de jogo conectada
 13 : não usado
 14-15: número de impressoras
 conectadas

Interrupção 12 (Função de Memória)

Função:	Obtém memória disponível
Registradores de Entrada:	nenhum
Registradores de Saída:	AX = tamanho da memória em Kb

Interrupção 13 (Funções de Disco)

Função 00:	Reseta sistema de disco
Registradores de Entrada:	AH = 00
Registradores de Saída:	nenhum

Interrupção 13 (Funções de Disco)

Função 01:	Obtém estado do disco
Registradores de Entrada:	AH = 01
Registradores de Saída:	AH = código de estado em hexa: A : flag de mau setor (F) AA : drive não pronto (F) BB : erro indefinido (F)

CC : gravação (F)
EO : erro de estado (F)
1 : comando errado
2 : marca de endereço não encontrado
3 : tentativa de gravação em disco protegido (D)
4 : setor não encontrado
5 : falha no reset (F)

Interrupção 13 (Funções de Disco)

Função 02: Lê setores do disco
Registadores de Entrada: AH = 02
AL = número de setores
CH = número de trilhas
CL = número de setor
DH = número de cabeça
DL = número de drive
ES:BX = ponteiro para o buffer
Registadores de Saída: CF = flag de sucesso/falha
AH = código de estado
(veja função 01)
AL = número de setores lidos

Interrupção 13 (Funções de Disco)

Função 03: Grava setores no disco
Registadores de Entrada: AH = 03

AL = número de setores
CH = número de trilhas
CL = número de setor
DH = número de cabeça
DL = número de drive
ES:BX = ponteiro para o buffer

Registadores de Saída:

CF = flag de sucesso/falha
AH = código de estado
(veja função 01)
AL = número de setores gravados

Interrupção 13 (Funções de Disco)

Função 04: Verifica setores do disco

Registadores de Entrada:

AH = 04
AL = número de setores
CH = número de trilhas
CL = número de setor
DH = número de cabeça
DL = número de drive

Registadores de Saída:

CF = flag de sucesso/falha
AH = código de estado
(veja função 01)
AL = número de setores verificados

Segue abaixo um exemplo de programa em linguagem C que utiliza o recurso da interrupção de hardware através das funções de ROM-BIOS.



```
/* Oculta e reexibe o cursor */
#include "stdio.h"
#include "dos.h"

#define TAMCUR 1
#define VIDEO 0x10
#define STOPBIT 0x20

void main()
{
    union REGS regs;
    regs.h.ch = STOPBIT;
    regs.h.ah = TAMCUR;

    int86(VIDEO, &regs, &regs);
}
```

A função **int86**() cria uma interrupção através do uso de registradores (*regs*) da linguagem C.