

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ELETRÔNICA

CURSO DE LINGUAGEM C APLICADA À ENGENHARIA

PROFESSOR: FERNANDO CÉSAR COMPARI DE CASTRO

1) INTRODUÇÃO:

Introdução básica:

```
#include <stdio.h>
/* Programa-exemplo #1. */
main()
{
    int idade;
    idade = 33;
    printf ("A minha idade é %d\n", idade);
}
```

```
#include <stdio.h>
/* Programa-exemplo #2 - conversão de pés para metros. */
main()
{
    int pés;
    float metros;
    printf("Informe o número de pés: ");
    scanf("%d", &pés);
    metros = pés * 0.3048;
    printf("%d pés é %f metros\n", pés, metros);
}
```

```
/* & -> operador "endereço de" */
/* conversão de pés para metros */
```

```
#include <stdio.h>
/* Programa-exemplo #2, 2ª versão - conversão de pés para metros. */
main()
{
    float pés, metros;
    printf("Informe o número de pés: ");
    scanf("%f", &pés);
    metros = pés * 0.3048;
    printf("%f pés é %f metros\n", pés, metros);
}
```

```
/* torna a variável pés um float */
/* lê um float */
/* conversão de pés para metros */
```

Introdução às Funções em C:

```
#include <stdio.h>
/* Um programa-exemplo com duas funções. */
main()
{
    hello();
}
hello()
{
    printf("Alô\n");
}
```

```
/* chama a função hello */
```

```

#include <stdio.h>
/* Um programa que usa uma função com um argumento. */
main()
{
int num;
num=100;
sqr(num);          /*chama sqr() com num */
}

sqr(int x)         /*a declaração do parâmetro está dentro dos
                  parênteses*/
{
printf("%d elevado ao quadrado é %d\n", x, x*x);
}

#include <stdio.h>
/* Outro exemplo de função com argumentos. */
main()
{
mul(10, 11);
}

mul(a,b)
int a,b;          /* observe a forma diferente de declarar a função
                  mul() */
{
printf("%d", a*b);
}

#include <stdio.h>
/* Um programa que usa retorno. */
main()
{
int resposta;
resposta = mul (10, 11);    /* atribui o valor de retorno */
printf("A resposta é %d\n", resposta);
}
/* Esta função retorna um valor */
mul(int a, int b)
{
return a*b;
}

```

Forma geral de uma função:

```
tipo-de-retorno nome-da-função (lista de parâmetros)
identificação dos parâmetros (caso não identificados na lista)
{
  corpo do código
}
```

Para funções sem parâmetros, não haverá lista de parâmetros.

Introdução ao controle de programas em C:

A declaração if:

if(condição)declaração;
onde condição é uma expressão que resulta V ou F. Em “C”, V é não zero e F é zero.

Exemplos:

```
if(10 < 11) printf("10 é menor que 11");           /* imprime */
if(10 == 11) printf("Alô");                       /* não imprime */
```

O loop for:

for(inicialização; condição; incremento)declaração;

“inicialização” é usado p/ atribuir à variável de controle do loop um valor inicial
“condição” é uma expressão que é testada toda vez que o loop continua a ser executado
“incremento” incrementa a variável de controle do loop

```
#include <stdio.h>
/* Um programa que ilustra o loop for */
main()
{
  int contador;
  for(contador=1; contador<=100; contador++) printf("%d ", contador);
}
```

Obs) contador++ tem o mesmo efeito que contador=contador+1
contador-- tem o mesmo efeito que contador=contador-1

Blocos de código:

Como “C” é uma linguagem ESTRUTURADA, ela suporta a criação de blocos de código:

```
if(x<10) {
int novo;
printf(“Muito baixo, tente outra vez!\n”);
novo=x+2;
printf(“Que tal x=%d?”,novo);
scanf(“%d”, &x);
}
```

Nota: a variável “novo” só existe dentro deste bloco de código, sendo eliminada da memória após o fechamento da última chave }.

Básico sobre Caracteres e Strings:

```
#include <stdio.h>
/* Um exemplo usando caracteres.*/
main()
{
char ch;
ch = 'A';
printf(“%c”, ch);
ch = 'B';
printf(“%c”,ch);
ch = 'C';
printf(“%c”, ch);
}
```

```
#include<stdio.h>
#include<conio.h>
main()
{
char ch;
ch = getch(); /* lê um caractere do teclado */
if(ch= 'H') printf(“você pressionou a minha tecla mágica\n”);
}
```

Em “C”, uma string é uma seqüência de caracteres (char) organizada em uma matriz unidimensional (vetor) terminada pelo caracter nulo (null). O caracter nulo é o 0. A linguagem “C” não tem uma variável tipo string propriamente como o Basic. Em vez disso você declara uma matriz de caracteres e usa as várias funções de strings encontradas na biblioteca adequada para manipulá-las.

Para ler uma string do teclado, primeiro cria-se uma matriz de caracteres suficientemente grande para armazenar a string mais o terminador nulo. Por exemplo, para armazenar a palavra "hello", é necessário uma matriz de caracteres de 6 elementos: 5 p/ a string e 1 p/ o terminador nulo: 'h' 'e' 'l' 'l' 'o' '\0'

Declara-se a matriz de caracteres como qualquer variável: char txt[6];

```
Assim txt[0] = 'h'
      txt[1] = 'e'
      txt[2] = 'l'
```

```
txt[3] = 'l'
txt[4] = 'o'
txt[5] = '\0'
```

Para a leitura de uma string do teclado, pode-se usar função `scanf()` como veremos adiante, ou usar a função `gets()`. A função `gets()` toma o nome da string como argumento e faz a leitura dos caracteres do teclado até que a tecla ENTER seja pressionada. A tecla ENTER não é armazenada como caractere, mas sim substituída pelo terminador nulo.

```
#include<stdio.h>
/* Um exemplo com string. */
main()
{
char str[80];
printf("Digite o seu nome: ");
gets(str);
printf("Alô %s", str);
}
```

Fazendo um breve resumo do que se viu de `printf()` até agora:

A forma geral da função `printf()` é:

```
printf("string de controle", lista de argumentos)
```

Código	Significado
%d	Exibe um inteiro no formato decimal
%f	Exibe um tipo float no formato decimal
%c	Exibe um caractere
%s	Exibe uma string

Exemplos:

```
printf("%s %d", "Esta é uma string ", 100);
exibe:
Esta é uma string 100
```

```
printf("esta é uma string %d", 100);
exibe:
esta é uma string 100
```

```
printf("o número %d é decimal, %f é ponto flutuante.",10, 110.789);
exibe:
o número 10 é decimal, 100.789 é ponto flutuante.
```

```
printf("%s", "Alô\n");
exibe:
Alô
e avança uma linha
```

2) VARIÁVEIS, CONSTANTES , OPERADORES E EXPRESSÕES:

Tipos de dados:

Tamanho e intervalo dos tipos básicos de dados do Turbo C:

Tipo	Tamanho(em bits)	Intervalo
char	8	-128 a 127
int	16	-32768 a 32767
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
void	0	sem valor

Modificadores de Tipos: signed, unsigned, long, short:

Todas as combinações possíveis dos tipos básicos e modificadores do Turbo C.

Tipo	Tamanho(em bits) [8 bits = 1 byte]	Intervalo decimal
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-32768 a 32767
unsigned short int	16	0 a 65535
signed short int	16	-32768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
long double	80	3.4E-4932 a 1.1E+4932

```
#include <stdio.h>
/* Ilustra a diferença entre inteiros com sinal e sem sinal. */
main()
{
int i;                               /* um inteiro com sinal */
```

```

unsigned int j;                                /* um inteiro com sinal */

j = 60000;
i = j;
printf("%d %u", i, j);
}

```

```

/* Este programa imprime o alfabeto. */
#include <stdio.h>
main()
{
char letra;
for(letra = 'A' ; letra <= 'Z' ; letra ++)
printf("%c ", letra);
}

```

Declarando variáveis:

variável local:	declarar após declaração de main() ou função().
variável global:	declarar antes da declaração de main().

```

int i, j, l;                                  /* Observe a separação por vírgulas */
short int si;
unsigned int ui;
double balanço, lucro, perda;

```

Exemplo:

```

/* Somatório de números de 0 a 9.*/
#include <stdio.h>
int soma;
main()
{
int count;
soma = 0;
for(count=0; count<10; count++) {
total(count);
display();
}
}

```

/* Variável Global */

/* Variável Local */

/* inicializacao*/

```

total(int x)                                  /* adiciona ao total corrente */
{                                              /* Parâmetro Formal */
soma = x + soma;
}
display()

```

```

{
it count;                               /* Variável Local, este count é diferente daquele count na */
for(count=0; count<10; count++) printf("."); /*função main()*/
printf("o somatório corrente é %d\n", soma);
}

```

Constantes:

Tipo de Dado	Exemplos de Constantes
Char	'a' 'n' '9'
int	1 123 21000 -234
long int	35000 -34
short int	10 -12 90
unsigned int	10000 987 40000
float	123.23 4.34e-3
double	123.23 12312333 -0.9876324

Constantes Hexadecimais e Octais :

```

hex = 0xFF;                               /* 255 em decimal */
oct 011;                                   /* 9 em decimal */

```

Códigos de Barra Invertida:

```

ch = '\t';
printf("este é um teste\n");

```

Códigos de Barra Invertida.

Código	Significado
\b	Retrocesso
\f	Alimentação de formulário
\n	Nova linha
\r	Retorno de carro
\t	Tabulação horizontal
\"	Aspas
\'	Apóstrofo
\0	Nulo
\\	Barra invertida
\v	Tabulação vertical
\a	Sinal sonoro
\N	Constante Octal (onde N é uma constante octal)
\xN	Constante hexadecimal (onde N é uma constante hexadecimal)

Inicialização de Variáveis:

```

tipo nome_da_variável=constante;
char ch = 'a';
int primeiro = 0;
float balanço = 123.23;

```

```

/* Um exemplo usando inicialização de variáveis. */
#include <stdio.h>
main()
{
int t;
printf("informe um número: ");
scanf("%d", &t);
total(t);
}
total(int x)
{
int sum=0, i, count;
for(i=0; i<x; i++) {
sum = sum + i;
for(count=0; count<10; count++) printf(".");
printf("a somatória corrente é %d\n", sum);
}
}

```

Operadores Aritméticos:

Operador	Ação
-	subtração, também o menos unário (-1*x)
+	adição
*	multiplicação
/	divisão
%	resto da divisão
--	decremento
++	incremento

Obs) Operador unário é aquele que opera sobre um único argumento.

```

/* cálculo do quociente e do resto de uma divisão */
#include <stdio.h>
main()
{
int x,y;
printf("informe o dividendo e o divisor: ");
scanf("%d%d", &x, &y);
printf("quociente da divisão = %d\n", x/y);
printf("resto da divisão = %d", x%y);
}

```

Incremento e Decremento:

x= x+1;

pode ser escrito como:

++x; ou

x++;

/* primeiro incrementa a variável depois usa o conteúdo */

/* primeiro usa o conteúdo da variável depois incrementa */

Exemplos:

```

x=10;          /* x=10 */
y= ++x;       /* primeiro faz x=x+1 e depois faz y=x */
              /* resultado: y=11 e x=11 */

x=10;          /* x=10 */
y= x++;       /* primeiro faz y=10 e depois faz x=x+1 */
              /* resultado: y=10 e x=11 */

```

O mesmo vale para decremento `x--` e `--x`.

Precedência de operadores:

- 1) ++ --
- 2) - (unário)
- 3) * / %
- 4) + -

Obs1) Operadores do mesmo nível de precedência são avaliados pelo compilador da esquerda para a direita.

Obs2) Parênteses podem ser colocados para alterar a ordem de avaliação, forçando uma operação ou conjunto de operações para um nível de precedência mais alto. Aconselha-se sempre o uso de parênteses, já que além de deixarem o código mais claro, eles não acrescentam nenhum tempo de execução adicional ou consumo extra de memória.

Operadores Relacionais e Lógicos:

Operadores Relacionais		Operadores Lógicos	
>	Maior que	&&	AND(E)
>=	Maior ou igual que		OR(OU)
<	Menor que	!	NOT(NÃO)
<=	Menor ou igual que		
==	Igual		
!=	Diferente		

```

/* Este programa ilustra os operadores relacionais. */
#include <stdio.h>
main()
{
int i, j;
printf("informe dois numeros: ");
scanf("%d%d", &i, &j);
printf("%d == %d é %d\n", i, j, i==j);
printf("%d != %d é %d\n", i, j, i!=j);
printf("%d <= %d é %d\n", i, j, i<=j);
printf("%d >= %d é %d\n", i, j, i>=j);
printf("%d < %d é %d\n", i, j, i<j);
printf("%d > %d é %d\n", i, j, i>j);
}

```

Obs) Os operadores relacionais podem ser aplicados a qualquer tipo de dado básico. O exemplo a seguir mostra a comparação entre dois char:

```
ch1='A';
ch2='B';
if(ch2 > ch1) printf("maior que");
```

```
/* Este programa ilustra os operadores lógicos. */
#include <stdio.h>
main()
{
int i, j;
printf("informe dois números(cada um sendo 0 ou 1): ");
scanf("%d%d", &i, &j);
printf("%d AND %d é %d\n", i, j, i && j);
printf("%d OR %d é %d\n", i, j, i || j);
printf("NOT %d é %d\n", i, !i);
}
```

Observe que todas as expressões lógicas e relacionais produzem um resultado 0 ou 1. Portanto, o seguinte programa não só está correto como produzirá o número 1 na tela:

```
#include <stdio.h>
main()
{
int x;
x=100;
printf("%d", x>10);
}
```

```
/* Imprime os números pares entre 1 e 100. */
#include <stdio.h>
main()
{
int i;
for(i=1; i<=100; i++)
if(!(i%2)) printf("%d ",i);
}
/* o operador de resto dará falso (zero) */
/* quando usada c/ número par. Esse resultado*/
/* é invertido pelo NOT */
```

Operador de Atribuição:

Em C o operador de atribuição é o sinal de igual '='. Ao contrário de outras linguagens, o C permite que o operador de atribuição seja usado em expressões relacionais ou lógicas:

```
#include <stdio.h>
main()
{
int x, y, produto;
printf("informe dois números: ");
scanf("%d%d", &x, &y);
if( (produto=x*y) < 0)
printf ("Um dos números é negativo\n");
else
printf("O produto dos números positivos é: %d", produto);
}
/* expressão relacional c/ operador de atribuição */
/* caso contrário da condição do if */
```

```
}
```

Modeladores(Casts):

(tipo) expressão
onde tipo é qualquer tipo de dado padrão em C.

Uso: Forçar o resultado de uma expressão a ser determinado tipo de dado.

Exemplo: Seja x um int e y um float.

```
x=3;
y= (float) x/2;          /* y=1.5 */
y= (float) (x/2)        /* y=1.0 */

#include <stdio.h>
main()                  /*imprime i e i/3 com a parte fracionária */
{
int i;
for(i=1; i<=100; ++i )
printf("%d/3 é: %f\n", i, (float) i/3);
}
```

3) DECLARAÇÕES DE CONTROLE DO PROGRAMA:

Declaração if:

A forma geral da declaração if é:
if(condição)declaração;
else declaração;

A cláusula else é opcional. Se condição é V (!=0), declaração (ou bloco de declarações) que forma o destino de if é executada, caso contrário a declaração ou bloco que forma o destino do else é executada.

O programa a seguir ilustra o uso do if. Este programa introduz dois novos formatos para scanf() e printf() que são o %x e %o, associados à leitura (scanf) e impressão (printf) respectivamente de dados hexadecimais e octais.

```
/* Programa de conversão de base numérica #1.
decimal    --> hexadecimal
hexadecimal --> decimal
decimal    --> octal
octal      --> decimal*/
#include<stdio.h>
main()
{
int opção;
int valor;
printf("Converter:\n");
printf(" 1: decimal para hexadecimal\n");
printf(" 2: hexadecimal para decimal\n");
printf(" 3: decimal para octal\n");
printf(" 4: octal para decimal\n");
printf("informe a sua opção:");
```

```

scanf("%d", &opção);
if(opcao==1) {
printf("informe um valor em decimal:");
scanf("%d", &valor);
printf("%d em hexadecimal é : %x", valor, valor);
}
if(opção==2) {
printf("informe um valor em hexadecimal:");
scanf("%x", &valor);
printf("%x em decimal é: %d", valor, valor);
}
if(opção==3){
printf("informe um valor em decimal:");
scanf("%d", &valor);
printf("%d em octal é: %o", valor, valor);
}
if(opção==4){
printf("informe um valor em octal:");
scanf("%o", &valor);
printf("%o em decimal é: %d", valor, valor);
}
}
}

```

Declaração else:

```

/* Um exemplo de if_else.*/
#include<stdio.h>
main()
{
int i;
printf("informe um número:");
scanf("%d", &i);
if(i<0) printf("o número é negativo:");
else printf ("o número é positivo ou zero");
}

```

Encadeamento if-else-if:

```

if(condição)
declaração;
else if(condição)
declaração;
else if(condição)
declaração;
.
.
.
else

```

declaração;

```
/* Programa de conversão de base numérica #2 - encadeamento if-else-if.
decimal    --> hexadecimal
hexadecimal --> decimal
decimal    --> octal
octal      --> decimal */
```

```
#include<stdio.h>
main()
{
int opção;
int valor;
printf("Converter:\n");
printf("    1: decimal para hexadecimal\n");
printf("    2: hexadecimal para decimal\n");
printf("    3: decimal para octal\n");
printf("    4: octal para decimal\n");
printf("informe a sua opção:");
scanf("%d", &opção);
if(opção==1) {
printf("informe um valor em decimal:");
scanf("%d", &valor);
printf("%d em hexadecimal é : %x", valor, valor);
}
else if(opção==2) {
printf("informe um valor em hexadecimal:");
scanf("%x", &valor);
printf("%x em decimal é: %d", valor, valor);
}
else if(opção==3){
printf("informe um valor em decimal:");
scanf("%d", &valor);
printf("%d em octal é: %o", valor, valor);
}
else if(opção==4){
printf("informe um valor em octal:");
scanf("%o", &valor);
printf("%o em decimal é: %d", valor, valor);
}
}
```

Validade da Expressão Condicional: Zero=Falso Não Zero=Verdadeiro:

Exemplo:

```
/* Divide o primeiro número pelo segundo. */
#include<stdio.h>
main()
{
int a, b;
printf("informe dois números: ");
scanf("%d%d", &a, &b);
if(b) printf("%f\n", (float) a/b);
else printf("não posso dividir por zero\n");
/* só executa se b!=0 */
```

```
}
```

Não é necessário explicitamente usar `if(b == 0) printf("%f\n", (float) a/b);`

Declaração switch: (similar a ON-GOTO do BASIC):

Difere do if-else-if por só poder testar igualdades. No entanto, gera um código .EXE menor e mais rápido que if-else-if.

```
switch(variável) {
case constante1:
    sequencia de declaracoes
    break;
case constante2:
    sequencia de declaracoes
    break;
case constante3:
    sequencia de declaracoes
    break;
.
.
.
default
    sequencia de declaracoes
}
```

```
/* Programa de conversao de base numerica #3 usando-se a declaracao switch.
```

```
decimal --> hexadecimal
```

```
hexadecimal --> decimal
```

```
decimal --> octal
```

```
octal --> decimal
```

```
*/
```

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

```
main()
```

```
{
```

```
int opcao;
```

```
int valor;
```

```
printf("Converter:\n");
```

```
printf(" 1: decimal para hexadecimal\n");
```

```
printf(" 2: hexadecimal para decimal\n");
```

```
printf(" 3: decimal para octal\n");
```

```
printf(" 4: octal para decimal\n");
```

```
printf("informe a sua opcao:");
```

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

```

switch(opcao) {
case 1:
    printf("informe um valor em decimal:");
    scanf("%d", &valor);
    printf("%d em hexadecimal é : %x", valor, valor);
    break;
case 2:
    printf("informe um valor em hexadecimal:");
    scanf("%x", &valor);
    printf("%x em decimal é: %d", valor, valor);
    break;
case 3:
    printf("informe um valor em decimal:");
    scanf("%d", &valor);
    printf("%d em octal é: %o", valor, valor);
    break;
case 4:
    printf("informe um valor em octal:");
    scanf("%o", &valor);
    printf("%o em decimal é: %d", valor, valor);
    break;
}
}

```

Declaracao default:

Usada para prever a possibilidade de nenhuma correspondencia ser encontrada:

```

switch(opcao) {

case 1:
    printf("informe um valor em decimal:");
    scanf("%d", &valor);
    printf("%d em hexadecimal e : %x", valor, valor);
    break;
case 2:
    printf("informe um valor em hexadecimal:");
    scanf("%x", &valor);
    printf("%x em decimal e: %d", valor, valor)
    break;
case 3:
    printf("informe um valor em decimal:");
    scanf("%d", &valor);
    printf("%d em octal e: %o", valor, valor);
    break;
case 4:
    printf("informe um valor em octal:");
    scanf("%o", &valor);
    printf("%o em decimal e: %d, valor, valor);
    break;

default: /* quando nenhuma correspondencia é encontrada */

```

```
printf(opcao invalida, tente outra vez\n");
break;
}
```

Loop for:

A forma geral do loop for é:

```
for(inicializacao; condicao; incremento) declaracao;
```

-inicialização é uma declaração de atribuição que inicializa a variável de controle do loop.

-condição é uma expressão relacional que determina qual situação terminará a execução do loop pelo teste da variável de controle contra algum valor. Uma vez falsa a condição o programa seguirá na declaração seguinte ao término do loop for.

-incremento define como a variável de controle mudará ao longo da execução do loop.

-declaração (ou bloco de declarações) é o que o loop for executa enquanto ativo.

```
/* Imprime numeros de 1 a 100 na tela */
#include <stdio.h>
main()
{
    int x;
    for(x=1; x<=100; x++) printf ("%d ",x);
}
```

```
/* Imprime numeros de 100 a 1 na tela */
#include <stdio.h>
main()
{
    int x;
    for(x=100; x>0; x--) printf("%d ",x);
}
```

```
/* Imprime numeros de 0 a 100 de 5 em 5 */
#include <stdio.h>
main()
{
    int x;
    for(x=0; x<=100; x=x+5) printf("%d ",x);
}
```

```

/* Imprime quadrado dos numeros de 0 a 99 */
#include <stdio.h>
main()
{
    int i;
    for(i=0; i<100; i++){
        printf("Este é o valor de i: %d ",i);
        printf(" E i ao quadrado é: %d\n", i*i);
    }
}

```

Obs) O teste condicional é realizado no início do loop. Isto significa que o código pode nunca ser executado se a condição é falsa logo de início:

```

x=10
for(y=10; y!=x, ++y) printf("%d", y);
printf("fim do loop");

```

O loop acima nunca é executado já que de início $x=y$. Portanto a condição é avaliada como falsa e o loop acaba prematuramente.

Variações do loop for:

```

/* Este programa imprime os numeros de 0 a 98 de 2 em 2 */
#include<stdio.h>
main()
{
    int x, y;
    for(x=0,y=0; x+y<100; ++x,y++) /* Duas variáveis de controle: x e y */
        printf("%d ", x+y);      /* Em C a vírgula é um operador que signifi- */
                                /* ca em essência: "faça isto E aquilo". Por */
                                /* isso é válida a construção deste loop for. */
}
                                /* O incremento poderia ser todas as com-
                                binações de incremento anterior e poste-
                                rior, já que neste caso este fato nao in-
                                fluencia a performance do programa */

```

```

/*Ensino da adição.*/
#include<stdio.h>
#include<conio.h>

main()
{
    int i, j, resposta;
    char fim = " ";

    for(i=1; i<100 && fim!='N';i++){ /* loop ativo quando i<100 e fim != 'N' */
        for(j=1; j<10; j++) {
            printf("quanto é %d + %d?", i, j);
            scanf("%d", &resposta);
            if(resposta != i+j) printf("errado\n");
            else printf("certo\n");
        }
    }
    printf("mais?");
}

```

```

fim=getche();
printf("\n");
}
}

```

Usos nao corriqueiros do loop for:

```

/* Um uso nao usual do laço for.*/
#include <stdio.h>
main()
{
    int t;
    for(prompt(); t=readnum(); prompt()) /* se t==0 loop for termina */
        sqrnum(t);
}

/* O loop for acima funciona porque (inicializacao;condicao;incremento)
podem consistir de qualquer expressao válida em C */

```

```

prompt()
{
    printf("informe um inteiro:");
}

```

```

readnum()
{
    int t;
    scanf("%d",&t);
    return t;
}
sqrnum(int num)
{
    printf("%d\n", num*num);
}

```

Inclusive é possível deixar faltando partes da definição do loop. O programa abaixo pega a senha do teclado e testa se é igual 135. Se for igual, condição torna-se falsa e o loop encerra com uma mensagem:

```

#include <stdio.h>
main()
{
    int x;

    for(x=0; x!=135;) scanf("%d", &x);
    printf("senha correta");

}

```

Loop infinito:

```

for(;;) {
    printf("este laço sera executado para sempre.\n");
    printf("use CTRL-BREAK para parar!");
}

```

Interrupção de um loop for:

```
for(;;){
    ch=getche(); /* pega um caractere */
    if(x=='A') break; /* break: sai do laço e vai p/ a declaração seguinte */
}
printf("voce digitou um A");
```

Loop while:

A forma geral é :

```
while(condição)declaração;
```

O conjunto de declarações em declaração é repetido enquanto condição é verdadeira.

Exemplo: A rotina a seguir pára o processamento e entra em loop até que o caractere A maiúsculo seja pressionado:

```
espera_por_caractere()
{
    char ch;
    ch='\0'; /*inicializa ch*/
    while(ch!='A') ch= getche();
}
```

/* Um programa que centraliza o texto na tela.*/

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

```
main()
{
    char str[255];
    printf("insira uma string:");
    gets(str);

    centraliza(strlen(str));
    printf(str);
}
```

/* Calcula e exibe um numero adequado de
espacos para centralizar uma string de comprimento tamanho.

```
*/
centraliza(int tamanho)
{
    tamanho = (80-tamanho)/2; /* 80-tamanho é o que sobra na linha de 80 col.*/
    while(tamanho>0){
        printf(" ");
        tamanho--;
    }
}
```

```
}
```

Onde muitas condições distintas podem terminar o loop while, é comum usar uma única variável como expressão condicional com o valor dessa variável sendo indicado em vários pontos ao longo do loop:

```
funcl()
{
  int trabalhando;
  trabalhando =1; /*isto é, verdadeira*/
  while(trabalhando){
    trabalhando=process1();
    if(trabalhando)
      trabalhando= process2();
    if(trabalhando)
      trabalhando= process3();
  }
}
```

No código acima, qualquer das três rotinas process() podem retornar falso (zero) e fazer com que o loop seja encerrado.

Não há em absoluto a necessidade de qualquer declaração no corpo do loop while. Por exemplo:

```
while( (ch=getche()) != 'A' ); /* a atribuição à ch dentro da expressão */
/* condicional do while funciona por que o */
/* sinal de igual é na realidade somente um */
/* operador que especifica o valor do seu */
/* operando à direita */
```

simplesmente será executado até que o caractere A seja digitado.

Loop infinito c/ while:

```
while(1){conjunto de declarações};
```

é idêntico ao

```
for(;;){conjunto de declarações};
```

já que a condição do while é sempre 1 = verdadeiro.

Loop do-while (faça enquanto):

```
do{
declaração:
}while(condição);
```

Exemplos:

```
/* lê numeros do teclado até que um deles seja menor que 100 */
#include<stdio.h>
main()
```

```

{
int num;
do {
scanf("%d",&num);
printf("voce digitou %d\n", num);
}while(num>100);
}

```

O uso mais comum do loop do-while é em rotinas de seleção de menu. Depois que a scanf() pegar a opção, o loop se repetirá até que o usuário digite uma opção válida no exemplo abaixo:

```

/* certifica-se de que o usuario especificou uma opção valida*/
#include <stdio.h>
main()
{
int opcao;
do{
printf("Converter:\n");
printf("    1: decimal para hexadecimal\n");
printf("    2: hexadecimal para decimal\n");
printf("    3: decimal para octal\n");
printf("    4: octal para decimal\n");
printf("informe a sua opção:");
scanf("%d", &opcao);
} while(opcao<1||opcao>4);

printf("Voce digitou a opção %d", opcao);

}

```

Loops aninhados:

/*Exibe uma tabela das quatro primeiras potencias dos numeros de 1 a 9.*/

```

#include<stdio.h>
main()
{
int i, j, k, temp;
printf("    i    i^2    i^3    i^4\n");
for(i=1;i<10; i++){ /* laço externo: define o numero base*/
for(j=1; j<5; j++){ /* primeiro nivel do aninhamento */
temp = 1;
for (k=0; k<j; k++)
{temp = temp*i;} /* laço mais interno: eleva a j */
printf("%9d",temp); /* se um numero é colocado entre o % e o d, ele */
} /* diz a printf() p/ adicionar os espaços neces- */
printf("\n"); /* sários à largura especificada */
}
}

```

/* Programa de conversao de base numerica: versao final

usando laços while aninhados.

```
        decimal --> hexadecimal
        hexadecimal --> decimal
        decimal --> octal
        octal --> decimal
*/
#include <stdio.h>
main()
{
    int opcao;
    int valor;
    /* repete ate que o usuario diga para terminar */
    do{
        /* certifica-se de que o usuario especifica uma opção valida */
        do{
            printf("\nConverter:\n");
            printf("    1: decimal para hexadecimal\n");
            printf("    2: hexadecimal para decimal\n");
            printf("    3: decimal para octal\n");
            printf("    4: octal para decimal\n");
            printf("informe a sua opcao:");
            scanf("%d", &opcao);
        } while(opcao<1||opcao>5);

        switch(opcao) {
            case 1:
                printf("informe um valor em decimal:");
                scanf("%d", &valor);
                printf("%d em hexadecimal é : %x", valor, valor);
                break;

            case 2:
                printf("informe um valor em hexadecimal:");
                scanf("%x", &valor);
                printf("%x em decimal é: %d", valor, valor);
                break;
            case 3:
                printf("informe um valor em decimal:");
                scanf("%d", &valor);
                printf("%d em octal é: %o", valor, valor);
                break;
            case 4:
                printf("informe um valor em octal:");
                scanf("%o", &valor);
                printf("%o em decimal é: %d", valor, valor);
                break;
        }
        printf("\n");
    } while(opcao!=5);
}
```

Quebrando um loop:

```
/* imprime na tela os numeros de 1 a 10 */
```

```

#include <stdio.h>
main()
{
    int t;
    for(t=0; t<100; t++){
        printf("%d ", t);
        if(t==10) break;
    }
}

/* Como esta o seu senso de tempo? */
#include <stdio.h>
#include <time.h>
#include <conio.h>

main()
{
    long tm;

    printf("Este programa testa o seu senso de tempo!\n");
    printf("Quando estiver pronto, pressione a tecla return, espere cinco segundos\n");
    printf("e pressione qualquer tecla.");
    getch();
    printf("\n");

    tm = time(0);
    for (;;) if(kbhit()) break;
    if(time(0)-tm==5) printf("Voce venceu!!!");
    else printf("O seu sincronismo está mal. Erro de %d segundos!",(int) time(0)-tm-5);
}

```

Observe que um break causará uma saída somente do laço mais interno. Por exemplo:

```

for(t=0; t<100; ++t){
    count=1;
    for(;;){
        printf("%d", count);
        count++;
        if(count==10) break;
    }
}

```

O código acima imprimirá os números de 1 a 10 cem vezes na tela. Toda vez que o break é encontrado, o controle é devolvido para o laço for externo.

Outra observação é o fato que um break usado dentro de uma declaração switch afetará somente os dados relacionados com o switch e não qualquer outro laço em que o switch estiver.

Declaração continue:

Continue opera da mesma maneira que o break, mas em vez de terminar o loop e seguir na declaração seguinte, continue força a ativação da próxima iteração do loop, pulando qualquer código que esteja entre eles.

Exemplos:

```

/* Imprime na tela os números pares até 99 */

#include<stdio.h>
main()
{
    int x;
    for(x=0; x<100; x++){
        if(x%2) continue; /* se o resto é zero o loop continua */
        printf("%d ", x);
    }
}

```

/*Um codificador simples.*/

```

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

main()
{
    printf("Insira as letras que voce quer codificar.\n");
    printf("Digite um $ quando estiver no final das letras.\n");

    code();
}

```

```

code() /* Codifica as letras */
{
    char pronto, ch;

    pronto=0;
    while(!pronto){
        ch=getch();

        if(ch=='$'){
            pronto=1;
            continue;
        }

        printf("%c", ch+1); /* desloca o alfabeto uma posição */
    }
}

```

Labels e goto:

Semelhante ao "GOTO line" do BASIC e FORTRAN, com a diferença que o destino é um LABEL (rótulo) e não um número de linha do programa. Somente pode ser usado em desvios dentro da mesma função, não podendo transferir a execução do programa de uma função para outra.

Exemplos:

O código abaixo implementa um loop de 1 a 100:

```
x=1;
loop1: /* 2 pontos após o nome do label! (loop1) */
x++;
if(x<100)goto loop1;
```

O goto torna-se bastante útil quando é necessário sair de uma rotina profundamente aninhada:

```
for(...){
  for(...){
    while(...){
      if(...)goto Lstop; /* desvio condicional p/ o label Lstop */
      .
      .
      .
    }
  }
}

Lstop:
printf("erro no programa\n")
```

Observe que se usarmos um break em vez do goto seremos forçados a usar uma série de testes adicionais, já que o break só sai do loop mais interno:

```
stop=0; /* stop é a variável de controle da saída da situação detetada pelo if */
for(...){
  for(...){
    while(...){
      if(...) /* se ocorre determinada situação stop=1 */
      {stop=1;
      break;}
    }
    .
    .
    .
  }
  if(stop)break;
}
if(stop) break;
}

printf("erro no programa\n")
```

4) MATRIZES E STRINGS:

Matrizes Unidimensionais:

A forma geral de uma matriz de simples dimensão é

tipo var_nome[tamanho]

"tipo" ----- declara o tipo base da matriz

"tipo base" --- determina o tipo de dado de cada elemento
que compreende a matriz

"tamanho" ---- define quantos elementos a matriz armazenara

int exemplo [10]

Na linguagem C, todas as matrizes tem o zero como indice do seu primeiro elemento. Portanto, a linha anterior declara uma matriz que possui 10 elementos tipo inteiro, de "exemplo[0]" a "exemplo[9]".

O programa seguinte carrega uma matriz de inteiros com os numeros de 0 a 9:

```
main()
{
  int x [10];/* esta declaração reserva 10 elementos inteiros*/
  int t;
  for(t=0; t<10; ++t) x[t]=t;
}
```

Para uma matriz unidimensional, o tamanho total da matriz em bytes é computado como mostrado aqui:

"Bytes total = tamanho do tipo base * numero de elementos"

```
/*Encontra a média de dez numeros. */
#include<stdio.h>
main()
{
  int exemplo [10], i, media;
  printf("\nInforme os números X[i] a medida que forem solicitados:\n\n");
  for(i=0; i<10; i++){
    printf("X[%d]? ",i);
    scanf("%d", &exemplo[i]);
  }
  media=0;
  /* agora soma os numeros */
  for (i=0; i<10; i++) media = media+exemplo[i];
  printf("A media é %f\n", (float)media/10);
}
```

Sem verificação de limites:

```
/* programa incorreto. Nao execute-o!*/
main()
{
  int quebra [10], i;
```

```
for (i=0; i<100; i++) quebra [i] =i; /* excede o índice de quebra[10] */
}
```

Matrizes Unidimensionais São Listas:

```
char str [7];
main()
{
    int i;
    for(i=0; i<7; i++) str i = 'A'+i;
}
```

a variável str ficará semelhante ao que segue:

```
str[0]  str[1]  str[2]  str[3]  str[4]  str[5]  str[6]
  A      B      C      D      E      F      G
```

Strings:

Uma "string" é definida como sendo constituída de uma matriz de caracteres (tipo de dado char - 1 byte) que é terminada por um "nulo". Um nulo é especificado usando-se '\0' que é zero.

Por exemplo, se quisermos declarar uma matriz "str" que possa armazenar uma string de 10 caracteres, escrevemos:

```
char str [11];
Uma "constante string" é uma lista de caracteres entremeados por aspas. Por exemplo:
```

```
"Alo"
"este é um teste"
```

Não é necessário adicionar manualmente o nulo no final das constantes string - o compilador faz isso automaticamente.

Lendo uma string do teclado:

A maneira mais fácil de inserir uma string pelo teclado é com a função de biblioteca gets(). A forma geral da função gets() é:

```
gets(nome_da_matriz);
```

```
/* Um programa simples com string.*/
#include <stdio.h>
main()
{
    char str [80];
    printf("informe uma string:");
    gets(str); /*le a string do teclado*/
    printf("%s", str);
}
```

Algumas funções de biblioteca que manipulam string:

A linguagem C suporta um grande numero de funcoes que manipulam strings. As mais comuns sao:

```
strcpy()
strcat()
strlen()
strcmp()
```

A funcao strcpy():

Uma chamada a strcpy() tem esta forma geral:
strcpy(para,de);

A funcao strcpy() é usada para copiar o conteudo da string "de" para a string "para".

```
/* exemplo de uso de strcpy() */
#include <stdio.h>
#include <string.h>
main()
{
    char str [80];
    strcpy(str, "Alo");
    printf("%s", str);
}
```

A funcao strcat():

Uma chamada a strcat() tem esta forma:

```
strcat(s1,s2);
```

A funcao strcat() anexa (concatena) "s2" em " s1"; "s2" nao é modificada. As duas strings devem ser terminadas com nulo e o resultado tambem será terminado com um nulo. Por exemplo, este programa imprimirá "Alo para todos" na tela:

```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[20], s2[15];
    strcpy(s1, "Alo");
    strcpy(s2, " para todos");
    strcat(s1,s2);
    printf("%s",s1);
}
```

Como strcpy(), strcat() retorna um valor, cujo significado será visto adiante.

A funcao strcmp():

Uma chamada a strcmp() tem esta forma:

```
strcmp(s1,s2);
```

A função strcmp() compara duas strings e retorna 0 se elas forem iguais. Se s1 é lexicograficamente maior (Exemplos: "BBBB">"AAAA" e "AAA">"X") que s2, então um número positivo é retornado; se for menor que s2, um número negativo é retornado.

A seguinte função pode ser usada como uma rotina de verificação de senha:

```
/* Retorna verdadeiro se a senha é aceita; falso, caso contrário.*/
#include<stdio.h>
main()
{
    char s[80];
    printf("informe a senha:");
    gets(s);
    if(strcmp(s, "senha"))
    { /*strings diferentes*/
        printf("senha invalida\n");
        return 0;
    }
    /* a string comparada é a mesma*/
    return 1;
}
```

O segredo da utilização da função "strcmp()" é que ela retorna falso quando as strings são iguais. Portanto, você precisará usar o operador NOT se desejar que alguma coisa ocorra quando a string é igual.

```
/* le strings até que se digite 'sair' */
#include <stdio.h>
#include <string.h>
main()
{
    char s[80];
    for (;;) {
        printf("Informe uma string:");
        gets(s);
        if(!strcmp("sair",s))break;
    }
}
```

A Função strlen():

A forma geral da chamada à "strlen()" é:

```
strlen(s);
```

onde s é uma string.

A função "strlen()" retorna o tamanho de s.

O programa seguinte imprimirá o tamanho da string que for inserida pelo teclado:

```
/* imprime o tamanho da string digitada */
#include<stdio.h>
#include<string.h>
main()
{
```

```

char str [80];
printf("digite uma string:");
gets(str);
printf("%d", strlen(str));
}

```

```

/* Imprime uma string na ordem inversa.*/
#include <stdio.h>
#include <string.h>
main()
{
char str [80];
int i;
printf("digite uma string:");
gets(str);
for(i=strlen(str)-1; i>=0; i--) printf("%c", str[i]);
}

```

```

/* informa tamanho e igualdade entre duas strings e concatena */
#include<stdio.h>
#include<string.h>

main()
{
char s1 [80], s2 [80];
printf("digite duas strings:");
gets(s1); gets(s2);
printf("tamanhos: %d %d\n", strlen(s1), strlen(s2));
if(!strcmp(s1,s2))printf("As strings sao iguais\n");
strcat(s1,s2);
printf("%s", s1);
}

```

Usando o terminador nulo:

```

/* converte uma string para letras maiúsculas */

#include<stdio.h>
#include<string.h>
#include<ctype.h>

main()
{
char str[80];
int i;
printf("Informe uma string em letras minúsculas:");
gets(str);
for(i=0;str[i];i++) str[i] = toupper(str[i]); /* for termina quando */
printf("%s",str); /* str[i] = \0' */
}

```

Obs) O complementar de toupper() é tolower().

Uma Variação da Função printf():

```
#include <stdio.h>
#include <string.h>
main()
{
char str[80];
strcpy(str, "Alo Tom");
printf(str);/*funciona porque o primeiro argumento de printf está
entre aspas e pode ser considerado uma string*/
}
```

Matrizes Bidimensionais:

Para declarar por exemplo uma matriz bidimensional de inteiros chamada de nome bid, com tamanho 10,20 , escrevemos:

```
int bid[10] [20];
```

Exemplos:

```
/* carrega uma matriz bidimensional com os números de 1 a 12 */
#include<stdio.h>
main()
{
int t, i, num[3][4];
for(t=0; t<3; t++){
for(i=0;i<4;i++){
num[t][i]=(t*4)+i+1; /* lei de formação dos elementos da matriz num */
printf("num[%d][%d]= %d\n",t,i,num[t][i]);
}
}
}
```

No caso de uma matriz bidimensional, a seguinte expressão encontrará o número de bytes da memória:

bytes=linha*coluna*tamanho do tipo de dado

Portanto, uma matriz de inteiros com dimensões [10][5] deverá ter 10x5x2 ou 100 bytes alocados, já que um int tem o tamanho de 16bits = 2bytes (1 byte= 8 bits).

Matrizes de strings:

A linha abaixo declara uma matriz de 30 strings, cada uma com tamanho máximo de 80 caracteres:

```
char matriz_str[30][80];
```

É muito fácil acessar uma string individual: voce especifica somente o índice da esquerda. Por exemplo, esta declaração chama a função gets() com a terceira string em "matriz_str":

```
gets(matriz_str[2]);
```

O que é equivalente a (embora não tenhamos visto ponteiros ainda):

```
gets(&matriz_str[2][0]);
```

Para entender melhor como as matrizes de caracteres funcionam, consideremos o programa seguinte, que aceita linhas de texto inseridas pelo teclado e exibe-as assim que uma linha em branco é inserida:

```
/*Recebe e exibe strings.*/
#include <stdio.h>
main()
{
    register int t, i;
    char texto [100][80];

    for(t=0;t<100;t++){
        printf("%d:",t);
        gets(texto[t]); /* t indexa linhas */
        if(!texto[t][0]) break; /*sai com uma linha em branco*/
    }

    /*reapresenta a string:*/
    for(i=0;i<t;i++)
        printf("%s\n",texto[i]); /* i indexa linhas aqui */
    }
}
```

Matrizes Multidimensionais:

A linguagem C permite matrizes com mais de duas dimensões.

A forma geral de uma matriz multidimensional é:

```
tipo_nome[tamanho1][tamanho2]...[tamanhoN];
```

Por exemplo, a linha abaixo cria uma matriz 4x10x3 de inteiros:

```
int trid [4][10][3];
```

Matrizes de três ou mais dimensões usualmente requerem uma quantidade grande de memória para armazenamento. A área requerida para armazenar elementos de matrizes é alocada permanentemente durante a execução do programa. Por exemplo, uma matriz quadridimensional de caracteres, com dimensões 10, 6, 9, 4, deve requerer 10x6x9x4 ou 2.160 bytes, já que 1 char = 8 bits = 1 byte.

Se a matriz for de inteiros de dois bytes, serão necessários 4.320 bytes. Se a matriz for do tipo "double"(8 bytes de comprimento), então 34.560 bytes serão requeridos. O espaço necessário aumenta exponencialmente com o número de dimensões. Um programa com matrizes com mais de três ou quatro dimensões pode ficar rapidamente sem memória suficiente para ser executado, dependendo do tipo de modelo de memória usado pelo programa (usualmente usa-se o modelo small, c/ 64Kb p/ código e 64Kb p/ dados).

Inicialização de Matrizes:

A linguagem C permite a inicialização de matrizes. A forma geral da inicialização de matrizes é similar à de outras variáveis, como mostrado aqui:

```
especificador_de_tipo nome da matriz[tamanho1]...[tamanhoN]={lista_de_valores};
```

A lista_de_valores é uma lista de constantes separadas por vírgulas que é do tipo compatível com o tipo base da matriz. A primeira constante será colocada na primeira posição da matriz, a segunda constante, na segunda posição e assim por diante. Note que um ponto e vírgula segue a chave }. No exemplo seguinte, uma matriz de 10 elementos inteiros é inicializada como os números de 1 a 10:

```
int i[10]= { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Isso significa que i[0] terá o valor 1 e i[9], o valor 10.

Matrizes de caracteres que armazenam strings permitem uma inicialização abreviada com a seguinte forma:

```
char nome_da_matriz[tamanho]="string";
```

Por exemplo, este fragmento de código inicializa a variável "str" com a palavra "Alô":

```
char str[4] = "Alô";
```

Que é o mesmo que escrever:

```
char str[4] = {'A', 'l', 'ô', '\0'};
```

Já que as strings na linguagem C devem terminar com um nulo, certifique-se de que a matriz que você declara é grande o suficiente para incluí-lo. É por isso que a variável "str" tem o tamanho de quatro caracteres apesar de "Alô" ter somente três. Quando a constante string é usada, o compilador provê automaticamente o terminador nulo.

Matrizes multidimensionais são inicializadas da mesma maneira que as unidimensionais. Por exemplo, as linhas abaixo inicializam a matriz "sqrs" com os números de 1 a 10 e seus respectivos quadrados:

```
int sqrs[10][2] = {
  1, 1,
  2, 4,
  3, 9,
  4, 16,
  5, 25,
  6, 36,
  7, 49,
  8, 64,
  9, 81,
  10, 100,
};
```

Inicialização de Matrizes sem Especificação de Tamanho:

Imagine que você esteja usando inicialização de matriz para construir uma tabela de mensagens de erros, como mostrado aqui:

```
char e1 [20] ="informação inválida\n";
char e2 [23] = "seleção fora_do_limite\n";
char e3 [19] = "autorização negada\n";
```

Como você pode supor, é bastante monótono contar manualmente os caracteres em cada mensagem para determinar a dimensão correta da matriz. É possível deixar a linguagem C dimensionar automaticamente as matrizes, neste exemplo, pelo uso de "matrizes sem determinação de tamanho".

Se, em uma declaração de inicialização de matriz, o tamanho não é especificado, o compilador criará uma matriz grande o suficiente para armazenar todos os inicializadores presentes.

Usando este método, a tabela de mensagem fica assim:

```
char e1 [ ]="informação inválida\n";
char e2 [ ]="seleção fora_do_limite\n";
char e3 [ ]="autorização negada\n"; /*O compilador conta ele próprio o
número de elementos e cria as matrizes de tamanho adequado*/
```

A inicialização de matrizes sem a especificação de tamanho não está restrita somente a matrizes unidimensionais. Para matrizes multidimensionais, deve-se especificar tudo, menos a dimensão mais à esquerda (compiladores mais modernos aceitam especificação em aberto da dimensão mais à direita), para permitir que a linguagem C indexe adequadamente a matriz. Dessa maneira, é possível construir tabelas de diferentes tamanhos com o compilador alocando espaço suficiente para elas. Por exemplo, a declaração da matriz "sqrs" como uma matriz sem especificação de tamanho é mostrada aqui:

```
int sqrs[ ][2] = {
1, 1,
2, 4,
3, 9,
4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100,
};
```

Vantagem: A tabela pode ser aumentada indefinidamente porque o compilador se encarrega de contar as linhas da matriz.

Um Jogo de Caça ao Submarino:

No jogo de caça ao submarino o computador controla um submarino e você um couraçado. O objetivo do jogo é uma embarcação destruir a outra. Contudo, os navios têm capacidades de radar limitadas. A única maneira do submarino encontrar o couraçado é estar diretamente sob ele. O vencedor é aquele que encontrar o outro primeiro, com o submarino e o couraçado alternando movimentos aleatórios sobre o "oceano".

O "oceano" é um pequeno campo de jogo que mede 76,2 x 76,2 mm². Tanto o submarino quanto o couraçado tratam esse plano como coordenadas X, Y, com 0,0 sendo o canto superior esquerdo. Uma matriz bidimensional, chamada de "matriz", é usada para guardar o tabuleiro de jogo e cada elemento nele é inicializado com um espaço, o que denota um elemento não usado.

A função main() c/ suas variáveis globais são mostradas aqui:

```
char matriz [3] [3] = { /* vars. globais */
' ', ' ', ' ',
' ', ' ', ' ',
' ', ' ', ' ',
};
int compX, compY, jogX, jogY;
```

```

main()
{
    randomize(); /*inicializa (gera a semente inicial) aleatoriamente o gerador de números randômicos*/
    compX=compY=jogX=jogY=0; /* inicializa vars de coordenadas */
    for(;;){
        if(tent_sub()){
            printf("O submarino venceu!\n");
            break;
        }
        if(tent_jog()){
            printf("O couraçado (você) venceu!\n");
            break;
        }
        exibe_tabuleiro();
    }
}

```

A função `randomize()` serve para randomizar o gerador de número randômico da linguagem C, que é usado para gerar o movimento do computador, como veremos em breve.

A primeira função necessária é aquela que cria o movimento do computador. O computador (submarino) gera o seu movimento usando o gerador de números randômicos da linguagem C `rand()`, que retorna um número randômico entre 0 e 32.767. Tanto a função `rand()` como `randomize()` usam o arquivo cabeçalho (header) `stdlib.h`. A função `randomize()` requer ainda o arquivo `time.h`. Um valor randômico é gerado tanto para a coordenada X quanto para a Y, toda a vez que o computador é chamado a gerar um movimento. Esses valores são usados então em uma operação de módulo para prover um número entre 0 e 2. Finalmente, o computador verifica se encontrou (e por definição destruiu) o couraçado. Se o movimento gerado tem as mesmas coordenadas da localização atual do couraçado, o submarino vence. Se o computador venceu, a função retorna verdadeira, caso contrário retorna falso. A função `tent_sub()` é apresentada aqui:

```

tent_sub() /* gera o próximo movimento do computador usando um
           gerador de números randômicos.*/
{
    matriz [compX] [compY]=' '; /* apaga última posição do submarino */
    compX=rand() % 3; /* o resto de uma divisão por 3 é sempre 0 1 ou 2 */
    compY=rand() % 3;
    if(matriz[compX][compY] == 'B')
        return 1; /* O submarino ganhou a batalha*/
    else{
        matriz[compX][compY]='S';
        return 0; /* Ele não acertou*/
    }
}

```

O couraçado recebe o seu próximo movimento do jogador. A função requisita ao usuário as novas coordenadas e então verifica se o submarino foi encontrado ou não. Em caso positivo, o jogador vence e a função retorna verdadeiro. Caso contrário a função retorna falso. A função `tent_jog()` é mostrada aqui:

```

tent_jog() /*pega o próximo movimento do jogador*/
{
    matriz[jogX] [jogY]=' '; /* apaga última posição do couraçado */
    do{
        printf("Informe as novas coordenadas (X,Y): ");

```

```

scanf("%d%d",&jogX,&jogY);
} while (jogX<0||jogX>2||jogY<0||jogY>2); /* testa validade das coords */
if(matriz[jogX][jogY]== 'S')
return 1; /*O couraçado ganhou a batalha*/
else{
matriz[jogX][jogY]='B';
return 0; /* Ele nao acertou*/
}
}

```

Após cada série de movimentos, o tabuleiro do jogo é exibido pela função "exibe_tabuleiro()". Uma célula nao usada fica vazia. A célula que guarda a última posição do submarino conterá um S e a célula que guarda a posição do couraçado, um B:

```

exibe_tabuleiro() /*Exibe tabuleiro do jogo*/
{
printf("\n");
printf("%c | %c | %c\n", matriz[0][0],matriz[0][1],matriz[0][2]);
printf("-|---|--\n");
printf("%c | %c | %c\n", matriz[1][0],matriz[1][1],matriz[1][2]);
printf("-|---|--\n");
printf("%c | %c | %c\n", matriz[2][0],matriz[2][1],matriz[2][2]);
}

```

O programa inteiro é mostrado aqui:

```

/* Jogo Couraçado x Submarino */
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /*requerido por randomize()*/

char matriz [3] [3] = { /* vars. globais */
' ', ' ', ' ',
' ', ' ', ' ',
' ', ' ', ' '
};
int compX, compY, jogX, jogY;

main()
{
randomize(); /*inicializa (gera a semente inicial) aleatoriamente o gerador de números randômicos*/
compX=compY=jogX=jogY=0;
for(;;){
if(tent_sub()){
printf("O submarino venceu!\n");
break;
}
if(tent_jog()){
printf("O couraçado (você) venceu!\n");
break;
}
exibe_tabuleiro();
}
}

```

```

tent_sub() /* gera o próximo movimento do computador usando um
          gerador de números randômicos.*/
{
    matriz [compX] [compY]= ' ';
    compX=rand() % 3;
    compY=rand() % 3;
    if(matriz[compX][compY] == 'B')
        return 1; /* O submarino ganhou a batalha*/
    else{
        matriz[compX][compY]= 'S';
        return 0; /* Ele nao acertou*/
    }
}
tent_jog() /*pega o próximo movimento do jogador*/
{
    matriz[jogX] [jogY]= ' ';
    do{
        printf("Informe as novas coordenadas (X,Y): ");
        scanf("%d%d",&jogX,&jogY);
    } while (jogX<0||jogX>2||jogY<0||jogY>2);
    if(matriz[jogX][jogY]== 'S')
        return 1; /*O couraçado ganhou a batalha*/
    else{
        matriz[jogX][jogY]='B';
        return 0; /* Ele nao acertou*/
    }
}
exibe_tabuleiro() /*Exibe tabuleiro do jogo*/
{
    printf("\n");
    printf("%c | %c | %c\n", matriz[0][0],matriz[0][1],matriz[0][2]);
    printf("-|---|--\n");
    printf("%c | %c | %c\n", matriz[1][0],matriz[1][1],matriz[1][2]);
    printf("-|---|--\n");
    printf("%c | %c | %c\n", matriz[2][0],matriz[2][1],matriz[2][2]);
}

```

5) PONTEIROS:

A compreensão e o uso correto de ponteiros são críticos para criação de muitos programas de êxito na linguagem C. Há três motivos para isso. Primeiro, os ponteiros provêm o meio para que as funções possam modificar os seus argumentos de chamada. Segundo, eles são usados para suportar rotinas de alocação dinâmica da linguagem C. Terceiro, podem ser substituídos pelas matrizes em muitas situações - proporcionando aumento de eficiência. Além disso, muitas das características do C apóiam-se firmemente nos ponteiros, portanto, um completo entendimento de ponteiros é muito importante. Além de ser uma das características mais fortes da linguagem C, os ponteiros também são muito perigosos. Por exemplo, quando não-inicializados ou descuidados, eles podem causar o travamento do sistema. E pior: é fácil usar ponteiros incorretamente, o que causa "bugs" (erros) difíceis de serem encontrados.

Ponteiros São Endereços:

Um ponteiro é uma variável que guarda um endereço de memória de outro objeto. Mais comumente, esse endereço é a localização de outra variável na memória, embora ele possa ser o endereço de uma porta ou de propósito-especial na RAM, como uma área auxiliar (buffer). Se uma variável contém o endereço de uma outra variável, a primeira é conhecida como um ponteiro para a segunda. Essa situação é ilustrada abaixo:

Seja o seguinte código:

```
float mult= 315.2;
int sum = 32000;
char letter = 'A', /* equivalente a char letter = 65 */
int *set; /* declara o ponteiro set */
float *ptr1;
set=&sum; /* inicializa pointer set */
ptr1=&mult; /* inicializa pointer ptr1 */
```

O código anterior gera o seguinte mapa de memória:

Endereço de memória: (hexadecimal)	Valor da variável na memória (decimal exceto indicação contrária):	
FA10	3.152 E 02	→ mult (4 bytes)
FA11		
FA12		
FA13		
FA14	32000	→ sum (2 bytes)
FA15		
FA16	65	→ letter (1 byte) ⇒ Valor ASCII do caracter 'A'
FA17	FA14 (hex)	→ set (2 bytes) ⇒ set aponta para sum
FA18		
FA19	FA10 (hex)	→ ptr1 (2 bytes) ⇒ ptr1 aponta para mult
FA1A		

Comentário:

Variáveis Ponteiro:

Se uma variável vai armazenar um ponteiro, ela deve ser declarada como tal. Uma declaração de ponteiro consiste em um tipo base, um * e o nome da variável. A forma geral para declaração de uma variável ponteiro é:

```
tipo * nome-da-variável;
```

onde o tipo pode ser qualquer tipo válido da linguagem C e o nome-da-variável é o nome da variável ponteiro. O tipo base do ponteiro define qual tipo de variáveis o ponteiro pode apontar. Por exemplo, estas declarações criam um ponteiro caractere e dois ponteiros inteiros.

```
char *p;
int *temp, *início;
```

Os Operadores de Ponteiros:

Existem dois operadores especiais de ponteiro: o `&` e o `*`. O `&` é um operador unário que retorna o endereço de memória do seu operando. (Um operador unário requer somente um operando.) Por exemplo,

```
count_addr = &count;
```

coloca em `count_addr` o endereço de memória da variável `count`. Esse endereço é a localização interna da variável no computador. Ele não faz nada com o valor de `count`. A operação do operador `&` pode ser lembrada como "o retorno de endereço da variável que a ele sucede". Portanto, a atribuição dada pode ser determinada como "a variável `count_addr` recebe o endereço de variável `count`".

Para entender melhor essa atribuição, assumamos que a variável `count` esteja localizada no endereço 2000. Então, após a atribuição, `count_addr` terá o valor de 2000.

O segundo operador é o `*`, e é o complemento do operador `&`. Ele é um operador unário que retorna o valor da variável localizada no endereço que o segue. Por exemplo, se `count_addr` contém o endereço de memória da variável `count`, então

```
val = *count_addr;
```

colocará o valor de `count` em `val`. Se `count` originalmente tem o valor 100, então `val` terá o valor de 100, porque este é o valor armazenado na localização 2000, que é o endereço de memória que foi atribuído a `count_addr`. A operação do operador `*` pode ser lembrada como "valor contido no endereço". Neste caso, então, a declaração pode ser lida como "val recebe o valor que está no endereço `count_addr`".

Com relação ao mapeamento de memória anterior, envolvendo as variáveis `mult`, `sum`, `letter` e os ponteiros `set` e `ptr1`, se tivéssemos feito a declaração de uma outra variável inteira, digamos, de nome `teste`, e fizéssemos a seguinte declaração:

```
teste = *set; /* teste recebe o valor da variável p/ a qual set aponta */
```

`teste` receberia o valor de `sum` que é 32000.

De mesma maneira se `teste` fosse declarada como `float` e fizéssemos

```
teste = *ptr1;
```

`teste` receberia o valor de `mult` que é 315.2.

Infelizmente, o sinal de multiplicação e o sinal "valor no endereço" são os mesmos. Isso algumas vezes confunde iniciantes na linguagem C. Esses operadores não se relacionam um com o outro. Tanto `&` como `*` têm precedência maior que todos os outros operadores aritméticos, exceto o menos unário, com o qual eles se igualam.

Aqui está um programa que usa esses operadores para imprimir o número 100 na tela:

```
/* imprime 100 na tela */
#include <stdio.h>
main()
{
```

```

int *count_addr, count, val;
count = 100;
count_addr= & count; /* pega o endereço de count */
val=*count_addr; /*pega o valor que está no endereço count_addr*/
printf("%d", val); /*exibe 100*/
}

```

Tipo Base:

Como o compilador sabe quantos bytes copiar em val do endereço apontado por count_addr? Mais genericamente, como o compilador transfere o número adequado de bytes em qualquer atribuição usando um ponteiro? A resposta é que o tipo base do ponteiro determina o tipo de dado que o compilador assume que o ponteiro está apontando. Nesse caso, uma vez que count_addr é um ponteiro inteiro, 2 bytes de informação são copiados em val a partir do endereço apontado por count_addr. Se ele fosse um ponteiro double, então 8 bytes teriam sido copiados.

```

#include <stdio.h>
/* Este programa não funciona adequadamente */
main()
{
float x=10.1, y;
int *p;
p=&x; /* observar warning emitido pelo compilador */
y=*p;
printf("%f",y);
}

```

Este programa não atribuirá o valor da variável x para y. Tendo em vista que p é declarado para ser um ponteiro para um inteiro, somente 2 bytes de informação serão transferidos para y, não os 4 bytes que normalmente formam um número de ponto flutuante.

Expressões com Ponteiros:

Atribuição de Ponteiros:

Como qualquer variável, um ponteiro pode ser usado no lado direito de uma declaração para atribuir seu valor para um outro ponteiro. Por exemplo:

```

#include<stdio.h>
main()
{
int x;
int *p1, *p2;
x=101;
p1=&x;
p2=p1;

printf("Na localização %p ", p2); /* imprime o valor hexadecimal do endereço */
/* de x e não o valor de x */
printf("está o valor %d\n", *p2); /* agora imprime o valor de x */
}

```

O endereço, em hexadecimal, de x é exibido usando-se outro dos códigos de formatação da função printf(). O %p especifica que um endereço de ponteiro é para ser exibido usando-se a notação hexadecimal.

Aritmética com Ponteiros:

Somente duas operações aritméticas podem ser usadas com ponteiros: adição e subtração. Para entender o que ocorre na aritmética com ponteiros, consideremos que p1 seja um ponteiro para um inteiro com o valor atual 2000.

Depois da expressão

```
p1++;
```

o conteúdo de p1 será 2002, não 2001. Cada vez que é incrementado, p1 apontará para o próximo inteiro. O mesmo é válido para decrementos. Por exemplo,

```
p1--;
```

fará com que a variável p1 tenha o valor 1998, assumindo-se que ela tinha anteriormente o valor 2000.

Cada vez que é incrementado, um ponteiro apontará para a próxima localização de memória do seu tipo base. Cada vez que é decrementado, apontará para a localização do elemento anterior. No caso de ponteiros para caracteres, parecerá ser uma aritmética normal. Porém, todos os outros ponteiros incrementarão ou decrementarão pelo tamanho do tipo dado para o qual apontam. Por exemplo, assumindo-se caracteres de 1 byte e inteiros de 2 bytes, quando um ponteiro caractere é incrementado, seu valor é incrementado de um; entretanto, quando um ponteiro inteiro é incrementado, seu valor é incrementado de dois bytes. Um ponteiro float seria incrementado de 4 bytes.

Exemplo:

```
char *chptr;
int *iptr;
float *flptr;

chptr=0x3000;
iptr=0x6000;
flptr=0x9000;

chptr++; /* ch=0x3001 */
iptr++; /* i=0x6002 */
flptr++; /* flptr=0x9004 */
```

Porém, você não está limitado a somente incrementar e decrementar. Pode-se adicionar ou subtrair inteiros para ou de ponteiros. A expressão:

```
p1=p1+9;
```

fará p1 apontar para o nono elemento do tipo base de p1 considerando-se que ele está apontando para o primeiro.

Exemplo:

```
double *p1;
p1=0x1000; /* p1 = 4096 */
p1=p1+9 /* p1 = 0x1048 = 4096 + 9*8 */
```

Comparação com Ponteiros:

É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros p e q, a seguinte declaração é perfeitamente válida:

```
if(p<q) printf("p aponta para um endereço de memória mais baixo que q\n");
```

Geralmente, comparações entre ponteiros devem ser usadas somente quando dois ou mais ponteiros estão apontando para um objeto em comum.

Ponteiros e Matrizes:

Há uma relação entre ponteiros e matrizes. Considere este fragmento de código:

```
char str[80], *p1;  
p1 = str;
```

Aqui, p1 foi associado ao endereço do primeiro elemento da matriz em str. Na linguagem C, um nome de matriz sem um índice é o endereço para o começo da matriz (em geral um ponteiro contém o endereço do início de uma área de armazenamento ou transferência de dados). O mesmo resultado - um ponteiro para o primeiro elemento da matriz str - pode ser gerado com a declaração abaixo:

```
p1=&str[0];
```

Entretanto, ela é considerada uma forma pobre por muitos programadores em C. Se você deseja acessar o quinto elemento em str, pode escrever:

```
str[4]
```

ou

```
*(p1+4)
```

As duas declarações retornarão o quinto elemento.

LEMBRE-SE: Matrizes começam em índice zero, então um 4 é usado para indexar str. Você também pode adicionar 4 ao ponteiro p1 para obter o quinto elemento, uma vez que p1 aponta atualmente para o primeiro elemento de str.

A linguagem C permite essencialmente dois métodos para acessar os elementos de uma matriz. Isso é importante porque a aritmética de ponteiros pode ser mais rápida do que a indexação de matriz. Uma vez que a velocidade é frequentemente importante em programação, o uso de ponteiros para acessar elementos da matriz é muito comum em programas em C.

Para ver um exemplo de como ponteiros podem ser usados no lugar de indexação de matriz, considere estes dois programas - um com indexação de matriz e um com ponteiros -, que exibem o conteúdo de uma string em letras minúsculas.

```
#include <stdio.h>  
#include <ctype.h>  
/* versao matriz */  
main()  
{
```

```

char str[80];
int i;
printf("digite uma string em letra maiúscula: ");
gets(str);
printf("aqui está a string em letra minúscula: ");
for(i=0; str[i]; i++) printf("%c", tolower(str[i]));
}
#include <stdio.h>
#include <ctype.h>
/* versao ponteiro */
main()
{
char str[80], *p;
printf("digite uma string em letra maiúscula: ");
gets(str);
printf("aqui está a string em letra minúscula: ");
p=str; /* obtém o endereço de str*/
while (*p) printf("%c", tolower(*p++));
}

```

A versao matriz é mais lenta que a versao ponteiro porque é mais demorado indexar uma matriz do que usar o operador *.

Algumas vezes, programadores iniciantes na linguagem C cometem erros pensando que nunca devem usar a indexação de matrizes, já que ponteiros são muito mais eficientes. Mas não é o caso. Se a matriz será acessada em ordem estritamente ascendente ou descendente, ponteiros são mais rápidos e fáceis de usar. Entretanto, se a matriz será acessada aleatoriamente, a indexação da matriz pode ser uma melhor opção, pois geralmente será tão rápida quanto a avaliação de uma expressão complexa de ponteiros, além de ser mais fácil de codificar e entender.

Indexando um Ponteiro:

Em C, é possível indexar um ponteiro como se ele fosse uma matriz. Isso estreita ainda mais o relacionamento entre ponteiros e matrizes. Por exemplo, este fragmento de código é perfeitamente válido e imprime os números de 1 até 5 na tela:

```

/* Indexando um ponteiro semelhante a uma matriz */
#include <stdio.h>
main()
{
int i[5]={1, 2, 3, 4, 5};
int *p,t;
p=i;
for(t=0; t<5; t++) printf("%d ",*(p+t));
}

```

Em C, a declaração <code>p[t]</code> é idêntica a <code>*(p+t)</code> .

Ponteiros e Strings:

Uma vez que o nome de uma matriz sem um índice é um ponteiro para o primeiro elemento da matriz, o que está realmente acontecendo quando você usa as funções de string discutidas nos capítulos anteriores é

que somente um ponteiro para strings é passado para função, e não a própria string. Para ver como isso funciona, aqui está uma maneira como a função `strlen()` pode ser escrita:

```
strlen(char *s)
{
    int i=0;
    while(*s){
        i++;
        s++;
    }
    return i;
}
```

Quando uma constante string é usada em qualquer tipo de expressão, ela é tratada como um ponteiro para o primeiro caractere na string. Por exemplo, este programa é perfeitamente válido e imprime a frase "este programa funciona" na tela:

```
#include <stdio.h>
main()
{
    char *s;
    s= "este programa funciona";
    printf(s)
}
```

Obtendo o Endereço de um elemento da Matriz:

É possível atribuir o endereço de um elemento específico de uma matriz aplicando-se o operador `&` para uma matriz indexada. Por exemplo, este fragmento coloca o endereço do terceiro elemento de `x` em `p`:

```
p= &x[2];
```

Essa declaração é especialmente útil para encontrar uma substring. Por exemplo, este programa imprimirá o restante de uma string, que é inserida pelo teclado, a partir do ponto onde o primeiro espaço é encontrado:

```
#include <stdio.h>
/* Exibe a string à direita depois que o primeiro espaço é encontrado.*/
main()
{
    char s[80];
    char *p;
    int i;
    printf("digite uma string: ");
    gets(s);
    /* encontra o primeiro espaço ou o fim da string */
    for (i=0; s[i] && s[i]!=' '; i++);
    p=&s[i];
    printf(p);
}
```

Esse programa funciona, uma vez que `p` estará apontando para um espaço ou para um nulo (se a string não contiver espaços). Se há um espaço, o restante da string será impresso. Se há um nulo, a função `printf()` não imprime nada. Por exemplo, se "Olá a todos" for digitado, "a todos" será exibido.

Matrizes de Ponteiros:

Podem existir matrizes de ponteiros como acontece com qualquer outro tipo de dado. A declaração para uma matriz de ponteiros, do tipo int, de tamanho 10 é:

```
int *x[10];
```

Para atribuir o endereço de uma variável ponteiro chamada var ao terceiro elemento da matriz de ponteiros, você deve escrever:

```
x[2] = &var;
```

Para encontrar o valor de var, você deve escrever:

```
*x[2];
```

Um uso bastante comum de matrizes de ponteiros é armazenar ponteiros para mensagens de erros. Você pode criar uma função que emite uma mensagem, dado o seu número de código, como mostrado na função `serror()` abaixo:

```
serror(int num)
{
    char *err[]={
        "nao posso abrir o arquivo\n",
        "erro de leitura\n",
        "erro de escrita\n",
        "falha na mídia\n"
    };
    printf("%s",err[num]);
}
```

Como você pode ver, a função `printf()` é chamada dentro da função `serror()` com um ponteiro do tipo caractere, que aponta para uma das várias mensagens de erro indexadas pelo número do erro passado para a função. Por exemplo, se `num` é passado com valor 2, a mensagem de erro de escrita é exibida.

Uma outra aplicação interessante para matrizes de ponteiros de caracteres inicializadas usa a função de linguagem C `system()`, que permite ao seu programa enviar um comando ao sistema operacional. Uma chamada a `system()` tem esta forma geral:

```
system("comando");
```

onde `comando` é um comando do sistema operacional a ser executado, inclusive outro programa. Por exemplo, assumindo-se o ambiente DOS, esta declaração faz com que o diretório corrente seja exibido:

```
system("DIR");
```

O programa seguinte implementa uma pequena interface com o usuário dirigida por menu que pode executar quatro comandos do DOS: `DIR`, `CHKDSK`, `TIME` e `DATE`.

```
/* Uma interface simples do sistema do sistema operacional com o usuário
```

```

dirigida por menu.*/
#include <stdio.h>
#include<stdlib.h>
#include <conio.h>
main()
{

/* cria uma matriz de strings */
char *comando[] = {
"DIR",
"CHKDSK",
"TIME",
"DATE"
};
char ch;
for(;;){
do {
printf("1: diretório\n");
printf("2: verifica o disco\n");
printf("3: atualiza a hora\n");
printf("4: atualiza a data\n");
printf("5: sair\n");
printf("\nopção: ");
ch=getche();
} while ((ch<'1') || (ch>'5'));
if(ch=='5')break; /* fim*/
/* executa o comando específico */
printf("\n");
system(comando[ch-'1']);
}
}

```

Qualquer comando do DOS, inclusive a chamada a outros programas pode ser implementada desta maneira.

Simples Tradutor Inglês-Português:

A primeira coisa que você precisa é a tabela inglês-português mostrada aqui, entretanto, você é livre para expandi-la se assim o desejar.

```

char trans[][20] = {
"is", "é",
"this", "isto",
"not", "nao",
"a", "um",
"book", "livro",
"apple", "maça",
"I", "eu",
"bread", "pao",
"drive", "dirigir",
"to", "para",
"buy", "comprar",
"" , ""
}

```

Cada palavra em inglês é colocada em par com a equivalente em português. Note que a palavra mais longa não excede 19 caracteres.

A função main() do programa de tradução é mostrada aqui junto com as variáveis globais necessárias:

```
char entrada[80];
char palavra[80];
char *p;
main()
{
    int loc;
    printf("Informe a sentença em inglês: ");
    gets(entrada);
    p = entrada; /* dá a p o endereço da matriz entrada */
    printf("tradução rústica para o português: ");
    pega_palavra(); /* pega a primeira palavra */
    /* Este é o laço principal. Ele lê uma palavra por vez da matriz entrada e
    traduz para o português.*/
    do {
        /* procura o índice da palavra em inglês em trans */
        loc = verifica(palavra);
        /* imprime a palavra em português se uma correspondência é encontrada*/
        if(loc!=-1) printf("%s ", trans[loc+1]);
        else printf("<desconhecida> ");
        pega_palavra(); /* pega a próxima palavra /
    } while(*palavra); /*repete até que uma string nula é encontrada */
}
```

O programa opera desta maneira: primeiro, o usuário é solicitado a informar uma sentença em inglês. Essa sentença é lida na string entrada. O ponteiro p é, então, associado ao endereço do início de entrada e é usado pela função pega_palavra() para ler uma palavra por vez da string entrada e colocá-la na matriz palavra. O laço principal verifica, então, cada palavra, usando a função verifica(), e retorna o índice da palavra em inglês ou -1 se a palavra não estiver na tabela. Adicionando-se um a este índice, a palavra correspondente em português é encontrada.

A função verifica() é mostrada aqui:

```
/* Esta função retorna a localização de uma correspondência entre a string
apontada pelo parâmetro s e a matriz trans.*/
```

```
verifica(char *s)
{
    int i;
    for(i=0; *trans[i]; i++)
        if(!strcmp(trans[i], s)) break;
    if(*trans[i]) return i;
    else return -1;
}
```

Você chama verifica() com um ponteiro para a palavra em inglês e a função retorna o seu índice se ela estiver na tabela e -1 se não for encontrada.

A função `pega_palavra()` é mostrada a seguir. Da maneira como a função foi concebida, as palavras são delimitadas somente por espaços ou por terminador nulo.

/* Esta função lerá a próxima palavra da matriz entrada. Cada palavra é considerada como sendo separada por um espaço ou pelo terminador nulo. Nenhuma outra pontuação é permitida. A palavra retornada será uma string de tamanho nulo quando o final da string entrada é encontrado. */

```
pega_palavra()
{
    char *q;
    /* recarrega o endereço da palavra toda vez que a função é chamada*/
    q = palavra;
    /* pega a próxima palavra*/
    while(*p && *p!=' ') {
        *q = *p;
        p++;
        q++;
    }
    if(*p==' ')p++;
    *q = '\0'; /* termina cada palavra com um terminador nulo */
}
```

Assim que retorna da função `pega_palavra()` a variável global conterá ou a próxima palavra em inglês na sentença ou um nulo.

O programa completo da tradução é mostrado aqui:

```
/* Um tradutor (bastante) simples de inglês para português.*/
/* Sugestao: Colocar Watches em: entrada<->p , palavra<->q , loc */
```

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

char trans[][20] = {
    "is", "é",
    "this", "isto",
    "not", "nao",
    "a", "um",
    "book", "livro",
    "apple", "maça",
    "I", "eu",
    "bread", "pao",
    "drive", "dirigir",
    "to", "para",
    "buy", "comprar",
    "", ""
};
```

```
char entrada[80];
char palavra[80];
char *p;
main()
{
    int loc;
    printf("Informe a sentença em inglês: ");
```

```

gets(entrada);
p = entrada; /* dá a p o endereço da matriz entrada */
printf("tradução rústica para o português: ");
pega_palavra(); /* pega a primeira palavra */
/* Este é o laço principal. Ele lê uma palavra por vez da matriz entrada e
traduz para o português.*/
do {
    /* procura o índice da palavra em inglês em trans */
    loc = verifica(palavra);
    /* imprime a palavra em português se uma correspondência é encontrada*/
    if(loc!=-1) printf("%s ", trans[loc+1]);
    else printf("<desconhecida> ");
    pega_palavra(); /* pega a próxima palavra */
} while(*palavra); /*repete até que uma string nula é encontrada */
}

```

/* Esta função retorna a localização de uma correspondência entre a string apontada pelo parâmetro s e a matriz trans.*/

```

verifica(char *s)
{
    int i;
    for(i=0; *trans[i]; i++) if(!strcmp(trans[i], s)) break;
    if(*trans[i]) return i; /* se nao é o fim da matriz trans retorna i */
    else return -1; /* se é, retorna -1 (desconhecida) */
}

```

/* Esta função lerá a próxima palavra da matriz entrada. Cada palavra é considerada sendo separada por um espaço ou pelo terminador nulo. Nenhuma outra pontuação é permitida. A palavra retornada será uma string de tamanho nulo quando o final da string entrada é encontrado. */

```

pega_palavra()
{
    char *q;
    /* recarrega o endereço da palavra toda vez que a função é chamada*/
    q = palavra; /* palavra é global */
    /* pega a próxima palavra: */
    while(*p && *p!=' ') { /* p é global */
        *q = *p;
        p++;
        q++;
    }
    if(*p==' ')p++;
    *q = '\0'; /* termina cada palavra com um terminador nulo */
}

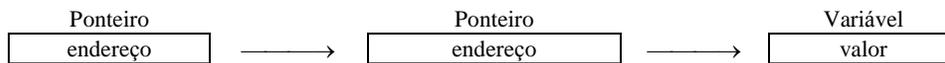
```

Ponteiros para Ponteiros:

Indireção simples:



Indireção múltipla:



Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. Isso é feito colocando-se um asterisco adicional na frente do seu nome. Por exemplo, esta declaração diz ao compilador que novobalanco é um ponteiro para um ponteiro do tipo float:

```
float **novobalanco;
```

É importante entender que novobalanco não é um ponteiro para um número de ponto flutuante, mas, antes, um ponteiro para um ponteiro float.

Para acessar o valor desejado apontado indiretamente por um ponteiro para um ponteiro, o operador asterisco deve ser aplicado duas vezes, como mostrado no pequeno exemplo a seguir:

```
#include <stdio.h>
main()
{
  int x, *p, **q;
  x = 10;
  p = &x;
  q = &p;
  printf("%d", **q); /* imprime o valor de x */
}
```

Aqui, p é declarado como um ponteiro para um inteiro e q como um ponteiro para um ponteiro para um inteiro. A chamada a printf() imprimirá o número 10 na tela.

Inicializando Ponteiros:

Depois que é declarado, mas antes que tenha sido associado a um valor, um ponteiro conterá um valor desconhecido. Se tentarmos usar um ponteiro antes de dar a ele um valor, provavelmente não só trará o programa como também o sistema operacional - um tipo de erro muito desagradável!

Por convenção, um ponteiro que está apontando para nenhum lugar deve ter valor nulo. Entretanto, somente o fato de um ponteiro ter um valor nulo não o torna "seguro". Se usarmos um ponteiro nulo no lado esquerdo de uma declaração de atribuição, correremos o risco de travar o programa e o sistema operacional.

Como um ponteiro nulo é considerado inútil, podemos usá-lo para tornar muitas rotinas de ponteiros fáceis de codificar e mais eficientes. Por exemplo, podemos usar um ponteiro nulo para marcar o final de uma matriz de ponteiros. Se isso é feito, a rotina que acessa aquela matriz saberá que chegou ao fim quando um valor nulo é encontrado. Esse tipo de abordagem é ilustrado pelo laço for mostrado aqui:

```
/* Observe um nome assumindo o último elemento de p como um nulo.*/
```

```
for(t=0; p[t]; ++t)
  if(!strcmp(p[t], nome)) break;
```

O laço será executado até que seja encontrada uma correspondência ou um ponteiro nulo. Uma vez que o fim da matriz está marcado com um nulo, a condição que controla o laço falhará quando ele for encontrado.

É uma prática muito comum em programas profissionais escritos na linguagem C inicializar strings. Vimos dois exemplos anteriores neste capítulo na seção de matrizes de ponteiros. Outra variação desse tema é o seguinte tipo de declaração de string:

```
char *p="Alô mundo\n";
```

Esse tipo de declaração coloca o endereço da string "Alô mundo" no ponteiro p. Por exemplo, o seguinte programa é perfeitamente válido:

```
#include <stdio.h>
#include <string.h>
char *p="Alô mundo";
main()
{
    register int i;
    /* imprime a string normal e inversa*/
    printf(p);
    for(t=strlen(p)-1; t>-1; t--) printf("%c",p[t]);
}
```

Problemas com Ponteiros:

Um problema com um ponteiro é difícil de ser encontrado. O ponteiro por si só não é um problema; o problema é que, cada vez que for realizada uma operação usando-o, pode-se estar lendo ou escrevendo para algum lugar desconhecido da memória. Se você ler, o pior que lhe pode acontecer é ler informação inválida, "lixo". Entretanto, se você escrever para o ponteiro, estará escrevendo sobre outros pedaços de código ou dados. Isso pode não ser mostrado mais tarde na execução do programa, levando-o a procurar pelo erro em lugar errado. Pode ser pouco ou não certo sugerir que o ponteiro é um problema. Esse tipo de erro tem feito com que programadores percam muitas noites de sono.

Tendo em vista que erros com ponteiros podem se transformar em pesadelos, você deve fazer o máximo para nunca gerar um! Alguns dos erros mais comuns são discutidos aqui, começando com o exemplo clássico de erro com ponteiro: o ponteiro-não-inicializado. Considere o seguinte:

```
/* Este programa está errado. Não o execute.*/
main()
{
    int x, *p;
    x = 10;
    *p = x;
}
```

Esse programa atribui o valor 10 a alguma localização desconhecida na memória. Ao ponteiro p nunca foi dado um valor; portanto ele contém um valor, "lixo". Embora o Turbo C emita uma mensagem de aviso neste exemplo, o mesmo tipo de problema surge quando um ponteiro está simplesmente apontado para um lugar indevido. Por exemplo, você pode acidentalmente atribuir um ponteiro para um endereço errado. Esse tipo de problema frequentemente passa despercebido quando o seu programa é muito pequeno, por causa da probabilidade de p conter um endereço "seguro" - um endereço que não esteja no seu código, na área de dados ou no sistema operacional. Entretanto, à medida que o seu programa cresce, a probabilidade de p apontar para algum lugar fundamental aumenta. Eventualmente, o seu programa pára de funcionar.

Para evitar esse tipo de problema, certifique-se sempre de que um ponteiro esteja apontado para alguma posição válida antes que seja usado.

Um segundo erro é causado pelo mal-entendido sobre como usar um ponteiro. Considere o seguinte:

```
#include <stdio.h>
/* Este programa está incorreto. Não o execute. */
main()
{
    int x, *p;
    x = 10;
    p = x;
    printf("%d", *p);
}
```

A chamada à função printf() não imprimirá o valor de x, que é 10, na tela. Imprimirá algum valor desconhecido. O motivo para isso é que a atribuição

```
p = x;
```

está errada. Aquela declaração atribui o valor 10 para o ponteiro p, que supostamente contém um endereço, não um valor. Para tornar o programa correto, você deve escrever:

```
p = &x;
```

Neste exemplo, o Turbo C avisará você sobre o erro no programa. Entretanto, nem todos os erros desse tipo geral podem ser verificados pelo compilador.

6) FUNÇÕES:

A forma Geral de uma Função:

A forma geral de uma função é:

```
especificador-de-tipo nome-da-função(declaração de parâmetros)
{
    corpo da função
}
```

O especificador de tipo determina o tipo de valor que a função retornará usando a declaração return. Ele pode ser qualquer tipo válido. Se nenhum tipo é especificado, então, por definição, a função retorna um resultado inteiro. A lista de declaração de parâmetros é uma lista de tipos de variáveis e nomes, separados por vírgula e que receberão os valores dos argumentos quando a função for chamada. Uma função pode estar sem parâmetro, caso em que a lista de parâmetros estará vazia (argumento void). Contudo, mesmo se não houver parâmetros, os parênteses ainda são requeridos.

A Declaração Return:

A declaração return tem dois importantes usos. Primeiro, ela fará com que haja uma saída imediata da função corrente. Isto é, uma declaração return fará com que a execução do programa retorne para o código chamador assim que ele é encontrado. Segundo, ela pode ser usada para retornar um valor. Esses dois recursos serão examinados aqui.

Retornando de uma Função:

Existem duas maneiras de uma função terminar a execução e retornar ao seu chamador. A primeira é quando a última declaração na função é executada e, conceitualmente, o finalizador da função } é encontrado.

Por exemplo, esta função simples imprime uma string inversa na tela:

```
void pr_inverso(char *s)
{
register int t;
for(t=strlen(s)-1; t>=0; t--) printf("%c", s[t]);
}
```

Uma vez que a string tiver sido exibida, não há nada para a função fazer a seguir, então ela retorna ao lugar de onde foi chamada.

A segunda maneira de uma função poder retornar é pelo uso da declaração return. A declaração return pode ser usada sem qualquer valor associado a ela. Por exemplo, a seguinte função imprime o resultado de um número elevado a uma potência positiva. Se o expoente é negativo, a declaração return faz com que a função termine antes que a chave final seja encontrada, mas nenhum valor é retornado.

```
void potência(int base, int exp)
{
int i;
if(exp<0) return; /* não pode calcular com expoente negativo */
i = 1;
for( ; exp-->0 i = base * i);
printf("A resposta é: %d ", i);
}
```

Retornando um Valor:

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

Note que uma função retorna um valor inteiro. Esse é o tipo de retorno padrão de uma função se nenhum outro é explicitamente especificado na definição da função.

É possível uma função conter duas ou mais declarações return. Por exemplo, a função max() é melhor escrita como mostrado aqui:

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

Outro exemplo de função que usa duas declarações return é a encontra_substr(), mostrada aqui. Ela retorna o índice de início de uma substring dentro de uma string ou -1, se nenhuma correspondência é

encontrada. Sem o uso da segunda declaração return, uma variável temporária e código extra precisariam ser usados.

```
encontra_substr(char *sub, char *str)
{
    register int t;
    char *p, *p2;
    for(t=0; str[t]; t++) {
        p = &str[t]; /* pega o ponto de início de str */
        p2 = sub;
        while(*p2 && *p2==*p) { /* enquanto nao for final de sub e for */
            p++; /* verificado igualdade entre os caracteres */
            p2++; /* de sub e str, avança. */
        }
        if(!*p2) return t; /* se está no final de sub, entao a correspondência foi
encontrada*/
    }
    return -1;
}
```

Se essa função é chamada com a substring contendo "dos" e a string principal contendo "Olá a todos", a função retornará o valor 8.

Todas as funções, exceto aquelas declaradas para serem do tipo void, retornam um valor. Esse valor é explicitamente especificado pela declaração return ou desconhecido, se nenhuma declaração return é especificada. Isso significa que uma função pode ser usada como um operando em qualquer expressao válida em C. Portanto, cada uma das seguintes expressoes sao válidas em C:

```
x = abs(y);
if(max(x,y) > 100) printf("excelente");
for(ch=getchar(); isdigit(ch);...;
```

Entretanto, uma função nao pode ser alvo de uma atribuição. Uma declaração como

```
swap(x, y)=100 ; /* declaração incorreta */
```

é errada. O turbo C sinalizará com um erro e nao compilará um programa que contenha tal declaração.

Funções Retornando Valores Nao-Inteiros:

Quando o tipo de uma função nao é explicitamente declarado, ele é automaticamente definido como int.

Como exemplo introdutório de uma função retornando um tipo nao-inteiro, aqui está um programa que usa a função sum(), que retorna um valor do tipo double que é a soma de seus dois argumentos double:

```
#include <stdio.h>

double sum(double a, double b); /* declaração inicial sempre necessária */
/* quando a função nao retorna int */

main()
{
    double primeiro, segundo;
```

```

primeiro = 1023.23;
segundo = 990.0;
printf("%lf", sum(primeiro, segundo)); /* %lf é o especificador p/ double */
}
double sum(double a, double b) /*retorna um float */
{
return a+b;
}

```

O protótipo antes de main() diz ao compilador que a função sum() retornará um tipo de dado double em ponto flutuante.

```
/* cálculo da área de um círculo */
```

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

```
float area(float raio); /* protótipo*/
```

```

main()
{
float r,res;
printf("Informe o raio: ");
scanf("%f", &r);
res=area(r);
printf("A área é: %f\n", res);
}
float area(float raio)
{
return M_PI * pow(raio,2); /* M_PI é amacro p/ o valor de pi em math.h */
}
/* pow(x,y) é igual ao x^y do BASIC e FORTRAN */
/* e é definida em math.h */

```

Obs1) Qualquer função em math.h devolve um valor double cujo valor máximo é 1.7E308. Estamos atribuindo o retorno de $\pi * \text{pow}(\text{raio}, 2)$ à variável float res (valor máximo = 3.4E38). Portanto devemos esperar overflow da função area para raios maiores que $1.04E19 = \sqrt{3.4E38/\pi}$. A função pow em si só acusará overflow para valores de retorno acima de 1.7E308 .

Obs2) A declaração dos argumentos no protótipo antes de main() é opcional.

Funções que retornam ponteiros:

Exemplos:

```

/* testa correspondencia entre uma string e um caractere */
#include<stdio.h>
char *corresp();

main()
{

```

```

char alfa[80];
char ch, *p;
int loc;

printf("\nDigite uma string: ");
gets(alfa);
printf("Digite um caractere: ");
ch=getche();
p= corresp(ch,alfa);
loc=strlen(alfa)-strlen(p)+1; /* dá a localização da correspondencia */

if(p) printf("\nExiste correspondencia no %dº caractere.",loc); /* existe correspondencia */
else printf("\nNenhuma correspondencia encontrada.");
}

/* Retorna um ponteiro p/ o primeiro caractere em s que corresponde a c: */
char *corresp(char c, char *s)
{
int count;
count=0;
while(c!=s[count]&&s[count] != '\0') count++; /* testa igualdade e nulo */
if(s[count] return (&s[count]); /* se há correspondencia retorna ponteiro */
else return (char *) '\0'; /* caso contrário retorna um ponteiro nulo */
}

```

Funções tipo void:

Funções que nao retornam valor sem especificação de void, farao o compilador emitir warning "Function should return a value". Nenhuma outra consequencia haverá.

```

#include<stdio.h>

void imprime_vertical(char *str);

void main()
{
char alfa[80];
printf("Digite uma string: ");
gets(alfa);
printf("\n\n");
imprime_vertical(alfa);
}
void imprime_vertical(char *str)
{
while(*str) printf("%c\n", *str++);
}

```

Blocos de código e variáveis locais:

```

void f(void)

{

char ch;
printf("continuar (s/n)? ");
ch=getche();

if(ch=='s'){
char s[80]; /* s só é criada como variável local de if se a resposta é 's' */
printf("informe o nome: "); /* Obs: s só é conhecido dentro do bloco de if */
gets(s);
process1(s); /* executa algum processo sobre a string s */
}
}

```

Chamando uma função por valor do argumento: (chamada por valor)

```

/* imprime o valor de pi e de pi^2 */
#include<stdio.h>
#include<math.h>

double sqr();

void main(void)

{
double pi=M_PI;
printf("%lf %lf",pi,sqr(pi));
}

double sqr(x) /* quando sqr() é chamada ela copia o valor de pi em x, deixan- */
/* do pi inalterado */
double x;

{
x=x*x;
return x;
}

```

Chamando uma função por referência ao argumento: (chamada por referência)

```

/* programa p/ demonstração da chamada de funções por referência */

#include <stdio.h>

void troca();

main(void)

{

```

```

double x,y;
printf("Digite dois números: ");
scanf("%f %f",&x,&y);
printf("Valores iniciais: x=%f y=%f\n",x,y);
troca(&x,&y);/* troca opera diretamente sobre os valores de x e y,alterando-os */
printf("Valores trocados: x=%f y=%f\n",x,y);

}

void troca(a,b)
double *a,*b;

{
double temp;
temp=*a; /* guarda em temp o valor contido no endereço de x */
*a = *b; /* coloca em x o valor de y */
*b = temp; /* coloca x em y */
}

```

Chamando funções com argumentos do tipo matriz:

Quando uma matriz é usada como argumento para uma função, somente o endereço da matriz é passado para a função e não a matriz inteira. Em outras palavras quando chamamos uma função com um argumento matriz, um ponteiro para o primeiro elemento da matriz é passado para a função. **LEMBRE:** Em C um nome de qualquer "array" (conjunto) sem índice é um ponteiro para o primeiro elemento do "array". Isto significa que a declaração do argumento no protótipo da função deve ser a de um ponteiro de tipo compatível.

Existem três maneiras de se declarar um argumento em uma função que irá receber um ponteiro de uma matriz.

Primeiro, o ponteiro-argumento pode ser declarado como uma matriz do mesmo tipo e tamanho daquela usada como argumento na chamada da função:

```

/* programa demonstração chamada de funções c/ matrizes */
/* imprime números 0 a 9 */
#include<stdio.h>

void display();
main(void)
{
int t[10],i;
for(i=0;i<10;++i) t[i]=i;
display(t);
}

void display(int num[10])
{
int i;
for(i=0;i<10;i++)printf("%d ",num[i]);
}

```

Ainda que o argumento `num` seja declarado uma matriz de 10 elementos, o compilador o converterá automaticamente em um ponteiro tipo `int`. Isto é necessário porque nenhum argumento a ser passado para uma função pode ser uma matriz com todos os seus elementos: quando uma função é chamada em `main()` todas as variáveis locais de `main()` são empurradas para a pilha, em seguida são empurrados os argumentos da função chamada e por último o endereço de retorno para `main()`. Então o programa continua a execução na função chamada. Como a pilha tem tamanho limitado, é óbvia a necessidade deste artifício. Uma vez que somente um ponteiro para a matriz será passado para a função, um argumento ponteiro deve estar declarado na função para receber os argumentos enviados da `main()` através da pilha.

A segunda maneira de declarar um parâmetro para uma matriz é especificando-o como uma matriz sem tamanho, como mostrado aqui:

```
void imprime(int num[ ])
{
    int i;
    for(i=0; i<10; i++) printf("%d ", num[ i ]);
}
```

Aqui, `num` é declarado como uma matriz de inteiros de tamanho desconhecido. Uma vez que a linguagem C não provê checagem de limites em matrizes, o tamanho real da matriz é irrelevante para o parâmetro (mas não para o programa, obviamente). Esse método de declaração define a variável `num` como um ponteiro inteiro.

O modo final como a variável `num` pode ser declarada, e a forma mais comum em programas profissionais escritos na linguagem C, é um ponteiro, como mostrado aqui:

```
void imprime(int *num)
{
    int i;
    for(i=0; i<10; i++) printf("d ", num[i]);
}
```

Isso é permitido já que qualquer ponteiro pode ser indexado usando-se `num[i]` como se fosse numa matriz, ou `*(num + i)`.

Reconheça que todos os três métodos de declaração de parâmetro matriz produzirão o mesmo resultado: um ponteiro.

É importante entender que, quando uma matriz é usada como argumento de função, seu endereço é passado para a função. Essa é uma exceção para a convenção de passagem de parâmetro por chamada de valor da linguagem C. Isso significa que o código na função estará operando sobre, e potencialmente alterando, o conteúdo atual da matriz usada para chamar a função. Por exemplo, considere a função `imprime_maiuscula()`, que imprime o seu argumento string em letra maiúscula.

```
/* Imprime uma string em letra maiúscula*/
```

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

```
void imprime_maiuscula();
```

```
main(void)
{
    char s[80];
```

```

printf("informe uma string: ");
gets(s);
imprime_maiuscula(s);
printf("\na string original é alterada: %s", s);
}
void imprime_maiuscula(char *string)
{
    register int t;
    for(t=0; string[t]; ++t) {
        string[t] = toupper(string[t]);
        printf("%c", string[t]);
    }
}

```

Depois de chamada à função `imprime_maiuscula()`, o conteúdo da matriz `s` em `main()` será mudado para letras maiúsculas. Se você não quer que isso aconteça, pode escrever assim:

```

/* Imprime uma string em letra maiúscula */

#include <stdio.h>
#include <ctype.h>

void imprime_maiuscula(char *string);
main(void)
{
    char s[80];
    printf("informe uma string: ");
    gets(s);
    imprime_maiuscula(s);
    printf("\na string original não é modificada: %s", s);
}
void imprime_maiuscula(char *string)
{
    register int t;
    for(t=0; string[t]; ++t) {
        printf("%c", toupper(string[t]));
    }
}

```

Nessa versão, o conteúdo da matriz `s` permanece inalterado uma vez que os seus valores são os mesmos.

Um exemplo clássico de passagens de matrizes em funções é encontrado na função de biblioteca `strcat()`. Ainda que `strcat()`, na biblioteca padrão, seja algo diferente, a função mostrada aqui lhe dará uma ideia de como ela funciona. Para evitar confusão com a função-padrão, ela será chamada `strcat2()`.

```

/* Demonstração da função strcat2() */

#include <stdio.h>

char *strcat2(char *s1, char *s2);

main(void)
{
    char s1[80], s2[80];
    printf("Informe duas strings:\n");
    gets(s1);
}

```

```

gets(s2);
strcat2(s1, s2);
printf("concatenado: %s", s1);
}
char *strcat2(char *s1, char *s2)
{
    char *temp;
    temp=s1;
    /* primeiro, encontra o final de s1 */
    while(*s1) s1++;
    /* adiciona s2*/
    while(*s2) {
        *s1=*s2;
        s1++;
        s2++;
    }
    *s1='\0'; /* acrescenta o terminador nulo */
    return temp;
}

```

A função strcat2() deve ser chamada com duas matrizes de caracteres, que, por definição, são ponteiros para caracteres. Logo na entrada, a função strcat2() busca o final da primeira string, onde acrescenta a segunda string. Depois, é colocado o terminador nulo na primeira string.

/* retornando mais de um valor de uma função: conversão polar <--> retangular
p/ números complexos complexos */

```

#include <stdlib.h>
#include <math.h>

double *rp();

main()
{
    char sign;
    char op;
    double *zo;
    double zi[2];
    double a,b;
    do{
        printf("R->P (r) ou P->R (p)? ");
        op=tolower(getche());
        printf("\n");
        switch(op)
            {
                case 'r':
                    printf("Z=Re+j*Im (entre no formato Re Im): ");
                    scanf("%lf %lf",&zi[0],&zi[1]);
                    zo = rp(zi,op);
                    printf("Z=Mag<Ang°= %lf<%lf°",zo[0],zo[1]);
                    break;

                case 'p':
                    printf("Z=Mag<Ang°= (entre no formato Mag Ang): ");
                    scanf("%lf %lf",&zi[0],&zi[1]);

```

```

zo = rp(zi,op);

if(zo[1]<0){sign='-';zo[1]=-zo[1];}
else{sign='+';}

printf("Z=Re+j*Im= %lf%cj*%lf",zo[0],sign,zo[1]);
break;
    }
} while(op!='r' && op!='p');

}

double *rp(zarg,op)
double *zarg;
char op;
{
double *zout;

zout=(double *)malloc(2*sizeof(double)); /* para zout é alocado 2*8 bytes */
if(!zout){exit(1);} /* se nao conseguiu alocar 16 bytes do heap, cai fora */

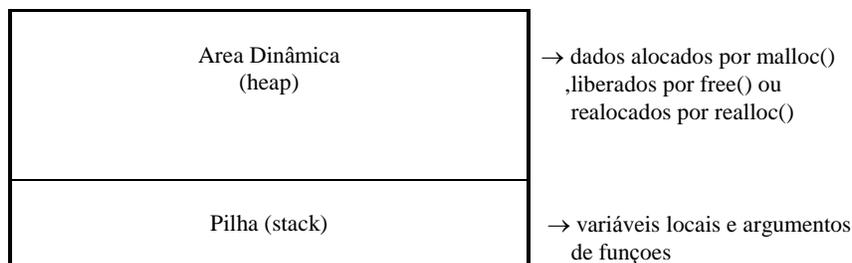
/* Observação: é obrigatório o uso de malloc(). Se declarássemos zout como
a matriz zout[2], perderíamos seus dados no retorno de *rp() porque esta fun-
ção retorna o endereço de uma variável que nao existe mais em main(). A única
maneira de manter um ponteiro existindo no retorno da main() é usando malloc()
que aloca permanentemente espaço no heap e nao na pilha */

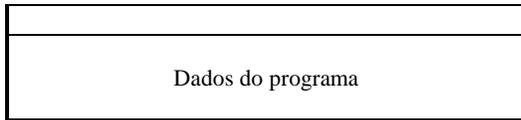
if(op=='r'){
zout[0]=sqrt((*zarg)*(*zarg)+*(zarg+1)**(zarg+1)); /* considerando zarg */
zout[1]=(180/M_PI)*atan2(*(zarg+1),(*zarg)); /* um ponteiro */
return zout;
}
if(op=='p'){

zout[0]= zarg[0]*cos((M_PI/180)*zarg[1]); /* considerando zarg uma matriz */
zout[1]= zarg[0]*sin((M_PI/180)*zarg[1]);
return zout;
}
puts("Erro!");
exit(1); /* erro de lógica: cai fora */
}

```

Obs1) O segmento de dados de 64Kb do modelo small é dividido como segue:





→ variáveis globais e static

Obs2) Uma variável static é uma variável local que mantém seu valor entre chamadas de funções, mas só é conhecida dentro da função que é declarada.

Obs3) sizeof() é um operador de tempo de compilação que retorna o tamanho em bytes do tipo de dado ou objeto usado como argumento.

Os Argumentos argc, argv e env para a Função main():

Algumas vezes, é muito útil passar informações para um programa quando ele é executado. O método geral é passar informações para a função main() por meio do uso de argumentos de linha de comando. Um argumento de linha de comando é a informação que segue o nome do programa na linha de comando do sistema operacional.

Existem três argumentos internos na função main(). Os dois primeiros, argc e argv, são usados para receber argumentos da linha de comando. O terceiro, env, é usado para acessar os parâmetros do ambiente do DOS que estão ativos quando o programa começa a execução. Estes são os únicos argumentos que main() pode ter.

O argumento argc armazena o número de argumentos digitados na linha de comando, e deve ser declarado como inteiro. Ele sempre será, no mínimo, o valor um, já que o próprio nome do programa é considerado como o primeiro argumento. O argumento argv é um ponteiro para uma matriz de strings. Cada elemento desta matriz unidimensional aponta para um argumento digitado na linha de comando.

Todos os argumentos de linha de comando são strings - qualquer número digitado deverá ser convertido por atof(), atoi() ou atol().

/ mostra os argumentos digitados e converte p/ double os dados numéricos */*

```
#include <stdio.h>
#include <process.h>
#include <math.h> /* necessario p/ atof() */

main(argc,argv)
int argc;
char *argv[]; /* equivalente a argv[][] */

{
int i,j;
char flag='v'/* indica: é numero */;
char str[80];

printf("argc=%d\n",argc);
for(i=0;i<argc;i++){
strcpy(str,argv[i]); /* copia argv[i] p/ str, mas argv[i] poderia ser impresso */
printf("argv[%d]=%s\n",i,str); /* diretamente em printf como uma string */
}

printf("\n Convertendo para double os argumentos numéricos:\n\n");
```

```

for(i=0;i<argc;i++){
flag='v';
for(j=0;j<strlen(argv[i]);j++) {
if(!isdigit(argv[i][j])&&argv[i][j]!='.'&&tolower(argv[i][j])!='e'){ flag='f';break;}
}
if(flag=='v'){
double x; /* x só existe neste bloco */
x=atof(argv[i]);
printf("argv[%d]=%lf\n",i,x);
}
}
}

```

Basicamente argc e argv são usados para se obter os argumentos iniciais para o programa a ser executado. Normalmente o DOS permite até 128 caracteres na linha de comando. Normalmente usa-se estes argumentos para indicar um arquivo a ser lido ou escrito, ou então uma opção (format a:, chkdsk /f, etc...).

Note que cada argumento da linha de comando deve ser separado do seguinte por um espaço ou uma tabulação (não são válidos a vírgula ou ponto e vírgula). Se for necessário passar um argumento de linha de comando contendo espaços, deve-se colocá-lo entre aspas.

O argumento env é declarado da mesma maneira que argv. Ele é um ponteiro para a matriz que contém as definições do ambiente do DOS (variáveis ambientais):

```

/* mostra o ambiente do DOS */

#include<stdio.h>

main(argc,argv,env)
int argc;
char *argv[];
char *env[];

{
int i;
for(i=0;env[i];i++)printf("%s\n",env[i]);
}

```

Retornando valores de main() para o DOS:

(usado no comando IF ERRORLEVEL do DOS: zero->sucesso não_zero->erro)

```

/* programa p/ executar comandos do DOS sequencialmente */

```

```

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

main(int argc, char *argv[])

{
int i;

```

```

for(i=1; i<argc; i++) {
if(system(argv[i])) {
printf("%s falha\n", argv[i]);
return -1
}
}

```

Funções recursivas:

```

/* determina o fatorial de um número inteiro RECURSIVAMENTE */
/* Vantagem: código menor se comparado com a versao nao recursiva */
/* Desvantagem: roda ligeiramente mais lenta (operacao com pilha) */
/* Obs) colocar watches em n n-1 e resposta para acompanhamento */

```

```

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

```

```

unsigned long int factr();

```

```

main()

```

```

{
unsigned long int n,f;

```

```

printf("Digite um número: ");
scanf("%ld",&n);
f=factr(n);
printf("O fatorial de %ld é %ld\n", n,f);
}

```

```

unsigned long int factr(n)
unsigned long int n;

```

```

{
unsigned long int resposta;
if(n==1||n==0)return(1);
resposta = factr(n-1)*n;
printf("%ld\n",resposta); /* opcional para visualizacao */
return(resposta);
}

```

```

/* determina o fatorial de um número inteiro NAO RECURSIVAMENTE */

```

```

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

```

```

unsigned long int factr();

```

```

main()

```

```

{

```

```

unsigned long int n,f;

printf("Digite um número: ");
scanf("%ld",&n);
f=factr(n);
printf("O fatorial de %ld é %ld\n", n,f);
}

```

```

unsigned long int factr(n)
unsigned long int n;

{
unsigned long int resposta , t;
for(t=1; t<n; t++) resposta = resposta*t;
return(resposta);
}

```

Quando uma função chama a si própria, as novas variáveis locais e os argumentos são alocados na pilha, e o código da função é executado com esses novos valores a partir do início. Uma chamada recursiva não faz uma nova cópia da função. Somente os argumentos e as variáveis são novos. Quando cada chamada recursiva retorna, as antigas variáveis locais e os parâmetros são removidos da pilha e a execução recomeça no ponto de chamada da função dentro da função dentro da função. Tendo em vista que o local para os argumentos de funções e para as variáveis locais é a pilha e que a cada nova chamada é criada uma cópia destas variáveis na pilha, é possível ocorrer overflow da pilha (stack overflow) e o programa trancar.

```

/* programa recursivo para geração de som em tonalidade crescente até 1KHz. */

```

```

#include <stdio.h>
#include <dos.h>

main()

{
int dur,salto;

printf("Digite a duração de cada ton em milissegundos: ");
scanf("%d",&dur);
printf("Digite o salto entre tons em Hz: ");
scanf("%d",&salto);
ton(1000,dur,salto);
return 0;
}

ton(freq,duracao,salto)
int freq,duracao,salto;

```

```

{
int i;
if(freq>0)
ton(freq-salto,duracao,salto);
sound(freq); /* liga o som */
delay(duracao); /* aguarda duracao milissegundos */
nosound();
}

```

7) ENTRADA E SAÍDA DO CONSOLE:

Lendo e escrevendo caracteres: getch(), getche(), getchar() e putchar():

As funções mais simples de E/S do console (teclado e vídeo) são `getche()`, que lê um caractere do teclado, e `putchar()`, que imprime um caractere na tela. A função `getche()` aguarda até que uma tecla seja pressionada e então retorna o seu valor. A tecla pressionada ecoa na tela automaticamente. A função `putchar()` escreve o seu argumento caractere na tela na posição corrente do cursor. Os protótipos são:

```

int getche(void); /* header: conio.h */
int putchar(int c); /* header: stdio.h */

```

Não nos perturbemos com o fato de `getche()` retornar um inteiro; o byte de mais baixa ordem contém o caractere e quando atribui-se um inteiro a uma variável `char` somente é transferido o byte mais baixo. Também a função `putchar()` embora seja declarada com um argumento inteiro, somente o byte mais baixo é enviado à tela.

Uma versão mais útil de `getche()` é `getch()` que não ecoa na tela o caractere digitado. O protótipo é idêntico ao de `getche()`.

```

/* inverte minúsculas<-->maiúsculas */
#include<stdio.h>
#include<conio.h>
#include<ctype.h>

main()
{
char ch;
printf("Insira caracteres e digite um ponto para parar.\n");
do{
ch=getch();
if(islower(ch))putchar(toupper(ch));
else putchar(tolower(ch));
} while(ch!='.');
return 0;
}

```

Uma versão antiga de `getche()`, ainda dirigida para ambientes UNIX, é `getchar()`, mas sofre de problemas com seu buffer intermediário em ambientes DOS iterativos. No entanto, permite o retorno do caractere 'CR' (Carriage Return = 'Enter').

Lendo e escrevendo strings: gets() e puts()

A função `gets()` lê uma string de caracteres entrada pelo teclado e coloca-a no endereço apontado pelo seu argumento ponteiro para caractere. Digita-se caracteres até que um 'Enter'(CR = Carriage Return) seja pressionado. O 'CR' não toma parte da string; em vez disto um terminador nulo '\0' é colocado no fim da sequência de caracteres e a função `gets()` retorna:

Protótipo: `char *gets(char *s);`
Header: `stdio.h`

onde `s` é uma matriz de caracteres que recebe os caracteres digitados. Observe que `gets()` retorna um ponteiro para `s`.

A função `puts()` escreve o seu argumento string para a tela seguido de "\n".

Protótipo: `int puts(char *s);`
Header: `stdio.h`

`puts()` reconhece os mesmos códigos de barra invertida de `printf()`, como o \t de tabulação. Uma chamada para `puts()` gera um código executável bem menor e mais rápido do que uma chamada a `printf()`.

Entrada e saída formatada do console: `scanf()` e `printf()`

O termo 'formatada' refere-se ao fato de que essas funções podem ler e escrever dados em vários formatos que estão sob seu controle.

`printf()`:

O protótipo de `printf()` é

```
int printf(char *string_de controle, lista_de_argumentos);
```

com header em `stdio.h`.

A string de controle consiste em dois tipos de itens. O primeiro tipo é o conjunto de caracteres que o usuário deseja que sejam impressos na tela. O segundo tipo é o conjunto de especificadores de formato, que definem a maneira como os argumentos são exibidos. Um especificador de formato começa com um sinal de porcentagem (%) e é seguido pelo código de formato. Deve haver o mesmo número de argumentos quantos forem os especificadores de formato, necessitando haver uma correspondência de tipo entre especificador de formato e argumento na ordem de ocorrência respectiva de cada um na string de controle e lista de argumentos:

```
printf("Olá %c %d %s", 'c', 10, "todos!");
```

mostra: Olá c 10 todos!

Especificadores de formato de <code>printf()</code> :	
Especificador:	Formato resultante na tela:
<code>%c</code>	Um único caractere
<code>%d</code>	Decimal inteiro
<code>%i</code>	Decimal inteiro
<code>%e</code>	Notação científica ('e' minúsculo: 6.22e17)
<code>%E</code>	Notação científica ('E' maiúsculo: 6.22E17)
<code>%f</code>	Decimal ponto flutuante

%g	Usa %e ou %f - o que ocupar menos espaço na tela
%G	Igual a %g só que usa 'E' maiúsculo no caso de acionar %e
%o	Octal inteiro
%s	String de caracteres
%u	Decimal inteiro sem sinal
%x	Hexadecimal inteiro (usando letras minúsculas)
%X	Hexadecimal inteiro (usando letras maiúsculas)
%%	Imprime o sinal %
%p	O endereço hexadecimal de 16 bits (deslocamento) para o qual aponta o argumento ponteiro associado dentro de um segmento de 64Kb (ponteiro near).
%Fp	O endereço hexadecimal de 20 bits para o qual aponta o argumento-ponteiro associado no formato segmento:deslocamento (ponteiro far).
%n	O argumento associado é um ponteiro para um inteiro onde é colocado o número de caracteres escritos até então (Ver exemplo abaixo)

```
/* Imprime o numero de caracteres impresso por printf() */
```

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

main()

{
char str[80];
int i;
gets(str);
printf("%s %n",str,&i); /* imprime str e 2 espaços */
printf("%d",i);
}
```

Os especificadores de formato podem ter modificadores que especificam o tamanho do campo, o número de casas decimais e um indicador de justificação a direita.

Modificadores:

Um inteiro colocado entre o sinal % e o caracterizador de formato atua como um especificador de largura mínima do campo. Este especificador preenche a saída com brancos ou zeros para assegurar que o campo esteja com pelo menos um certo comprimento mínimo. Se a string ou número é maior que o mínimo, será impressa por completo, mesmo se o mínimo for ultrapassado. O preenchimento padrão é feito com espaços. Se houver intenção de preencher com zeros, colocar um 0 antes do especificador de comprimento do campo. Por exemplo, %05d preenche um número de menos de cinco dígitos com zeros, de modo que o tamanho total seja cinco.

Para especificar o número de casas decimais impressas por um número em ponto flutuante, coloca-se um ponto decimal depois do especificador de tamanho de campo, seguido pelo número de casas decimais que se deseja mostrar. Por exemplo, %10.4f exibe um número de , no mínimo, dez caracteres de comprimento com quatro casas decimais. Quando isto é aplicado a strings ou inteiros, o número seguinte ao ponto especifica o comprimento máximo do campo. Por exemplo, %5.7s exibe uma string de no mínimo cinco caracteres e no máximo sete. Se a string a ser impressa exceder sete caracteres, será truncada.

Por definição, toda saída é justificada à direita: se a largura de campo é maior que o dado impresso, o dado será colocado na extremidade direita do campo. Pode-se forçar a informação a ser justificada à esquerda

colocando-se um sinal de menos (-) diretamente depois do %. Por exemplo, %-10.2f justificará a esquerda um número em ponto flutuante, com duas casas decimais, em um campo de 10 caracteres.

Existem dois modificadores dos especificadores de formato que permitem printf() exibir inteiros short e long. Esses modificadores podem ser aplicados aos especificadores de tipo d,i,o,u,x.

O modificador l indica que um dado tipo long será impresso na tela:

```
%ld long int
%lu unsigned long int
```

O modificador h indica que um dado tipo short int será impresso na tela:

```
%hu unsigned short int
%hd short int
%ho unsigned short int octal
```

O modificador l também pode alterar os especificadores e,f,g para indicar que um double será impresso:

```
%lf double em ponto flutuante
%Lf long double em ponto flutuante
%LE long double em notação científica c/ 'E' maiúsculo
/* Roda exemplos de especificadores de formato e campo de printf() */
```

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

```
main()
{
puts("012345678901234567890");
printf("%-7.2f\n",123.234);
printf("%7.2f\n",123.234);
printf("%-5.2f\n",123.234);
printf("%5.2f\n",3.324);
printf("%10s\n","Alô");
printf("%-10s\n","Alô");
printf("%5.7s\n","123456789");
}
```

scanf():

O protótipo de scanf() é

```
int scanf(char *string_de_de controle,lista_de_argumentos)
```

com header em stdio.h.

A string de controle consiste de três conjunto de caracteres:

- especificadores de formato
- caracteres de espaço em branco
- caracteres de espaço não branco

Os especificadores de formato de entrada são precedidos pelo sinal % e dizem a scanf() qual tipo de dado deve ser lido em seguida. A lista de argumentos contém uma lista de ponteiros para as variáveis-argumentos correspondentes em tipo, número e ordem de ocorrência aos especificadores de formato na string de controle.

Especificadores de formato de scanf():	
Especificador	Dado a ser lido:
%c	Um único caractere
%d	Decimal inteiro
%i	Decimal inteiro
%e ou %E	Notação científica
%f	Decimal ponto flutuante
%g ou %G	Lê %e ou %f
%o	Octal inteiro
%s	String de caracteres
%u	Decimal inteiro sem sinal
%x	Hexadecimal inteiro
%p	Lê o valor de um ponteiro
%Fp	Lê o valor de um ponteiro far
%n	O argumento associado é um ponteiro para um inteiro onde é colocado o número de caracteres lidos até então. Ver printf()
%le ou %lg	Idêntico ao caso sem o modificador l só que lê double em vez de float
%l seguido de d,i,o,u,x	Idêntico ao caso sem o modificador l só que lê long
%h seguido de d,i,o,u x	Idêntico ao caso sem o modificador h só que lê short

Caracteres de espaço em branco:

Um caractere de espaço em branco na string de controle faz com que a função scanf() salte um ou mais caracteres em branco no buffer de entrada.

Um caractere de espaço em branco pode ser:

- um espaço
- ou
- uma tabulação
- ou
- um newline ('Enter')

Essencialmente, um caractere de espaço em branco na string de controle fará com que a scanf() leia, mas não armazene, qualquer número de espaços em branco (incluindo zero, isto é, nenhum) de caracteres de espaço em branco até o primeiro caractere que não seja um espaço em branco. Por exemplo, "%d,%d" faz com que scanf() leia um inteiro, então leia e descarte uma vírgula e, finalmente, leia um outro inteiro. Se o caractere especificado não é encontrado scanf() termina prematuramente e não lê o segundo inteiro.

O programa a seguir usa o fato de que scanf() retorna o número de argumentos lidos com sucesso para demonstrar o efeito dos caracteres de espaço não branco na entrada de dados:

```
/* testa caracteres de espaço não branco em scanf() */
/* Digitar dois inteiros separados por # e depois separados por qualquer */
/* outro caractere */
```

```

#include<stdio.h>
main()
{
int i,j,c;

printf("Digite dois números inteiros separados por #: ");
c=scanf("%d#%d",&i,&j);/* c recebe o número de argumentos lidos por scanf() */

if(c==2)printf("Você digitou: %d e %d\n",i,j);
else printf("Insucesso. Lido somente %d campo por scanf(!",c);
}

```

Um * colocado depois do % e antes do código de formato lerá um dado de um tipo especificado, mas suprimirá sua atribuição. Assim,

```
scanf("%d%*c%d",&x,&y);
```

dando-se a entrada 10!20, colocará o valor 10 em x descartando o caractere '!' e dará a y o valor 20.

Os comandos de formato podem especificar um campo modificador de comprimento máximo. Esse modificador é um número inteiro colocado entre o sinal % e o código de comando de formato, que limita o número de caracteres lidos para qualquer campo. Por exemplo, para ler não mais do que 20 caracteres em str, escreve-se:

```
scanf("%20s",str);
```

Se o buffer de entrada contiver mais do que 20 caracteres, scanf() termina a leitura e só atribui a str os 20 primeiros caracteres deixando os restantes no buffer. Uma chamada subsequente a scanf() começará a leitura do buffer de entrada onde ela terminou. Por exemplo, se digitarmos

ABCDEFGHIJKLMNPNOPQRSTUVWXYZ

em resposta à chamada de scanf() deste exemplo, somente os 20 primeiros caracteres, até 'T', serão colocados em str. Na próxima chamada a scanf() serão colocadas em str a string "UVWXYZ":

```

/* demonstra modificador de comprimento máximo de campo de scanf() */
#include<stdio.h>
main()
{
char str[80];

puts("Digite abcdefghijklmnopqrstuvwxyz:");
scanf("%20s",str);
puts(str);
scanf("%20s",str); /* o programa não parará nesta chamada a scanf() porque */
puts(str); /* ainda haverá caracteres não transferidos no buffer de entrada */
}

```

Ainda que espaços, tabulações e newlines ('Enter') sejam usados como separadores de campos, quando da leitura de um único caractere, esses últimos são lidos como qualquer outro caractere. Por exemplo, entrando no teclado com "x y",

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

fará: a='x', b=' ' e c='y' .

Cuidado: se tivermos quaisquer outros caracteres na string de controle no exemplo anterior - incluindo espaços, tabulações e newlines - eles serão usados para correspondência-e-descarte de caracteres de entrada. Qualquer caractere que corresponder é descartado e se não corresponder scanf() termina:

```
/* digite xyz e depois x yz */
#include<stdio.h>
main()
{
char a,b,c;

puts("Digite 'xyz' na primeira tentativa e na segunda 'x yz'");
scanf("%ct%c%c",&a,&b,&c);

printf("a=%c b=%c c=%c",a,b,c);
}
```

Outro exemplo: No programa a seguir, scanf() não retornará até que seja digitado um caractere depois de digitar um caractere de espaço em branco. Isso ocorre em razão de espaço após o %s, que instruiu a scanf() a ler e descartar espaços, tabulações e newlines:

```
#include<stdio.h>
main()
{
char str[80];

puts("Digite uma string um espaço e um caractere. Depois não obedeça esta regra de entrada.");
scanf("%s ",str);
puts(str);
}
```

A função scanf() inclui também o recurso chamado 'scanset'. Scanset define um conjunto de caracteres que serão correspondidos por scanf(). A função scanf() continuará a ler caracteres enquanto eles estiverem no scanset. Assim que um caractere entrado não corresponder a nenhum caractere do scanset, segue para o próximo especificador de formato (se existir). Um scanset é definido colocando a lista de caracteres que se quer que seja examinada entre colchetes. Um sinal % deve anteceder o colchete inicial. Por exemplo, %[01234567] diz à scanf() para ler somente os dígitos de 0 a 7.

A variável-argumento correspondente a scanset deve ser um ponteiro para uma matriz de caracteres (isto é, uma string).

```
/* analisa os dados de entrada e converte os números para float */
```

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

main()
{
char s1[80];
char s2[80];
float x;

puts("Digite um número seguido de caracteres não numéricos: ");
```

```
scanf("%[123456789.eE]%s",s1,s2);
printf("Voce digitou o número %f\n",atof(s1));
printf("E a sequência nao numérica foi %s.",s2);
}
```

Podemos especificar um intervalo dentro de um scanset usando um hífen. Por exemplo, o código abaixo diz à scanf() para aceitar os caracteres de A a Z:

```
%[A-Z]
```

Podemos também especificar mais de um intervalo dentro de um scanset. O programa abaixo por exemplo lê dígitos e letras minúsculas. Ele demonstra também o uso de especificador de campo máximo (78 no caso) para scanset:

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

```
main()
{
char str[80];
puts("Informe dígitos e letras minúsculas:");
scanf("%78[a-z0-9]",str);
puts(str);
}
```

Scanset permite também a exclusão de conjuntos de caracteres mediante o especificador ^ colocado como primeiro caractere do scanset:

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

```
main()
{
char str[80];
puts("Informe dígitos e letras minúsculas excluindo a,b,c:");
scanf("%[^a-b]",str);
puts(str);
}
```

8) ENTRADA E SAÍDA DE DISCO:

O ponteiro de arquivo:

A linha comum que une o sistema de E/S de disco do Turbo C é o ponteiro de arquivo. Um ponteiro de arquivo é um ponteiro para uma área na memória(buffer) onde estão contidos vários dados sobre o arquivo a ler ou escrever, tais como o nome do arquivo, estado e posição corrente. O buffer apontado pelo ponteiro de arquivo é a área intermediária entre o arquivo no disco e o programa.

Este buffer intermediário entre arquivo e programa é chamado 'fluxo', e no jargão dos programadores é comum falar em funções que operam fluxos em vez de arquivos. Isto se deve ao fato de que um fluxo é uma entidade lógica genérica, que pode estar associada a uma unidade de fita magnética, um disco, uma porta serial, etc. Adotaremos esta nomenclatura aqui.

Um ponteiro de arquivo é uma variável-ponteiro do tipo FILE que é definida em stdio.h.

Para ler ou escrever em um arquivo de disco, o programa deve declarar uma (ou mais de uma se for necessário) variável ponteiro de arquivo. Para obter uma variável ponteiro de arquivo, usa-se uma declaração semelhante a esta ao declararmos as demais variáveis do programa:

FILE *fp;

onde fp é o nome que escolhemos para a variável (podia ser qualquer outro).

As Funções Mais Comuns do Sistema de Arquivo

Função	Operação
fopen()	Abre um fluxo
fclose()	Fecha um fluxo
putc()	Escreve um caractere para um fluxo
getc()	Lê um caractere para um fluxo
fseek()	Procura por um byte especificado no fluxo
fprintf()	É para um fluxo aquilo que printf() é para o console
fscanf()	É para um fluxo aquilo que scanf() é para o console
feof()	Retorna verdadeiro se o fim do arquivo é encontrado
ferror()	Retorna verdadeiro se ocorreu um erro
fread()	Lê um bloco de dados de um fluxo
fwrite()	Escreve um bloco de dados para um fluxo
rewind()	Reposiciona o localizador de posição de arquivo no começo do arquivo
remove()	Apaga um arquivo

Abrindo um Arquivo:

A função fopen() serve a dois propósitos. Primeiro, ela abre um fluxo para uso e liga um arquivo com ele. Segundo, retorna o ponteiro de arquivo associado àquele arquivo. Mais frequentemente, e para o resto desta discussão, o arquivo é um arquivo em disco. A função fopen() tem este protótipo:

```
FILE *fopen(char *nome_de_arquivo, char *modo);
```

onde modo é uma string contendo o estado desejado para abertura. O nome do arquivo deve ser uma string de caracteres que compreende um nome de arquivo válido para o sistema operacional e onde possa ser incluída uma especificação de caminho (PATH).

Como determinado, a função fopen() retorna um ponteiro de arquivo que não deve ter o valor alterado pelo seu programa. Se um erro ocorre quando se está abrindo um arquivo, fopen() retorna um nulo.

Como a Tabela abaixo mostra, um arquivo pode ser aberto ou em modo texto ou em modo binário. No modo texto, as seqüências de retorno de carro e alimentação de formulários são transformadas em seqüências de novas linhas na entrada. Na saída, ocorre o inverso: novas linhas são transformadas em retorno de carro e alimentação de formulário. Tais transformações não acontecem em um arquivo binário.

Os valores legais para modo.

Modo	Significado
"r"	Abre um arquivo para leitura
"w"	Cria um arquivo para escrita
"a"	Acrescenta dados para um arquivo existente
"rb"	Abre um arquivo binário para leitura
"wb"	Cria um arquivo binário para escrita
"ab"	Acrescenta dados a um arquivo binário existente

"r+"	Abre um arquivo para leitura/escrita
"w+"	Cria um arquivo para leitura/escrita
"a+"	Acrescenta dados ou cria um arquivo para leitura/escrita
"r+b"	Abre um arquivo binário para leitura/escrita
"w+b"	Cria um arquivo binário para leitura/escrita
"a+b"	Acrescenta ou cria um arquivo binário para leitura/escrita
"rt"	Abre um arquivo texto para leitura
"wt"	Cria um arquivo texto para escrita
"at"	Acrescenta dados a um arquivo texto
"r+t"	Abre um arquivo-texto para leitura/escrita
"w+t"	Cria um arquivo texto para leitura/escrita
"a+t"	Acrescenta dados ou cria um arquivo texto para leitura/escrita

Se você deseja abrir um arquivo para escrita com o nome test você deve escrever

```
FILE *ofp;
ofp = fopen("test", "w");
```

Entretanto, você verá frequentemente escrito assim:

```
FILE *out;
if((out=fopen("test", "w"))==NULL){
    puts("nao posso abrir o arquivo\n");
    exit(1);
}
```

A macro NULL é definida em STDIO.H. Esse método detecta qualquer erro na abertura do arquivo como um arquivo protegido contra escrita ou disco cheio, antes de tentar escrever nele. Um nulo é usado porque no ponteiro do arquivo nunca haverá aquele valor. Também introduzido por esse fragmento está outra função de biblioteca: exit(). Uma chamada à função exit() faz com que haja uma terminação imediata do programa, não importando de onde a função exit() é chamada. Ela tem este protótipo (encontrado em STDLIB.H):

```
void exit(int val);
```

O valor de val é retornado para o sistema operacional. Por convenção, um valor de retorno 0 significa término com sucesso para o DOS. Qualquer outro valor indica que o programa terminou por causa de algum problema (retornado para a variável ERRORLEVEL em arquivos "batch" do DOS).

Se você usar a função fopen() para abrir um arquivo, qualquer arquivo preexistente com o mesmo nome será apagado e o novo arquivo iniciado. Se não existirem arquivos com o nome, então será criado um. Se você quiser acrescentar dados ao final do arquivo, deve usar o modo "a". Para se abrir um arquivo para operações de leitura é necessário que o arquivo já exista. Se ele não existir, será retornado um erro. Finalmente, se o arquivo é aberto para escrita/leitura, as operações feitas não apagarão o arquivo, se ele existir; entretanto, se não existir, será criado.

Escrevendo um caractere:

A função putc() é usada para escrever caracteres para um fluxo que foi previamente aberto para escrita pela função fopen(). A função é declarada como

```
int putc(int ch, FILE *fp);
```

onde fp é o ponteiro de arquivo retornado pela função fopen() e ch, o caractere a ser escrito. Por razões históricas, ch é chamado formalmente de int, mas somente o caractere de mais baixa ordem é usado.

Se uma operação com a função `putc()` for satisfatória, ela retornará o caractere escrito. Em caso de falha, um EOF é retornado. EOF é uma macrodefinição em `STDIO.H` que significa fim do arquivo.

Lendo um caractere:

A função `getc()` é usada para ler caracteres do fluxo aberto em modo de leitura pela função `fopen()`. A função é declarada como

```
int getc(FILE *fp)
```

onde `fp` é um ponteiro de arquivo do tipo `FILE` retornado pela função `fopen()`.

Por razões históricas, a função `getc()` retorna um inteiro, porém o byte de mais alta ordem é zero.

A função `getc()` retornará uma marca EOF quando o fim do arquivo tiver sido encontrado ou um erro tiver ocorrido. Portanto, para ler um arquivo-texto até que a marca de fim de arquivo seja mostrada, você poderá usar o seguinte código:

```
ch=getc(fp);
while(ch!=EOF) {
    ch=getc(fp);
}
```

Usando a função feof():

O sistema de arquivo do Turbo C também pode operar com dados binários. Quando um arquivo é aberto para entrada binária, é possível encontrar um valor inteiro igual à marca de EOF. Isso pode fazer com que, na rotina anterior, seja indicada uma condição de fim de arquivo, ainda que o fim de arquivo físico não tenha sido encontrado. Para resolver esse problema, foi incluída a função `feof()` que é usada para determinar o final de um arquivo quando da leitura de dados binários. A função `feof()` tem este protótipo:

```
int feof(FILE *fp);
```

O seu protótipo está em `STDIO.H`. Ela retorna verdadeiro se o final do arquivo tiver sido encontrado; caso contrário, é retornado um zero. Portanto, a seguinte rotina lê um arquivo binário até que o final do arquivo seja encontrado.

```
while(!feof(fp))ch=getc(fp);
```

Obviamente, o mesmo método pode ser aplicado tanto a arquivos textos como a arquivos binários.

Fechando um arquivo:

A função `fclose()` é usada para fechar um fluxo que foi aberto por uma chamada à função `fopen()`. Ela escreve quaisquer dados restantes na área intermediária (fluxo) no arquivo e faz um fechamento formal em nível de sistema operacional do arquivo. Uma falha no fechamento de um arquivo leva a todo tipo de problemas, incluindo-se perda de dados, arquivos destruídos e a possibilidade de erros intermitentes no seu programa. Uma chamada à função `fclose()` também libera os blocos de controle de arquivo (FCB) associados ao fluxo e deixa-os disponíveis para reutilização.

Como você provavelmente sabe, o sistema operacional limita o número de arquivos abertos que você pode ter em um dado momento, de modo que pode ser necessário fechar alguns arquivos antes de abrir outros. (Comando `FILES=` no `autoexec.bat`).

A função `fclose()` é declarada como:

```
int fclose(FILE*fp);
```

onde `fp` é o ponteiro do arquivo retornado pela chamada à função `fopen()`. Um valor de retorno igual a zero significa uma operação de fechamento com sucesso; qualquer outro valor indica um erro. Você pode usar a função padrão `ferror()` (discutida a seguir) para determinar e reportar quaisquer problemas. Geralmente, o único momento em que a função `fclose()` falhará é quando um disquete tiver sido prematuramente removido do drive ou não houver mais espaço no disco.

`ferror()` e `rewind()`:

A função `ferror()` é usada para determinar se uma operação em um arquivo produziu um erro. Se um arquivo é aberto em modo texto e ocorre um erro na leitura ou na escrita, é retornado EOF. Você usa a função `ferror()` para determinar o que aconteceu. A função `ferror()` tem o seguinte protótipo:

```
int ferror(FILE*fp);
```

onde `fp` é um ponteiro válido para um arquivo. `ferror()` retorna verdadeiro se um erro ocorreu durante a última operação com o arquivo e falso, caso contrário. Uma vez que cada operação em arquivo determina uma condição de erro, a função `ferror()` deve ser chamada imediatamente após cada operação com o arquivo; caso contrário, um erro pode ser perdido. O protótipo de função `ferror()` está no arquivo `STDIO.H`.

A função `rewind()` recolocará o localizador de posição do arquivo no início do arquivo especificado como seu argumento. O seu protótipo é:

```
void rewind(FILE*fp);
```

onde `fp` é um ponteiro de arquivo. O protótipo para a função `rewind()` está no arquivo `STDIO.H`.

Usando `fopen()`, `getc()`, `putc()` e `fclose()`:

```
/* Grava em disco dados digitados no teclado até teclar $ */

#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    if(argc!=3) {
        puts("Você se esqueceu de informar o nome do arquivo e o modo!");
        exit(1);
    }
    if((fp=fopen(argv[1], argv[2]))==NULL){ /* experimentar "wb"/"wt" e testar newline */
        puts("O arquivo não pode ser aberto!");
        exit(1);
    }
    do { /* getche() não funcionaria aqui porque getchar() */
        ch=getchar(); /* pega o caracteres do teclado vai armazenando no buffer até o newline. */
        if(EOF==putc(ch, fp)){
            puts("Erro ao escrever no arquivo!");
            break;
        }
    }
}
```

```

    } while (ch!='$');
fclose(fp);
return 0; /* código de retorno de final OK p/ o DOS. OPCIONAL */
}

/*Lê arquivos e exhibe-os na tela.*/

#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
FILE *fp;
char ch;
if(argc!=3) {
puts("Você se esqueceu de informar o nome do arquivo e o modo!");
exit(1);
}
if((fp=fopen(argv[1], argv[2]))==NULL){ /* experimentar "rb"/"rt" e testar newline */
puts("O arquivo nao pode ser aberto!");
exit(1);
}
ch= getc(fp); /* lê um caractere */
while(ch!=EOF){
putchar(ch); /* imprime na tela */
ch=getc(fp);
}
return 0;
}

```

getw() e putw():

Funcionam exatamente como putc() e getc() só que em vez de ler ou escrever um único caractere, lêem ou escrevem um inteiro.

Protótipos:

```
int putw(int i, FILE *fp);
int getw(FILE *fp);
```

Header: stdio.h

Exemplo:

```
putw(100,saida);
```

escreve o inteiro 100 no arquivo apontado por saida.

fgets() e fputs():

Funcionam como gets() e puts() só que lêem e escrevem em fluxos.

Protótipos:

```
char *fputs(char *str, FILE *fp);
```

```
char *fgets(char *str, int tamanho, FILE *fp);
```

Obs) fgets() lê uma string de um fluxo especificado até que um newline seja lido OU tamanho-1 caracteres tenham sido lidos. Se um newline é lido, ele fará parte da string (diferente de gets()). A string resultante termina com um nulo.

Header: stdio.h

fread() e fwrite():

Lêem e escrevem blocos de dados em fluxos.

Protótipos:

```
unsigned fread(void *buffer, int num_bytes, int count, FILE *fp);
```

```
unsigned fwrite(void *buffer, int num_bytes, int count, FILE *fp);
```

Header: stdio.h

No caso de fread(), buffer é um ponteiro para uma área da memória que receberá os dados lidos do arquivo.

No caso de fwrite(), buffer é um ponteiro para uma área da memória onde se encontram os dados a serem escritos no arquivo.

Buffer usualmente aponta para uma matriz ou estrutura(a ser visto adiante).

O número de bytes a ser lido ou escrito é especificado por num_bytes.

O argumento count determina quantos itens (cada um tendo num_bytes de tamanho) serão lidos ou escritos.

O argumento fp é um ponteiro para um arquivo de um fluxo previamente aberto por fopen().

A função fread() retorna o número de itens lidos, que pode ser menor que count caso o final de arquivo (EOF) seja encontrado ou ocorra um erro.

A função fwrite() retorna o número de itens escritos, que será igual a count exceto na ocorrência de um erro de escrita.

Quando o arquivo for aberto para dados binários, fread() e fwrite() podem ler e escrever qualquer tipo de informação. O programa a seguir escreve um float em um arquivo de disco chamado teste.dat:

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

```
main()
{
FILE *fp;
float f=M_PI;
```

```
if((fp=fopen("teste.dat","wb"))==NULL){
puts("Nao posso abrir arquivo!");
```

```

exit(1);
    }
if(fwrite(&f, sizeof(float), 1, fp)!=1) puts("Erro escrevendo no arquivo!");
fclose(fp);
return 0;
}

```

Como o programa anterior ilustra, a área intermediária de armazenamento pode ser (e frequentemente é) simplesmente uma variável.

Uma das aplicações mais úteis de fread() e fwrite() é o armazenamento e leitura rápidos de matrizes e estruturas (estrutura é uma entidade lógica a ser vista adiante) em disco:

```

/* escreve uma matriz em disco */
#include<stdio.h>
#include<stdlib.h>

main()
{

FILE *fp;
float exemplo[10][10];
int i,j;

if((fp=fopen("exemplo.dat","wb"))==NULL){
puts("Nao posso abrir arquivo!");
exit(1);
    }

    for(i=0; i<10; i++){
for(j=0; j<10; j++){
exemplo[i][j] = (float) i+j; /* lei de formação dos elementos da matriz */
    }
    }

/* o código a seguir grava a matriz inteira em um único passo: */

if(fwrite(exemplo, sizeof(exemplo), 1, fp)!=1)
{ puts("Erro ao escrever arquivo!");exit(1);}
fclose(fp);
return 0;
}

/* lê uma matriz em disco */

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

main()
{

FILE *fp;

```

```

float exemplo[10][10];
int i,j;

if((fp=fopen("exemplo.dat","rb"))==NULL){
puts("Nao posso abrir arquivo!");
exit(1);
}

/* o código a seguir lê a matriz inteira em um único passo: */

if(fread(exemplo, sizeof(exemplo), 1, fp)!=1)
{ puts("Erro ao ler arquivo!");exit(1);}

for(i=0; i<10; i++){
for(j=0; j<10; j++) printf("%3.1f ", exemplo[i][j]);
printf("\n");
}

fclose(fp);
return 0;
}

```

Acesso randômico a arquivos: fseek()

A função `fseek()` indica o localizador de posição do arquivo.

Protótipo:

```
int fseek(FILE *fp, long numbytes, int origem);
```

Header: `stdio.h`

- `fp` é um ponteiro para o arquivo retornado por uma chamada à `fopen()`.
- `numbytes` (long int) é o número de bytes a partir da origem até a posição corrente.
- `origem` é uma das seguintes macros definidas em `stdio.h`:

Origem	Nome da Macro	Valor numérico
começo do arquivo	<code>SEEK_SET</code>	0
posição corrente	<code>SEEK_CUR</code>	1
fim do arquivo	<code>SEEK_END</code>	2

Portanto, para procurar `numbytes` do começo do arquivo, origem deve ser `SEEK_SET`. Para procurar da posição corrente em diante, origem é `SEEK_CUR`. Para procurar `numbytes` do final do arquivo de trás para diante, origem é `SEEK_END`.

O seguinte código lê o 235º byte do arquivo chamado `teste`:

```

FILE *fp;
char ch;

if((fp=fopen("teste","rb"))==NULL){

```

```
puts("Nao posso abrir arquivo!");
exit(1);          }
```

```
fseek(fp,234,SEEK_SET);
ch=getc(fp); /* lê um caractere na posição 235º */
```

A função `fseek()` retorna zero se houve sucesso ou um valor não-zero se houve falha no posicionamento do localizador de posição do arquivo.

```
/* utilitário de visualização de disco usando fseek() */
/* visualiza setores de 128 de bytes do disco e apresenta em ASCII e hexadecimal */
/* digite -1 para deixar o programa */
```

```
#include <stdio.h>
#include <ctype.h> /* isprint() necessita */
#include <stdlib.h>
```

```
#define TAMANHO 128 /* TAMANHO = 128 */
```

```
char buf[TAMANHO]; /* global */
```

```
void display();
```

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

```
{
FILE *fp;
```

```
long int setor, numlido;
```

```
if(argc!=2){puts("Uso: tmp nome_do_arquivo");exit(1);} /* esqueceu de digitar */
/* o nome de arquivo */
```

```
if((fp=fopen(argv[1],"rb"))==NULL){
puts("Nao posso abrir arquivo!");
exit(1);          }
```

```
for(;;) {
printf("Informe o setor (-1 p/ terminar): ");
scanf("%ld",&setor);
if(setor<0) break;
```

```
if(fseek(fp, setor*TAMANHO , SEEK_SET)){puts("Erro de busca!");exit(1);}
```

```
if((numlido=fread(buf,1,TAMANHO,fp)) != TAMANHO){puts("EOF encontrado!");}
```

```
display(numlido);
}
```

```
return 0;
}
```

```
void display(long int numlido)
{
```

```
long int i,j;
```

```

for(i=0; i<=numlido/16 ; i++) { /* controla a indexação de linhas: 0 a 8 (128/16=8) */
for(j=0; j<16; j++) printf("%2X ", buf[i*16+j]); /* imprime 16 col. em hexa */
printf(" "); /* separa mostrador hexa do ascii */
for(j=0; j<16; j++) {
    /* imprime 16 col. em ascii: (só imprimíveis) */
    if(isprint(buf[i*16+j]))printf("%c",buf[i*16+j]);
    else printf(".");
}
printf("\n");
}
}

```

Os Fluxos-Padros:

Toda vez que um programa começa a execução, são abertos 5 fluxos padrões:

```

stdin (aponta p/ o teclado se não redirecionado pelo DOS. Ex: MORE < FILE.TXT)
stdout (aponta p/ a tela se não redirecionado pelo DOS. Ex : DIR > PRN)
stderr (recebe mensagens de erro - aponta para a tela)
stdprn (aponta p/ a impressora)
stdaux (aponta p/ a porta serial)

```

Para entender o funcionamento destes fluxos, note que putchar() é definida em stdio.h como:

```
#define putchar(c) putc(c,stdout)
```

e a função getchar() é definida como

```
#define getchar() getc(stdin)
```

Ou seja, estes fluxos permitem serem lidos e escritos como se fossem fluxos de arquivos. Toda entrada de dado de um programa é feita por stdin e toda saída por stdout:

```

/* localiza uma palavra especificada pela linha de comando em um arquivo
redirecionado para stdin, mostra a linha e seu número */
/* rodar no prompt do DOS o seguinte comando: tmp argv < tmp.c */

```

```

#include<string.h> /* strstr() necessita */
#include<stdio.h>

```

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

```
{
```

```
char string[128];
int line=0;
```

```
while(fgets(string, sizeof(string),stdin))
```

```

{
++line;
if(strstr(string, argv[1])) printf("%02d %s",line,string);
}
}

```

O fluxo stderr é usado para mostrar mensagens de erro na tela, quando a saída do programa esta redirecionada p/ outro dispositivo que nao seja a tela:

```
#include <stdio.h>
main(int argc, char *argv[])
{
FILE *fp;
int count;
char letter;

if(argc!=2){puts("Digite o nome do arquivo!");exit(1);}
if(!(fp = fopen(argv[1],"w"))){
    fputs("Erro na abertura do arquivo",stderr);
/* 2ª opção: puts("Erro na abertura do arquivo"); -> vai p/ stdout,isto é,
tela ou outro dispositivo */
    exit(1);
}
else for(letter='A';letter<='Z'; letter++) putc(letter, fp);
fclose(fp);
}
```

Se a saída deste programa (stdout) for redirecionada para algum outro arquivo, a mensagem de erro forçosamente aparecerá na tela porque estamos escrevendo em stderr.

fprintf() e fscanf():

Comportam-se como printf() e scanf() só que escrevem e lêem de arquivos de disco. Todos os códigos de formato e modificadores sao os mesmos.

Protótipo:

```
int fprintf(FILE *fp, char *string_de_controle, lista_de_argumentos);
```

```
int fscanf(FILE *fp, char *string_de_controle, lista_de_argumentos);
```

Header: stdio.h

Embora fprintf() e fscanf() sejam a maneira mais fácil de ler e escrever tipos de dados nos mais diversos formatos, elas nao sao as mais eficientes em termos de tamanho de código resultante e velocidade. Quando o formato de leitura e escrita for de importância secundária, deve-se dar preferência a fread() e fwrite().

```
/* imprime os quadrados de 0 a 10 no arquivo quad.dat no formato
número - quadrado */
```

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

```
main()
{
int i;
FILE *out;
```

```
if((out=fopen("quad.dat","wt"))==NULL){ /* sempre texto c/ fscanf() e fprintf() */
puts("Nao posso abrir arquivo!");
```

```

exit(1);          }

for (i=0; i<=10; i++){
fprintf(out,"%d  %d\n", i , i*i);
}

fclose(out);
}

```

Deixaremos o exemplo de uso de fscanff() para o próximo capítulo (Alocação Dinâmica).

Apagando arquivos: remove()

```
int remove(char *nome_arquivo);
```

Retorna zero em caso de sucesso e nao zero se falhar.

9) ALOCAÇÃO DINÂMICA DA MEMÓRIA:

Existem três funções básicas de gerenciamento dinâmico da memória:

malloc()	aloca determinado tamanho de memória
free()	libera a memória alocada por malloc() p/ ser reutilizada
realloc()	encolhe ou expande o bloco de memória alocado por malloc(), se necessário move-o para caber no espaço disponível

Protótipos: (header em stdlid.h)

```
void *malloc (unsigned numbytes);
```

malloc() retorna um ponteiro void, isto é, ele pode apontar p/ o que se desejar: uma matriz, estrutura, número. numbytes() normalmente é calculado por sizeof(). Se nao houver memória suficiente retorna um ponteiro nulo.

```
void free(void *p);
```

free() libera a memória alocada a partir do ponteiro gerado por malloc(). NUNCA use free() p/ liberar o bloco de memória apontada por um ponteiro que nao tenha sido gerado por malloc(), ou a memória usada será perdida.

```
void *realloc(void *p, unsigned tamanho);
```

realloc() altera o tamanho da memória alocada por p para o tamanho especificado por 'tamanho'. Se nao houver memória suficiente retorna um ponteiro nulo e libera (executa free()) o bloco original.

```
/* lê um arquivo numérico de nome a especificar no formato de 2 colunas */
```

```

#include<dos.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>
#include<alloc.h>

```

```
main(argc,argv)
```

```

int argc;
char *argv[];
{
FILE *in;
float *x, *y;
float tempx, tempy;
int i, teste,numlido;

if(argc!=2){puts("Digite o nome do arquivo!");exit(1);}

if((in=fopen(argv[1],"rt"))==NULL){ /* sempre texto c/ fscanf() e fprintf() */
puts("Nao posso abrir arquivo!");
exit(1);
}

x=(float*)malloc(sizeof(float));
if(!x){LBLMEM:puts("Memória insuficiente!");exit(1);}
y=(float*)malloc(sizeof(float));
if(!y){goto LBLMEM;}

i=0;
while(1){
teste=fscanf(in,"%f%f",&tempx,&tempy);
if(teste==-1)break; /* testa EOF: fscanf() retorna -1 quando encontra-o */
*(x+i)=tempx; /* para fugir de erro de linkagem do compilador - BUG da versao 1.0 do Turbo C++ */
*(y+i)=tempy;
i++;

x=realloc(x,((unsigned)(i+1))*sizeof(float));
if(!x){LBLALOC:puts("Erro de alocação!");exit(1);}
y=realloc(y,((unsigned)(i+1))*sizeof(float));
if(!y){goto LBLALOC;}
}
fclose(in);
numlido=i-1;
for(i=0;i<=numlido;i++)printf("%f %f\n",*(x+i),*(y+i));
}

```

10) ESTRUTURAS:

Em C, uma estrutura é uma coleção de variáveis referenciada sob um nome, provendo um meio de manter informações relacionadas juntas. Uma declaração de estrutura permite que se crie variáveis de estruturas. As variáveis que compreendem uma estrutura são chamadas de elementos da estrutura:

```

struct endereco {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
};

```

Obs1) O nome "endereco" é chamada etiqueta da estrutura

Obs2) Até aqui não foi alocada nenhuma memória

```

struct endereco info_adr;

```

```
/* declara a variável info_adr como sendo do tipo estrutura endereco */
/* somente agora é alocada memória */
```

Outra maneira, alocando memória já na declaração da estrutura:

```
struct endereco {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
}info_adr, binfo, cinfo;
```

```
/*Isto definirá um tipo de estrutura chamada endereco e declarará as variáveis
info_adr, binfo, cinfo como sendo deste tipo.*/
```

Se você necessita somente de uma variável de um determinado tipo, a etiqueta não é necessária:

```
struct {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
}info_adr;
```

```
/* declara uma variável chamada info_adr como definida pela estrutura que a
precedeu */
```

A forma geral de uma declaração de estrutura é:

```
struct nome_do_tipo_da_estrutura {

tipo nome_de_elemento_1;
tipo nome_de_elemento_2;
tipo nome_de_elemento_3;
.
.
.
tipo nome_de_elemento_N;
}variáveis_estruturura;
```

onde o nome do tipo da estrutura (etiqueta) ou as variáveis da estrutura pode ser omitido, mas não os dois.

Referenciando os membros da estrutura: operador ponto

nome_da_variável_estrutura.nome_do_elemento

Exemplo:

```
info_adr.cep=91720260; /* atribui */
printf("%lu", info_adr.cep); /* imprime cep na tela */
```

```
printf("%s",info_adr.nome); /* imprime nome */  
for(t=0;info_adr.nome[t];++t)putchar(info_adr.nome[t]); /* idem caracter por caractere */
```

Matrizes de estruturas:

```
struct endereco info_adr[100]; /* cria 100 conjuntos de variáveis que são */  
/* organizadas como definido na estrutura */  
/* endereco */  
  
printf("%lu", info_adr[2].cep); /* imprime na tela o cep da estrutura 3 da */  
/* matriz de 100 estruturas */
```

Matrizes de estruturas são muito comuns em planilhas de cálculo tipo Lotus, em que se armazena uma grande quantidade de dados encadeados.

Um exemplo de uso de estruturas: uma lista de endereços simples

Para ajudar a mostrar como as estruturas e matrizes de estruturas são usadas, um programa simples de listas de endereços que usa uma matriz de estruturas para armazenar a informação do endereço será desenvolvido. As funções neste programa interagem com as estruturas e seus elementos para ilustrar o uso de estruturas.

Neste exemplo, a informação que será armazenada inclui:

```
nome  
rua  
cidade  
estado  
cep
```

Para definir a estrutura de dados básica, endereco, que guardará esta informação, você poderá escrever:

```
struct endereco{  
char nome[40];  
char rua[40];  
char cidade[30];  
char estado[3];  
char cep[10];  
}info_adr[tamanho];
```

Note que essa estrutura usa um string para armazenar o código do cep em vez de um inteiro sem sinal. Essa abordagem acomoda tanto os códigos postais com letras como aqueles com números, como os usados pelo Canadá e outros países. A matriz info_adr contém TAMANHO estruturas do tipo endereco, onde TAMANHO pode ser definido para satisfazer uma necessidade específica. Fica aqui a sugestão que se tente reescrever este programa utilizando as funções de alocação dinâmica do Turbo C no sentido de automatizar a alocação de memória para a matriz info_adr.

A primeira função necessária ao programa é main() mostrada aqui:

```
main(void)
```

```

{
char opcao;

inic_list();

for(;;) {
opcao=menu();
switch(opcao) {
case 'i': insere();
break;
case 'e': exhibe();
break;
case 'c': carrega();
break;
case 's': salva();
break;
case 't': return 0;
}
}
}

```

Primeiro, a função `inic_list()` prepara a matriz de estrutura para uso, colando um caractere nulo no primeiro byte do campo nome de cada estrutura na matriz. O programa assume que uma variável estrutura não está em uso se o campo nome está vazio.

A função `inic_list()` é escrita como mostrado aqui:

```

/* Inicializa a matriz info_adr.*/
void inic_list(void)
{
register int t;

for(t=0; t<TAMANHO; t++) *info_adr[t].nome='\0';
/* um nome com comprimento zero significa vazio*/
}

```

A função `menu()` exibirá as opções e retornará à opção do usuário:

```

/* Obtém uma seleção no menu.*/
menu(void)
{

char ch;

do{
printf("(I)nsere\n");
printf("(E)xibe\n");
printf("(C)arrega\n");
printf("(S)alva\n");
printf("(T)ermina\n\n");
printf("sua opção:");
ch= getche();
printf("\n\n");
} while(!strchr("iecost", tolower(ch)));
return tolower(ch);
}

```

```
}
```

Essa função faz uso de outra função de biblioteca do Turbo C, strchr(), que tem este protótipo:

```
char *strchr(char *str,char ch);
```

Essa função pesquisa na string apontada por str uma ocorrência do caractere em ch. Se o caractere é encontrado, um ponteiro para ele é retornado. Por definição, é um valor verdadeiro. Contudo, se nenhuma correspondência for encontrada, um nulo é retornado, que por definição é falso.

Essa função é usada neste programa para ver se o usuário informa uma opção válida. A função insere() solicita ao usuário informar os dados e então coloca a informação passada na primeira estrutura livre. Se a matriz estiver cheia, a mensagem "lista cheia" é impressa na tela.

```
/*Insere nomes na lista.*/
void insere (void)
{
    register int i;
    /* encontra a primeira estrutura livre*/
    for(i=0; i<TAMANHO; i++)
        if(!*info_adr[i].nome)break;

    /* i será igual a tamanho se a lista estiver cheia.*/
    if(i==TAMANHO){
        printf("lista cheia\n");
        return;
    }
    /* Insere a informação*/
    printf("nome: ");
    gets(info_adr[i].nome);

    printf("rua: ");
    gets(info_adr[i].rua);

    printf("cidade: ");
    gets(info_adr[i].cidade);

    printf("estado: ");
    gets(info_adr[i].estado);

    printf("cep: ");
    gets(info_adr[i].cep);

}

```

As rotinas salva() e carrega() mostradas aqui são usadas para salvar e carregar o banco de dados da lista de endereços. Note a pouca quantidade de código em cada rotina por causa do poder das funções fread() e fwrite():

```
/* Salva a lista.*/
void salva(void)
{
    FILE *fp;
    register int i;

```

```

if((fp=fopen("listend.dat","wb"))==NULL){
    printf("o arquivo nao pode ser aberto\n");
    return;
}

for(i=0; i<TAMANHO; i++)
    if(*info_adr[i].nome)
        if(fwrite(&info_adr[i], sizeof(struct endereco), 1, fp)!=1)
            printf("erro de escrita no arquivo\n");
fclose(fp);
}

/* Carrega o arquivo.*/
void carrega (void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("listend.dat","rb"))==NULL){
        printf("o arquivo nao pode ser aberto\n");
        return;
    }

    inic_list();
    for(i=0; i<TAMANHO; i++)
        if(fread(&info_adr[i], sizeof(struct endereco), 1, fp)!=1){
            if(feof(fp)){
                fclose(fp);
                return;
            }
            printf("erro de leitura no arquivo\n");
        }
}

```

As duas rotinas confirmam uma operação com sucesso no arquivo checando o valor retornado pela função fread() ou fwrite(). Ainda, a função carrega() deve checar explicitamente o fim do arquivo pelo uso da função feof(), uma vez que a função fread() retorna o mesmo valor se o fim do arquivo é encontrado ou se tiver ocorrido um erro.

A função final que o programa necessita é exibe(). Ela imprime a lista de endereços inteira na tela:

```

/* Exibe a lista.*/
void exibe(void)
{
    register int t;

    for(t=0; t<TAMANHO; t++){
        if(*info_adr[t].nome){
            printf("Nome: %s\n", info_adr[t].nome);
            printf("Rua: %s\n", info_adr[t].rua);
            printf("Cidade: %s\n", info_adr[t].cidade);
            printf("Estado: %s\n", info_adr[t].estado);
            printf("CEP: %s\n\n", info_adr[t].cep);
        }
    }
}

```

```
}
```

A listagem completa do programa Lista de Endereços é mostrada aqui:

```
/* uma lista de endereços simples utilizando uma matriz de estruturas */
```

```
#include<conio.h>
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define TAMANHO 100

struct endereco{
char nome[40];
char rua[40];
char cidade[30];
char estado[3];
char cep[10];
}info_adr[TAMANHO];

void insere(void), inic_list(void), exhibe(void);
void salva(void), carrega(void);
int menu(void);

main(void)
{
char opcao;

inic_list();

for(;;) {
opcao=menu();
switch(opcao) {
case 'i': insere();
break;
case 'e': exhibe();
break;
case 'c': carrega();
break;
case 's': salva();
break;
case 't': return 0;
}
}
}

/* Inicializa a matriz info_adr.*/
void inic_list(void)
{
register int t;

for(t=0; t<TAMANHO; t++) *info_adr[t].nome='\0';
/* um nome com comprimento zero significa vazio*/
}
```

```

/* Obtém uma seleção no menu.*/
menu(void)
{

char ch;

do{
printf("(I)nsere\n");
printf("(E)xibe\n");
printf("(C)arrega\n");
printf("(S)alva\n");
printf("(T)ermina\n\n");
printf("sua opção:");
ch= getch();
printf("\n");
} while(!strchr("iecst", tolower(ch)));
return tolower(ch);
}

/*Insere nomes na lista.*/
void insere (void)
{
register int i;
/* encontra a primeira estrutura livre*/
for(i=0; i<TAMANHO; i++)
if(!*info_adr[i].nome)break;

/* i será igual a tamanho se a lista estiver cheia.*/
if(i==TAMANHO){
printf("lista cheia\n");
return;
}
/* Insere a informação*/
printf("nome: ");
gets(info_adr[i].nome);
printf("rua: ");
gets(info_adr[i].rua);
printf("cidade: ");
gets(info_adr[i].cidade);
printf("estado: ");
gets(info_adr[i].estado);
printf("cep: ");
gets(info_adr[i].cep);
}

/* Exibe a lista.*/
void exibe(void)
{
register int t;

for(t=0; t<TAMANHO; t++){
if(*info_adr[t].nome){
printf("Nome: %s\n", info_adr[t].nome);
printf("Rua: %s\n", info_adr[t].rua);
}
}
}

```

```

        printf("Cidade: %s\n", info_adr[t].cidade);
        printf("Estado: %s\n", info_adr[t].estado);
        printf("CEP: %s\n\n", info_adr[t].cep);
    }
}

/* Salva a lista.*/
void salva(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("listend.dat","wb"))==NULL){
        printf("o arquivo nao pode ser aberto\n");
        return;
    }

    for(i=0; i<TAMANHO; i++)
        if(*info_adr[i].nome)
            if(fwrite(&info_adr[i], sizeof(struct endereco), 1, fp)!=1)
                printf("erro de escrita no arquivo\n");
    fclose(fp);
}

/* Carrega o arquivo.*/
void carrega (void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("listend.dat","rb"))==NULL){
        printf("o arquivo nao pode ser aberto\n");
        return;
    }

    inic_list();
    for(i=0; i<TAMANHO; i++)
        if(fread(&info_adr[i], sizeof(struct endereco), 1, fp)!=1)
        {
            if(feof(fp)){
                fclose(fp);
                return;
            }
            printf("erro de leitura no arquivo\n");
        }
}

```

Atribuindo Estruturas:

```

#include<stdio.h>

main(void)
{
struct exemplo {
int i;
double d;
    } um,dois;

um.i=10;
um.d=98.6;
dois=um; /* atribui a estrutura um a estrutura dois */
printf("%d %lf", dois.i, dois.d);
return 0;
}

```

Obs) Não é possível atribuir uma estrutura a outra se elas são de tipos diferentes.

Passando estruturas para funções:

Passando elementos de estrutura para funções:

Seja

```

struct {
char x;
int y;
float z;
char s[10];
    } exemplo;

```

Chamando funções por valor:

```

func(exemplo.x); /* passa p/ func() o valor do char em x */
func2(exemplo.y); /* passa p/ func2() o valor do int em y */
func3(exemplo.z); /* passa p/ func3() o valor do float em z */
func4(exemplo.s); /* passa p/ func4() o endereço da string em s */
func(exemplo.s[2]); /* passa p/ func() o valor do char em s[2] */

```

Chamando funções por referência:

```

func(&exemplo.x); /* passa p/ func() o endereço do char em x */
func2(&exemplo.y); /* passa p/ func2() o endereço do int em y */
func3(&exemplo.z); /* passa p/ func3() o endereço do float em z */
func4(exemplo.s); /* passa p/ func4() o endereço da string em s */
func(&exemplo.s[2]); /* passa p/ func() o endereço do char em s[2] */

```

Observe que o operador & precede o nome da variável estrutura, não o nome do elemento individual. Note também que o elemento string s já significa um endereço, assim o operador & não é requerido.

Passando estruturas inteiras para funções:

Quando uma estrutura é usada como argumento para uma função, a estrutura inteira é passada usando-se o método padrão de chamada por valor. Isto significa, obviamente, que qualquer mudança feita no conteúdo de uma estrutura dentro da função para qual ela é passada, não afeta a estrutura usada como argumento.

A consideração mais importante para se ter em mente quando se usa uma estrutura como argumento é que os tipos devem ser compatíveis na chamada e na declaração da função. Este programa declara os argumentos `arg` e `parm` para serem do mesmo tipo de estrutura:

```
#include<stdio.h>

/* define um tipo de estrutura: */

struct exemplo {
    int a,b;
    char ch;
}; /* nenhuma memória alocada até aqui */

void fl();

main(void)
{
    struct exemplo arg; /* aloca memória para arg */

    arg.a=1000;
    fl(arg);
    return 0;
}

void fl(parm)
struct exemplo parm;

{
    printf("%d",parm.a);
}
```

Ponteiros para estruturas:

Existe uma desvantagem maior em passar tudo, até a mais simples estrutura para funções: a sobrecarga da pilha implicando em perda de tempo na transferência de dados entre uma função chamadora e função chamada. Além do perigo óbvio do overflow da pilha (stack overflow). Em estruturas simples, com poucos elementos, esse esforço não é tão importante, mas se muitos elementos são usados e se algum deles é matriz, a velocidade de execução pode ser completamente degradada ao utilizarmos chamada de funções por valor. A solução é passar para a função chamada somente um ponteiro para a estrutura argumento. Quando um ponteiro para estrutura é passado para uma função, somente o endereço da estrutura é passado para a função: somente o endereço da estrutura é empurrado para a pilha. Isto significa que uma chamada de função extremamente rápida pode ser executada. Também, como estaremos referenciando a estrutura argumento em si e não uma cópia na pilha, a função chamada terá condições de modificar o conteúdo dos elementos da estrutura argumento usada na chamada.

Para encontrar o endereço de uma variável estrutura, o operador `&` é colocado antes do nome da variável estrutura:

```

struct sal {
float saldo;
char nome[80];
    } cliente;

```

```

struct sal *ptr; /* declara um ponteiro para estruturas tipo sal */

```

```

ptr= &cliente; /* coloca o endereço de cliente em ptr */

```

Para acessar o elemento saldo através de ptr temos duas sintaxes:

```

float x,y;

```

```

x= (*ptr).saldo /* atribui a x o saldo de cliente */

```

```

y= ptr->saldo /* atribui a y o saldo de cliente */

```

A segunda sintaxe de acesso usa o chamado operador seta e é universalmente preferida em relação a primeira.

Seguem dois exemplos de programas, um chamando funções por valor e outro por referência a estruturas de dados. Ambos utilizam a palavra reservada #typedef que define um novo nome para um tipo de dado já existente. Não confundir com #define que atribui a uma sequência de caracteres um identificador:

```

#define ERRO "Impossível abrir arquivo\n" /* atribui o identificador ERRO à */
/* sequência de caracteres que o segue */

```

```

puts(ERRO);

```

```

/-----/

```

```

typedef struct cliente_tipo {
float divida;
int atraso;
char nome[40];
char tipo_divida[80];
    } cliente;

```

```

cliente lista_clientes[NUM_CLIENTES] /* define uma matriz de estruturas do */
/* tipo cliente chamada lista_clientes */

```

```

/* Obs) cliente não é uma variável do tipo cliente_tipo mas sim um outro nome
para a struct cliente_tipo ! */

```

```

puts(lista_clientes[4].nome);/*imprime na tela o 5º nome da lista de clientes*/

```

```

/* Programa de operações simples c/ complexos para demonstração */
/* da passagem de estruturas p/ funções com chamada por valor */

```

```

#include<dos.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>

```

```

typedef struct z { /* a etiqueta z é opcional */

```

```

double re;
double im;
    } cpx;

typedef struct zplus { /* a etiqueta zplus é opcional */

double re;
double im;
double mag;
double ang;
    } cpxp;

cpxp soma();
cpxp subt();
cpxp mult();
cpxp divi();
cpxp polar();

main()
{
    char op,sign;
    cpx z1,z2;
    cpxp zout;

    do{
        printf("\nQual operação com complexos é desejada? [+ - * /] ");
        op=getche();
        }while(op!='+'&&op!='-'&&op!='*'&&op!='/' );

        puts("\n\nInforme dois complexos Z1=R1+j*I1 e Z2=R2+j*I2 entrando");
        puts("na seguinte forma: R1 I1 R2 I2 ");
        scanf("%lf %lf %lf %lf",&z1.re,&z1.im,&z2.re,&z2.im);

        switch(op) {

        case '+':
            zout = soma(z1,z2);
            break;

        case '-':
            zout = subt(z1,z2);
            break;

        case '*':
            zout = mult(z1,z2);
            break;

        case '/':
            zout = divi(z1,z2);
            break;

        default:
            puts("Erro!");

```

```

        exit(1);
    }

    if(zout.im<0){ sign='-'; zout.im= -(zout.im);} /* sign dá o sinal da */
    else{ sign='+';} /* parte imaginária de */
    /* zout no formato da printf */

    printf("\nZ1 %c Z2 = %lf %c j*%lf =",op,zout.re,sign,zout.im);
    printf(" %lf<%lf°",zout.mag,zout.ang);

}

cpxp soma(arg1,arg2)
cpx arg1,arg2;
{
    cpxp res;

    res.re = arg1.re + arg2.re;
    res.im = arg1.im + arg2.im;
    res = polar(res);
    return res;
}

cpxp subt(arg1,arg2)
cpx arg1,arg2;
{
    cpxp res;

    res.re = arg1.re - arg2.re;
    res.im = arg1.im - arg2.im;
    res = polar(res);
    return res;
}

cpxp mult(arg1,arg2)
cpx arg1,arg2;
{
    cpxp res;

    res.re = (arg1.re)*(arg2.re)-(arg1.im)*(arg2.im);
    res.im = (arg1.im)*(arg2.re)+(arg1.re)*(arg2.im);

    /* (a+jb)(c+jd) = (ac-bd) + j(bc+ad) */

    res = polar(res);
    return res;
}

cpxp divi(arg1,arg2)
cpx arg1,arg2;
{
    cpxp res;

    res.re = (arg1.re)*(arg2.re)+(arg1.im)*(arg2.im);
    res.im = (arg1.im)*(arg2.re)-(arg1.re)*(arg2.im);
}

```

```

        res.re = res.re/(arg2.re*arg2.re + arg2.im*arg2.im);
        res.im = res.im/(arg2.re*arg2.re + arg2.im*arg2.im);

        /* (a+jb)/(c+jd) = ((ac+bd) + j(bc-ad))/(c^2 + d^2) */

        res = polar(res);
        return res;
    }

cpxp polar(arg)
cpxp arg;

{
    cpxp res;
    res.mag = sqrt((arg.re)*(arg.re)+(arg.im)*(arg.im));
    res.ang =(180/M_PI)*atan2(arg.im,arg.re);
    res.re = arg.re;
    res.im = arg.im;
    return res;
}

/-----/

/* Programa de operaçoes simples c/ complexos para demonstracao */
/* da passagem de estruturas p/ funçoes com chamada por referència */

#include<dos.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>

typedef struct z { /* a etiqueta z é opcional */

double re;
double im;
    } cpx;

typedef struct zplus { /* a etiqueta zplus é opcional */

double re;
double im;
double mag;
double ang;
    } cpxp;

cpxp soma();
cpxp subtr();
cpxp mult();
cpxp divi();
cpxp polar();

main()
{

```

```

char op,sign;
cpx z1,z2;
cpxp zout;

do{
printf("\nQual operação com complexos é desejada? [+ - * /] ");
op=getche();
} while(op!='+'&&op!='-'&&op!='*'&&op!='/');

puts("\n\nInforme dois complexos Z1=R1+j*I1 e Z2=R2+j*I2 entrando");
puts("na seguinte forma: R1 I1 R2 I2 ");
scanf("%lf %lf %lf %lf",&z1.re,&z1.im,&z2.re,&z2.im);

switch(op) {

case '+':
zout = soma(&z1,&z2);
break;

case '-':
zout = subt(&z1,&z2);
break;

case '*':
zout = mult(&z1,&z2);
break;

case '/':
zout = divi(&z1,&z2);
break;

default:
puts("Erro!");
exit(1);
}

if(zout.im<0){sign='-'; zout.im= -(zout.im);} /* sign dá o sinal da */
else{sign='+';} /* parte imaginária de */
/* zout no formato da printf */

printf("\nZ1 %c Z2 = %lf %c j*%lf =",op,zout.re,sign,zout.im);
printf(" %lf<%lf°",zout.mag,zout.ang);
}
cpxp soma(arg1,arg2)
cpx *arg1,*arg2;
{
cpxp res;

res.re = arg1->re + arg2->re;
res.im = arg1->im + arg2->im;
res = polar(&res);
return res;
}

```

```
cpxp subt(arg1,arg2)
```

```
cpx *arg1,*arg2;
```

```
{
```

```
    cpxp res;
```

```
    res.re = arg1->re - arg2->re;
```

```
    res.im = arg1->im - arg2->im;
```

```
    res = polar(&res);
```

```
    return res;
```

```
}
```

```
cpxp mult(arg1,arg2)
```

```
cpx *arg1,*arg2;
```

```
{
```

```
    cpxp res;
```

```
    res.re = (arg1->re)*(arg2->re)-(arg1->im)*(arg2->im);
```

```
    res.im = (arg1->im)*(arg2->re)+(arg1->re)*(arg2->im);
```

```
    /* (a+jb)(c+jd) = (ac-bd) + j(bc+ad) */
```

```
    res = polar(&res);
```

```
    return res;
```

```
}
```

```
cpxp divi(arg1,arg2)
```

```
cpx *arg1,*arg2;
```

```
{
```

```
    cpxp res;
```

```
    res.re = (arg1->re)*(arg2->re)+(arg1->im)*(arg2->im);
```

```
    res.im = (arg1->im)*(arg2->re)-(arg1->re)*(arg2->im);
```

```
    res.re = res.re/(arg2->re*arg2->re + arg2->im*arg2->im);
```

```
    res.im = res.im/(arg2->re*arg2->re + arg2->im*arg2->im);
```

```
    /* (a+jb)/(c+jd) = ((ac+bd) + j(bc-ad))/(c^2 + d^2) */
```

```
    res = polar(&res);
```

```
    return res;
```

```
}
```

```
cpxp polar(arg)
```

```
cpxp *arg;
```

```
{
```

```
    cpxp out;
```

```
    out.mag = sqrt((arg->re)*(arg->re)+(arg->im)*(arg->im));
```

```
    out.ang = (180/M_PI)*atan2(arg->im,arg->re);
```

```
    out.re = arg->re;
```

```
    out.im = arg->im;
```

```
    return out;
```

```
}
```

Matrizes dentro de estruturas:

```
struct x {
int a[10][10]; /* matriz de 100 inteiros */
float b;
}y;

valor = y.a[3][7]; /* atribui a valor o elemento (3,7) da matriz a da
                estrutura y */
```

Estruturas dentro de estruturas (estruturas aninhadas):

```
struct endereco {
char nome[30];
char rua[40];
char cidade[20];
char estado[3];
unsigned long int cep;
};

struct empregado {
struct endereco residencia; /* residencia é uma struct do tipo endereco */
float salario; /* aninhada na struct funcionario que é do */
} funcionario; /* tipo empregado */

funcionario.residencia.cep = 90500657; /* atribui o cep 90500657 para o campo
                cep da residencia do funcionario */
```

Resumindo: Os elementos de cada estrutura são referenciados da esquerda para a direita do mais externo para o mais interno.

11) CAMPOS DE BITS:

Ao contrário de muitas outras linguagens, a linguagem C tem um modo próprio de acessar um ou mais bits dentro de um byte ou palavra. Isto pode ser útil por diversas razões: 1º) se o espaço é limitado pode-se armazenar muitas variáveis Boolean (V/F) em um byte ; 2º) certas interfaces de dispositivos transmitem informações codificadas em bits dentro de um byte ; 3º) certas rotinas de codificação de dados necessitam acessar os bits dentro de um byte. Um dos métodos que a linguagem C usa para acessar bits baseia-se na estrutura chamada campo-bit. Um campo-bit é na realidade, somente um tipo de membro de estrutura que define o quão grande, em bits, cada elemento deve ser.

A forma geral de uma declaração campo-bit é feita como segue:

```
struct nome_do_tipo_estrutura {

    tipo nome_1: tamanho;
    tipo nome_2: tamanho;
    tipo nome_3: tamanho;
    .
    .
    .
    tipo nome_N: tamanho;
}
```

Um campo-bit deve ser declarado como int unsigned ou signed. Campos-bit de tamanho 1 devem ser declarados como unsigned, já que um único bit não pode ter sinal.

Exemplo aplicativo:

A função biosequip() retorna uma lista dos periféricos instalados no computador por meio de um valor de 16 bits codificado como mostrado aqui:

```
int biosequip(void); /* header em bios.h */
```

```
-----
```

Bit:	Periférico:
0	obriga o boot através do drive de disco flexível
1	coprocessador matemático presente
2,3	tamanho da ram na placa mae:
	0,0: 16Kb
	0,1: 32Kb
	1,0: 48Kb
	1,1: 64Kb
4,5	modo inicial de vídeo:
	0,0: não usado
	0,1: colorido ou BW 40x25
	1,0: colorido ou BW 80x25
	1,1: monocromático 80x25
6,7	número de drives de disco flexível:
	0,0: 1
	0,1: 2
	1,0: 3
	1,1: 4
8	chip DMA instalado
9,10,11	número de portas seriais:
	0,0,0: 0
	0,0,1: 1
	0,1,0: 2
	0,1,1: 3
	1,0,0: 4
	1,0,1: 5
	1,1,0: 6
	1,1,1: 7
12	joystick instalado
13	impressora serial instalada (somente PCjr)
14,15	número de impressoras:
	0,0: 0
	0,1: 1
	1,0: 2
	1,1: 3

```
-----
```

Isto pode ser representado como um campo-bit, usando-se esta estrutura:

```
struct perif {
unsigned boot_floppy: 1;
unsigned tem_coproc: 1;
```

```

unsigned ram_placa:    2;
unsigned modo_video:  2;
unsigned drive_floppy: 2;
unsigned DMA:         1;
unsigned portas:      3;
unsigned joystick:    1;
unsigned sem_uso:     1;
unsigned num_impressoras: 2;
    } eq;

```

O programa seguinte usa este campo-bit para exibir o número de drives de discos flexíveis e o número de portas seriais:

```

#include<stdio.h>
#include<bios.h>
main()
{
struct perif {
unsigned boot_floppy:  1;
unsigned tem_coproc:   1;
unsigned ram_placa:    2;
unsigned modo_video:  2;
unsigned drive_floppy: 2;
unsigned DMA:         1;
unsigned portas:      3;
unsigned joystick:    1;
unsigned sem_uso:     1;
unsigned num_impressoras: 2;
    } eq;

int *i;
i=(int *)&eq;

*i= biosequip();

printf("%d drives \n", eq.drive_floppy+1); /* (0,0) = 1 drive */
printf("%d portas \n", eq.portas);

return 0;
}

```

O endereço eq é atribuído ao ponteiro para inteiro i e o valor de retorno da função biosequip() é atribuído a eq por meio deste ponteiro. Isto é necessário, uma vez que a linguagem C não permitirá que um inteiro seja um ponteiro para uma estrutura.

Como se pode ver por este exemplo, cada campo-bit é associado usando-se o operador ponto. Entretanto, se a estrutura é referenciada por meio de um ponteiro, deve-se usar o operador seta.

Não é necessário nomear cada campo-bit. Isto torna fácil encontrar o bit que se quer, rejeitando os não usados. Por exemplo, o campo sem_uso pode ser deixado sem nome, como mostrado:

```

struct perif {
unsigned boot_floppy:  1;
unsigned tem_coproc:   1;
unsigned ram_placa:    2;
unsigned modo_video:  2;

```

```

unsigned drive_floppy: 2;
unsigned DMA: 1;
unsigned portas: 3;
unsigned joystick: 1;
unsigned: 1;
unsigned num_imprensoras: 2;
    } eq;

```

Variáveis do tipo campo-bit têm certas restrições. Não se pode obter o endereço de uma variável campo-bit. As variáveis campo-bit não podem ser colocadas em matrizes. Não se pode ultrapassar o limite dos inteiros. Se a máquina usa a regra memória_baixa-byte_baixo (DEC e National) ou memória_baixa-byte_alto (Intel e Motorola) afeta o modo de endereçamento dos campos. Isto implica que qualquer código que usa os campos-bit pode ter alguma dependência de máquina.

Pode-se misturar membros normais de estruturas com elementos campos-bit. Por exemplo:

```

struct emp {

struct addr endereco;
float salario;
unsigned situacao: 1; /* demitido ou em atividade */
unsigned contrato: 1; /* horista ou mensalista */
unsigned deducos: 3; /* deducos: IR,IAPAS,...(até 2^3=8) */

};

```

Esta estrutura define um registro de empregados que usa somente um byte para armazenar três pedaços de informação: situação, contrato e deduções. Sem o uso do campo-bit estas informações poderiam requerer 3 bytes.

12) UNIONS:

Em C uma union é uma localização de memória usada por muitos tipos de variáveis diferentes. A declaração de uma union é similar a de uma estrutura, como mostrado:

```

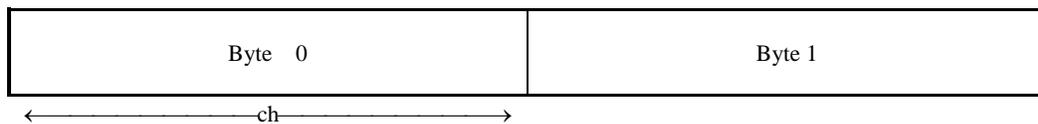
union u_tipo {
int i;
char ch;
};

```

Assim como com estruturas, esta declaração não declara qualquer variável. Pode-se declarar uma variável tanto colocando-se o nome dela no fim da declaração como pelo uso de uma declaração em separado. Para declarar uma variável union chamada `cnvt`, do tipo `u_tipo` usando a declaração dada anteriormente escreve-se:

```
union u_tipo cnvt;
```

Em `cnvt`, o inteiro `i` e o caractere `ch` compartilham a mesma localização de memória. Obviamente `i` ocupa 2 bytes e `ch` ocupa 1 byte. A figura a seguir mostra como `i` e `ch` compartilham o mesmo endereço:




```

        };

union bits {
char ch;
struct byte bit;
        }ascii;

void decode(union bits b);

main()
{

do{
ascii.ch=getche();
printf(": ");
decode(ascii);
}while(ascii.ch!='q'); /* encerra se digitado 'q' */

return 0;
}

/* exibe a sequencia de bits para cada caractere: */
void decode(union bits b)
{
if(b.bit.h)printf("1 ");
else printf("0 ");
if(b.bit.g)printf("1 ");
else printf("0 ");
if(b.bit.f)printf("1 ");
else printf("0 ");
if(b.bit.e)printf("1 ");
else printf("0 ");
if(b.bit.d)printf("1 ");
else printf("0 ");
if(b.bit.c)printf("1 ");
else printf("0 ");
if(b.bit.b)printf("1 ");
else printf("0 ");
if(b.bit.a)printf("1 ");
else printf("0 ");
printf("\n");
}

```

Para encerrar, vamos ver como uma union pode prover um meio de carregar um inteiro em um campo de bit. No programa anterior que determinava o número de drives e portas seriais, tivemos que usar um artifício devido à linguagem C não permitir que um inteiro aponte para uma estrutura. Contudo, este programa cria uma union que contém um inteiro e um campo de bit. Quando a função bioequip() retorna a lista de periféricos codificada como um inteiro, este é atribuído para o inteiro no campo de bit. Entretanto, o programa está livre para usar o campo-bit quando for reportar os resultados.

```
/* Mostra o número de acionadores de disco flexível e portas usando union */
```

```

#include<stdio.h>
#include<bios.h>
main()

```

```

{
struct perif {
unsigned boot_floppy: 1;
unsigned tem_coproc: 1;
unsigned ram_placa: 2;
unsigned modo_video: 2;
unsigned drive_floppy: 2;
unsigned DMA: 1;
unsigned portas: 3;
unsigned joystick: 1;
unsigned sem_uso: 1;
unsigned num_impessoras: 2;
};

union {
struct perif eq;
unsigned i;
} eq_union;

eq_union.i= biosequip();

printf("%d drives \n", eq_union.eq.drive_floppy+1); /* (0,0) = 1 drive */
printf("%d portas \n", eq_union.eq.portas);
return 0;
}

```

13) OPERADORES LÓGICOS DE BIT:

Diferente de muitas outras, a linguagem C suporta um arsenal completo de operadores de bit. Já que a linguagem C foi projetada para tomar o lugar da linguagem Assembly em muitas tarefas de programação, era importante que ela tivesse a habilidade de suportar todas (ou, no mínimo a maior parte) as operações que podem ser feitas em Assembler. Operações bit-a-bit referem-se a testar, indicar ou movimentar os bits em um byte que corresponde aos tipos char, int ou long em C. Operadores de bit não podem ser usados com os tipos float, double, long double, void ou outro tipo mais complexo.

Operadores de bit:

Operador	Ação
a&b	cada bit de a AND cada bit de b
a b	cada bit de a NOR cada bit de b
a^b	cada bit de a XOR cada bit de b
~a	NOT a (complemento de 1)
v>>b	v é deslocado b bits p/ a direita
v<<b	v é deslocado b bits p/ a esquerda

Exemplos:

```

/* converte para binário */
#include<stdio.h>

void exhibe_binario();

```

```

main()

{
int i;
Linit:
printf("Digite um inteiro: ");
scanf("%d",&i);
if(i>256){puts("O inteiro deve ser menor que 256!");goto Linit;}
puts("\nRepresentação binária do inteiro: ");
exibe_binario(i);
return 0;
}
/*Exibe os bits dentro de um byte: */
void exibe_binario(int i)
{
register int t;
for(t=128;t>0;t=t/2)
if(i&t) printf("1 ");
else printf("0 ");
printf("\n");
}

/* Desloca um bit dentro de um byte e mostra */
#include <stdio.h>

void exibe_binario();

main()
{
int i=1, t;

for(t=0;t<8;t++){
exibe_binario(i);
i=i<<1;    }
printf("\n");

for(t=0;t<8;t++){
i=i>>1;
exibe_binario(i);
}

return 0;
}

/*Exibe os bits dentro de um byte*/

void exibe_binario(int i)
{
register int t;
for(t=128;t>0;t=t/2)
if(i&t) printf("1 ");
else printf("0 ");
printf("\n");
}

```

```

/* codifica e decodifica uma string */
#include<stdio.h>

main()
{
char str[80];
int i;

puts("Digite uma string: ");
gets(str);

for(i=0;i<strlen(str);i++) str[i]=~str[i]; /* NOT em cada um dos 8 bits de */
/* cada char que compoe str */

puts("A string codificada/decodificada é: ");
puts(str);
return 0;
}

```

14) FUNÇÕES DE CONTROLE DA TELA:

O controle da tela do Turbo C é dividido em duas partes: modo texto e modo gráfico. As funções de modo texto gerenciam a aparência da tela quando a placa de vídeo está selecionada para um modo de texto e as funções gráficas usam a tela quando a placa de vídeo está selecionada para um modo de exibição gráfica. Ambos os modos são selecionáveis via software, por comandos específicos que veremos adiante.

14.1) AS FUNÇÕES BÁSICAS DE MODO TEXTO:

Antes de explorar algumas das mais importantes funções de controle e manipulação de tela no modo texto, há alguns pontos a considerar. Primeiro, todas as funções de modo texto no Turbo C requerem a presença do arquivo-cabeçalho conio.h em qualquer programa que as use. Segundo, as funções de modo texto consideram que a tela está em um modo texto, não gráfico. Já que esse modo é o padrão quando da inicialização (boot) do PC, isto não deve ser problema. Brevemente aprenderemos a fixar os diversos modos de vídeo.

Janelas de Texto:

Muitas das rotinas do modo texto do Turbo C operam em uma janela, não na tela. Felizmente, a janela padrão é a tela inteira, de modo que não é necessário se preocupar com a criação de qualquer janela especial para usar as rotinas de modo texto (e gráfico). Entretanto, é importante entender os conceitos básicos das janelas, a fim de se obter o máximo das funções de tela do Turbo C. Uma janela é uma área retangular que o seu programa usa para enviar mensagens para o usuário. Ela pode ser tão grande quanto a tela inteira ou tão pequena quanto uns poucos caracteres. Em programas sofisticados, não é incomum a tela ter muitas janelas ao mesmo tempo - uma para cada tarefa realizada pelo programa.

O Turbo C permite definir a localização e as dimensões de uma janela. Depois de se definir uma janela, as rotinas que manipulam texto afetam somente a janela definida e não a tela inteira. Por exemplo, a função clrscr() apaga a janela ativa, não a tela inteira (a menos, obviamente, que, como é por definição, a janela ativa seja a tela inteira). Adicionalmente, todas as coordenadas de posição são relativas à janela ativa em vez de à tela.

Um dos aspectos mais importantes desse assunto é que o Turbo C automaticamente previne as saídas de irem além dos limites de uma janela. Se alguma saída ultrapassar um limite, somente a parte que couber será exibida, o resto é iniciado na próxima linha.

Para janelas de texto, as coordenadas do canto superior esquerdo são (1,1). As funções de tela modo texto que usam coordenadas aplicam-nas relativas à janela ativa não à tela. Assim, se existem duas janelas, o canto superior esquerdo de ambas é considerado a localização (1,1) quando alguma está ativa. Para o momento, não nos preocuparemos com a criação de qualquer janela, simplesmente usaremos a janela padrão, a tela inteira.

Limpando a janela:

Uma das funções mais comuns de manipulação de tela é a que limpa a janela ativa (e somente a janela ativa):

```
void clrscr(void);
```

Posicionando o cursor:

Para posicionar o cursor na janela ativa usa-se:

```
void gotoxy(int x, int y);
```

Aqui, x e y especificam as coordenadas em relação à janela ativa para onde o cursor é posicionado. Se ambas as coordenadas estão fora do limite, nenhuma ação é realizada, contudo isso não é considerado um erro.

Exemplo:

```
/*Demonstração das funções clrscr() e gotoxy()*/
#include <conio.h>
#include <stdio.h>
#include <string.h>

char mensagem[]="Funções de Tela Modo Texto São Um Tédio!";

main()
{
register int x,y;
char *p;

clrscr();
p=mensagem;
for(x=1;x<=strlen(mensagem);x++){
for (y=1; y<=12; y++){
gotoxy (x,y);
delay(50); /* espera 0.05 segundos */
printf("%c",*p); /* imprime de cima para baixo */
gotoxy (x, 25-y);
printf("%c",*p); /* imprime de baixo para cima */
}
p++;
}

getche();
gotoxy(1,25);
```

```
    return 0;
}
```

Apagando até o fim da linha:

A função `clreol()` limpa da esquerda para a direita, uma linha a partir da posição corrente do cursor até o final da janela. É útil quando se requisita informação do usuário:

```
void clreol(void);
```

Pode-se combinar esta função com `gotoxy` para exibir uma mensagem de aviso nas coordenadas especificadas, após ter limpaado a linha. É útil para manter a tela livre de mensagens perdidas:

```
#include <conio.h>
#include <stdio.h>
void aviso();
main()
{
    clrscr();
    aviso("isto é um aviso",1,10);
    getch();
    aviso("isto também", 1, 10);
    getch();
    return 0;
}
```

```
void aviso(char *s, int x, int y)
{
    gotoxy(x,y);
    clreol();
    printf(s);
}
```

Apagando e Inserindo Linhas:

Quando se tem a tela cheia de texto e se quer remover (nao simplesmente apagar) uma ou mais linhas, reposicionando as linhas de texto abaixo da removida (scroll) ou quando se quer criar uma linha em branco onde se quer inserir um texto usa-se `delline()` e `insline()`:

```
void delline(void);
void insline(void);
```

Exemplo:

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

main()
{
    int x, y;
    clrscr();

    /* Preenche a tela inteira com algumas linhas: */
    for(y=1; y<25; y++)
```

```

for(x=1; x<80; x++){
gotoxy(x,y);
printf("%c",(y-1)+'A');
}

```

```

getch();

```

```

/*apaga todas as linhas de dois em dois a partir dos B's: */
/* e rola a tela para cima */

```

```

for(y=2;y<26;y+=2){
gotoxy(1,y);
delline();
}

```

```

return 0;
}

```

Variação do programa anterior:

```

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

```

```

main()
{
int x, y;
clrscr();

```

```

/* Preenche a tela inteira com algumas linhas: */
for(y=1; y<25; y++)
for(x=1; x<80; x++){
gotoxy(x,y);
printf("%c",(y-1)+'A');
}

```

```

getch();

```

```

/*apaga todas as linhas alternadas e insere uma linha em branco no lugar*/
for(y=2;y<26;y+=2){
gotoxy(1,y);
delline();
insline();
}

```

```

return 0;

```

```

}

```

Criando Janelas:

Para se criar janelas de qualquer tamanho e em qualquer lugar, desde que caibam na tela, usa-se window():

```
void window(int esquerda, int acima, int direita, int abaixo);
```

Se alguma coordenada é inválida, window não realiza qualquer ação.

```
/* demonstra window() */
#include <conio.h>
#include <stdio.h>
main()
{
system("cls"); /*limpa a tela */

gotoxy(2,3);
printf("coord (2,3) da tela");

window(10,10,60,15);
gotoxy(2,3);
printf("coord (2,3) da janela");
}
```

Observe que as coordenadas usadas para chamar window são absolutas da tela e não relativas à janela correntemente ativa. Isto significa que múltiplas janelas podem ser abertas sem terem coordenadas uma relativa à outra, isto é, não são consideradas aninhadas. No entanto, uma janela pode superpor-se à outra janela.

```
/*Um programa de demonstração de janela de texto*/
#include <conio.h>
#include <stdio.h>

void borda();
main()
{
clrscr();

/*Desenha uma borda em volta da tela*/
borda(1,1,79,25);

/*Cria a primeira janela*/
window(3,2,40,9);
borda(3,2,40,9);
gotoxy(3,2);
printf("Primeira janela");

delay(700); /* espera 0.7 segundos */

/*Cria uma segunda janela*/
window(30,10,60,18);
borda(30,10,60,18);
gotoxy(3,2);
printf("Segunda janela");
delay(700); /* espera 0.7 segundos */
gotoxy(5,4);
printf("Ainda na segunda janela");

delay(700); /* espera 0.7 segundos */
```

```

/*Retorna para a primeira janela*/
window(3,2,40,9);
gotoxy(3,3);
printf("De volta à primeira janela");

delay(700); /* espera 0.7 segundos */

/*Demonstra sobreposição de janelas*/
window(5,5,50,15);
borda(5,5,50,15);
gotoxy(2,2);
printf("Esta janela sobrepoe às duas primeiras");
getch();
return 0;
}

/*Desenha uma borda em torno de uma janela de texto*/

void borda(int iniciox, int inicioy, int fimx, int fimy)

{
register int i;

gotoxy(1,1);
for(i=0; i<=fimx-iniciox; i++) putch('*'); /* putch() escreve um char p/ a */
/* janela ativa */

gotoxy(1, fimy-inicioy);
for(i=0; i<=fimx-iniciox; i++) putch('*');

for(i=2; i<fimy-inicioy; i++) {
gotoxy(1,i);
putch('*');
gotoxy(fimx-iniciox+1,i);
putch('*');
}
}
}

```

Algumas funções de E/S em janelas:

Quando se usa a janela padrao, que é a tela inteira, nao importa se sao usadas as funcoes de E/S padrao (como printf, etc..) ou as funcoes de E/S de janela. No entanto, se estivermos usando uma janela pequena em algum lugar da tela, seremos obrigados a usar as funcoes de janela.

Função:	Uso:
cprintf()	como printf() só que escreve para a janela ativa
cputs()	como puts() só que escreve para a janela ativa
putch()	escreve um char para a janela ativa
getche()	lê um caractere de dentro da janela ativa
cgets()	lê uma string de dentro da janela ativa

Quando se usa uma função de janela o texto de entrada ou saída se quebra automaticamente no limite da janela ativa:

```
/* demonstração de algumas das funções de janela de texto */
#include<conio.h>
#include<stdio.h>

void borda();

main()
{
    clrscr();

    /*cria uma janela*/
    window(3,2,20,9);
    borda(3,2,20,9);
    gotoxy(2,2);
    cprintf("Esta linha quebrará na borda da janela ");
    printf("Esta linha nao quebrará na borda da janela");
    getch();
    return 0;
}

/*Desenha uma borda em torno de uma janela de texto*/
void borda(int iniciox, int inicioy, int fimx, int fimy)
{
    register int i;
    gotoxy(1,1);
    for(i=0; i<=fimx-iniciox; i++) putchar('*'); /* putchar() escreve um char p/ a */
                                                /* janela ativa */
    gotoxy(1, fimy-inicioy);
    for(i=0; i<=fimx-iniciox; i++) putchar('*');

    for(i=2; i<fimy-inicioy; i++) {
        gotoxy(1,i);
        putchar('*');
        gotoxy(fimx-iniciox+1,i);
        putchar('*');
    }
}
```

Outra característica importante das funções de janela de texto é que elas não são redirecionáveis pelo DOS nem a nível de entrada nem a nível de saída. Experimente redirecionar a saída do programa anterior para um arquivo e observe a tela e arquivo(use type).

Modos Texto:

O padrão PC-IBM para modo de vídeo é uma tela com 80 colunas por 25 linhas. Dependendo da placa de vídeo que um computador dispõe é possível selecionar vários modos de vídeo para texto pela função `textmode()`:

```
void textmode(int modo);
```

modo (macro):	inteiro equivalente:	descrição:
---------------	----------------------	------------

BW40	0	40 col. PB
C40	1	40 col. colorido
BW80	2	80 col. 16 tons cinza
C80	3	80 col. colorido
MONO	7	80 col. monocromático
LASTMODE	-1	modo anterior (original)

Exibindo textos em cores: (obviamente é necessário um vídeo colorido)

A função `textcolor(cor)` muda a cor dos textos subsequentes para a cor especificada pelo argumento `cor`. Só funciona se for usada uma das funções de janela (`printf()` não funciona):

```
void textcolor(int cor);
```

Obs: $0 \leq \text{cor} \leq 15$

Do mesmo modo, podemos definir uma cor de fundo para os textos:

```
void textbackground(int cor);
```

Obs: $0 \leq \text{cor} \leq 7$

É importante observar que estas funções só alteram saídas de texto após sua chamada no programa.

cor(macro):	inteiro equivalente:
-------------	----------------------

BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGREY	7
DARKGREY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

/ este programa define todas as combinações de cores para textos */*

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

```

main()
{

register int ft,fd;
textmode(C80);

for(ft=BLUE;ft<=WHITE;ft++){
for(fd=BLACK;fd<=LIGHTGRAY;fd++){

textcolor(ft);
textbackground(fd);
delay(70);
cprintf("Isto é um teste ");
}
cprintf("\n\r");
}
textcolor(WHITE);
textbackground(BLACK);
cprintf("pronto");
textmode(LASTMODE);
return 0;
}

```

Para fazer com que um texto pisque, deve-se fazer o OR da cor com 128 (BLINK):

```
textcolor(GREEN | BLINK);
```

14.2) UMA INTRODUÇÃO AO SISTEMA GRÁFICO DO TURBO C:

Acima de Tudo uma Janela:

Como as funções de controle de tela do modo texto, todas as funções gráficas operam por meio de uma janela. Na terminologia do Turbo C, uma janela gráfica é chamada de porta de visualização, mas uma porta de visualização tem essencialmente as mesmas características das janelas de texto. A única diferença real entre uma janela e uma porta de visualização é que o canto superior esquerdo de uma porta de visualização é a coordenada(0,0) e não (1,1) como em uma janela.

Por definição, uma tela inteira é uma porta de visualização, contudo, pode-se criar portas de visualização de outras dimensões, como veremos mais tarde. É importante ter em mente que todas as saídas gráficas são relativas à porta de visualização corrente, que não é necessariamente a mesma que a tela.

Inicializando a Placa de Vídeo:

Antes que qualquer das funções gráficas possam ser usadas, é necessário colocar a placa de vídeo em um dos modos gráficos. Obviamente o computador deve ter uma placa gráfica de vídeo, caso contrário, nenhuma das funções e comandos vistos daqui por diante serão válidos. Por definição, a grande maioria dos sistemas PC-IBM inicializam (boot) o computador no modo texto 80 colunas. Já que este não é o modo gráfico, as funções gráficas não podem funcionar nele. Para inicializar a placa de vídeo em um modo gráfico, usa-se a função `initgraph()`:

```
void far initgraph(int far *driver,int far *modo, char far *path);
```

A função `initgraph()` carrega um driver gráfico correspondente ao número na memória apontado por `driver`. Sem um driver gráfico carregado na memória, nenhuma função gráfica pode operar. O argumento `modo` aponta para o inteiro que especifica um dos modos de vídeo disponível no driver gráfico. O argumento `path`

especifica em que local da estrutura de diretórios encontra-se o arquivo do driver gráfico utilizado. Se nenhum path é especificado, o diretório de trabalho corrente é pesquisado.

Os drivers gráficos estão contidos nos arquivos *.BGI, que devem estar disponíveis no sistema.

Drive Gráfico	Inteiro Equivalente
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Quando se usa DETECT, initgraph detecta automaticamente o tipo de placa e monitor de vídeo presente no sistema e seleciona o modo de vídeo com a melhor resolução. Isso faz com que os valores dos argumentos driver e modo sejam fixados automaticamente.

O valor de modo deve ser um dos modos gráficos mostrados na tabela abaixo.

Placa Vídeo (driver *.BGI)	Modo	Inteiro Equivalente	Resolução
CGA	CGAC0	0	320x200
	CGAC1	1	320x200
	CGAC2	2	320x200
	CGAC3	3	320x200
	CGAHI	4	640x200
MCGA	MCGAC0	0	320x200
	MCGAC1	1	320x200
	MCGAC2	2	320x200
	MCGAC3	3	320x200
	MCGAMED	4	640x200
	MCGAHI	5	640x480
EGA	EGALO	0	640x200
	EGAHI	1	640x350
EGA64	EGA64LO	0	640x200
	EGA64HI	1	640x350
EGAMONO	EGAMONOH	3	640x350
HERC	HERCMONOH	0	720x348
ATT400	ATT400C0	0	320x200
	ATT400C1	1	320x200
	ATT400C2	2	320x200
	ATT400C3	3	320x200
	ATT40CMED	4	640x200
	ATT400CHI	5	640x400
VGA	VGALO	0	640x200
	VGAMED	1	640x250

	VGAHI	2	640x480
PC3270	PC3270HI	0	720x350
IBM8514	IBM8514LO	0	640x480
	IBM8514HI	1	1024x768

Por exemplo, para fazer com que o sistema seja inicializado no modo gráfico CGA 4 cores 320x200 pontos de resolução, use:

```
7
#include<graphics.h>
int driver, modo;

driver=CGA;
modo=CGAC0;

initgraph(&driver,&modo,""); /*assume o arquivo CGA.BGI no diretório corrente*/
```

Deixando o modo gráfico temporariamente e definitivamente:

```
#include<graphics.h>
int driver, modo;
driver=CGA;
modo=CGAC0;
initgraph(&driver,&modo,""); /*assume o arquivo CGA.BGI no diretório corrente*/
{
faz algo no modo gráfico
}

restorecrtmode() /* volta p/ texto temporariamente */
{
faz algo no modo texto
}

setgraphmode(modo) /* volta ao modo gráfico dado por modo */

{
faz algo no modo gráfico novamente
}

closegraph() /* sai definitivamente do modo gráfico: esvazia a memória do driver *.BGI */
```

Cores e Paletas: (EGA/VGA):

O tipo de placa de vídeo conectada ao seu sistema determina os tipos e cores disponíveis quando se está em modo gráfico. A tendência atual é a linha de monitores VGA (onde se incluem os EGA).

A CGA 4 cores dá acesso a 4 cores por paleta e 4 paletas para escolher. Comparado com os demais monitores, o CGA é de uma limitação ímpar no que diz respeito a apresentação de modo gráfico colorido. Não cobriremos cores no CGA.

Nos monitores EGA, VGA e SVGA quando se quer estabelecer uma cor para as diversas funções de desenho do Turbo C sobre uma cor de fundo faz-se:

setcolor(cor); /* dá a cor do traçado de uma função de desenho
subsequentemente chamada */

setbkcolor(cor_de_fundo); /* dá a cor de fundo do traçado de uma função de desenho subsequentemente
chamada */

onde cor e cor_de_fundo são dados pelas tabelas abaixo:

cor	inteiro equivalente:
EGA_BLACK	0
EGA_BLUE	1
EGA_GREEN	2
EGA_CYAN	3
EGA_RED	4
EGA_MAGENTA	5
EGA_BROWN	20
EGA_LIGHTGREY	7
EGA_DARKGREY	56
EGA_LIGHTBLUE	57
EGA_LIGHTGREEN	58
EGA_LIGHTCYAN	59

cor	inteiro equivalente:
EGA_LIGHTRED	60
EGA_LIGHTMAGENTA	61
EGA_YELLOW	62
EGA_WHITE	63

cor_de_fundo	inteiro equivalente:
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGREY	7
DARKGREY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

As Funções Básicas de Desenho:

As funções gráficas mais fundamentais são aquelas que desenharam um ponto, uma linha e um círculo. Essas funções são chamadas de `putpixel()`, `line()` e `circle()`, respectivamente. Os seus protótipos são mostrados aqui:

```
void far putpixel(int x, int y, int cor);
void far line(int começox, int começoy, int fimx, int fimy);
void far circle(int x, int y, int raio);
```

A função `putpixel()` escreve a cor especificada na localização determinada por `x` e `y`. A função `line()` desenha uma linha a partir da localização especificada por `(começox, começoy)` até `(fimx, fimy)`, na cor corrente estabelecida por `setcolor()`. A cor de desenho padrão é branco. A função `circle()` desenha um círculo de raio igual a `raio`, na cor estabelecida por `setcolor()`, com centro determinado por `(x,y)`.

Se qualquer das coordenadas estiver fora dos limites, será exibida somente a parte (se houver) do círculo dentro do limite.

```
/*Demonstração de pontos, linhas ,círculos e saída de texto*/

#include<graphics.h>
#include<conio.h>
main()
{
int driver,modo;
register int i;
driver=DETECT;

initgraph(&driver,&modo,"");

setcolor(EGA_LIGHTGREEN);
line(0,0,200,150);
setcolor(EGA_LIGHTRED);
line(50,100,200,125);

/*Desenha alguns pontos*/
for(i=0;i<319;i+=10)putpixel(i,100,EGA_YELLOW);

/*Desenha alguns círculos*/

setcolor(EGA_LIGHTBLUE);
circle(50,50,35);
setcolor(EGA_LIGHTMAGENTA);
circle(100,160,100);

sleep(1); /* espera 1 segundo */
cleardevice(); /* limpa a tela gráfica */

/* escreve uma string de texto iniciando no meio da tela grafica */

setbkcolor(LIGHTBLUE); /* cor de fundo azul claro */
setcolor(EGA_LIGHTRED);

outtextxy(getmaxx()/2,getmaxy()/2,"Isto é um teste"); /* imprime a string
iniciando no meio da tela */
```

```

getch();
closegraph();
return 0;
}

```

Preenchendo uma área:

Para preencher qualquer forma geométrica fechada, usa-se a função floodfill():

```
void far floodfill(intx, inty, int cor_da_borda);
```

Para usar essa função chame-a com as coordenadas(x,y) de um ponto dentro da figura e a cor da linha que forma a borda da figura fechada. É importante certificar-se de que o objeto a ser preenchido está completamente fechado, caso contrário, a área fora da figura será preenchida também.

O que preencherá o objeto é determinado pelo padrão e cor corrente de preenchimento. Por definição, a cor de fundo definida por setbkcolor() é usada. Entretanto, é possível mudar o modo como os objetos são preenchidos usando-se a função setfillstyle():

```
void far setfillstyle(int padrao, int cor);
```

Onde padrao é definido pelo tópico 'constants, data types and global variables' item 'fill_pattern' em graphics.h.

A Função Rectangle:

A função rectangle desenha uma caixa definida pelas coordenadas (esquerda,acima) e (direita,baixo) na cor de desenho corrente:

```
void far rectangle(int esquerda, int acima, int direita, int abaixo)
```

```
/*programa de demonstração de cor, retângulos e padrões de
preenchimento*/
```

```
#include<graphics.h>
```

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

```
main()
```

```
{
```

```
int driver, modo;
```

```
register int i;
```

```
driver=VGA;
```

```
modo=VGAMED;
```

```
initgraph(&driver,&modo,"");
```

```
/*Desenha uma borda na tela*/
```

```
setcolor(EGA_RED);
```

```
rectangle(0,0,639,349);
```

```
sleep(1); /* espera 1 segundo */
```

```
/* desenha diagonal */
```

```
setcolor(EGA_YELLOW);
```

```
line(0,0,639,349);
```

```

sleep(1);

/* desenha um retangulo */
setcolor(EGA_LIGHTGREEN);
rectangle(100,100,300,200);

sleep(1);

/* preenche o retangulo anterior */
setfillstyle(SOLID_FILL,BLUE);
floodfill(110,110,EGA_LIGHTGREEN);

sleep(1);

/* desenha uma linha */
setcolor (EGA_BLUE);
line(50,200,400,125);

sleep(1);

/*Desenha o setor superior em 'V'*/
setcolor (EGA_LIGHTRED);
for(i=0;i<640;i+=3) line(320,174,i,0);

sleep(1);

/* desenha alguns circulos */
setcolor(EGA_LIGHTGREEN);
circle(50,50,35);
setfillstyle(SOLID_FILL,MAGENTA); /* preenche */
floodfill(50,50,EGA_LIGHTGREEN);

setcolor(EGA_LIGHTMAGENTA); /* nao preenche este */
circle(320,175,100);

setcolor(EGA_LIGHTCYAN); /* nao preenche este */
circle(100,100,200);

sleep(1);

/*Desenha um alvo*/
setcolor(EGA_GREEN);
circle(500,250,90);
setfillstyle(SOLID_FILL,GREEN);
floodfill(500,250,EGA_GREEN); /* preenche círculo mais externo */

setcolor (EGA_RED);
circle (500,250,60);
setfillstyle(SOLID_FILL,RED);
floodfill(500,250,EGA_RED);

setcolor (EGA_GREEN);
circle (500,250,30);
setfillstyle(SOLID_FILL,GREEN);
floodfill(500,250,EGA_GREEN);

```

```

setcolor (EGA_RED);
circle (500,250,10);
setfillstyle(SOLID_FILL,RED);
floodfill(500,250,EGA_RED);

getch();
closegraph();
}

```

Criando Portas de Visualização:

É possível criar portas de visualização em modo gráfico de maneira similar à forma como se cria janelas no modo texto. Como afirmado anteriormente, todas as saídas gráficas são relativas às coordenadas da porta de visualização ativa. Isto significa que as coordenadas do canto superior esquerdo da porta são (0,0), não interessando onde a porta de visualização esteja. A função usada para criar uma porta de visualização gráfica chama-se `setviewport()`:

```
void far setviewport(int esquerda, int acima, int direita, int abaixo, int clipflag);
```

Para usá-la especifica-se as coordenadas do canto superior esquerdo e do canto inferior direito da porta. Se o parâmetro `clipflag` é diferente de zero, será realizado um truncamento automático da saída que pode exceder os limites da porta de visualização. Isto evita consumo inútil da memória de vídeo. Com clipping desabilitado é possível a saída ir além da porta de visualização.

Pode-se encontrar as dimensões da porta de visualização corrente usando-se a função `getviewsettings()`:

```
void far getviewsettings(struct viewporttype far *info);
```

A estrutura `viewporttype` é definida em `graphics.h` como mostrado aqui:

```

struct viewporttype{
int left, top, right, bottom;
int clipflag;
}

```

Os campos `left`, `top`, `right` e `bottom` armazenam as coordenadas do canto superior esquerdo e canto superior direito da porta de visualização. Quando o `clipflag` é zero, nenhum truncamento da saída ocorre caso esta ultrapasse os limites da porta.

O uso mais importante da função `getviewsettings` é permitir que os programas se ajustem automaticamente às dimensões da tela ditadas pela placa de vídeo presente no sistema.

Consultando a estrutura `viewporttype` o programa pode saber as dimensões da porta de visualização corrente e fazer o ajuste necessário em suas variáveis internas para que a área gráfica caia na tela.

```

/*Demonstração de portas de visualização*/
#include<graphics.h>
#include<stdlib.h>
#include<conio.h>
main()
{
int driver, modo;
struct viewporttype porta;
int largura, altura;
register int i;

```

```

driver=DETECT;
initgraph(&driver, &modo,"");

/*Obtém as dimensoes da porta.*/
getviewsettings(&porta);

/*Desenha linhas verticais na primeira porta.*/
for (i=0;i<porta.right;i+=20) line(i,porta.top,i,porta.bottom);

/*Cria uma nova porta no centro da porta.*/
altura=porta.bottom/4; largura=porta.right/4;
setviewport(largura,altura,largura*3,altura*3,1);
getviewsettings(&porta);

/*Desenha linhas horizontais dentro da segunda porta.*/
for(i=0;i<porta.right;i+=10)line(0,i,porta.bottom,i);
getch();
closegraph();
return 0;
}

```

15) APÊNDICE I: TABELA DE CARACTERES ASCII E GERADOR

33	!	34	"	35	#	36	\$	37	%	38	&	39	'	40	<	41	>	42	*
43	+	44	,	45	-	46	.	47	/	48	0	49	1	50	2	51	3	52	4
53	5	54	6	55	7	56	8	57	9	58	:	59	;	60	<	61	=	62	>
63	?	64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G	72	H
73	I	74	J	75	K	76	L	77	M	78	N	79	O	80	P	81	Q	82	R
83	S	84	T	85	U	86	V	87	W	88	X	89	Y	90	Z	91	[92	\
93]	94	^	95	_	96	`	97	a	98	b	99	c	100	d	101	e	102	f
103	g	104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o	112	p
113	q	114	r	115	s	116	t	117	u	118	v	119	w	120	x	121	y	122	z
123	{	124		125	}	126	~	127	Δ	128	Ç	129	ü	130	é	131	â	132	ä
133	à	134	â	135	ç	136	ê	137	ë	138	è	139	ï	140	î	141	ì	142	ä
143	â	144	é	145	æ	146	œ	147	ô	148	ö	149	ò	150	û	151	ù	152	ÿ
153	ö	154	Ü	155	ç	156	£	157	¥	158	ℳ	159	f	160	á	161	í	162	ó
163	ú	164	ñ	165	Ñ	166	œ	167	ø	168	¿	169	ı	170	ı	171	½	172	¼
173	ı	174	«	175	»	176	»	177	»	178	»	179	ı	180	ı	181	ı	182	ı
183	ı	184	ı	185	ı	186	ı	187	ı	188	ı	189	ı	190	ı	191	ı	192	ı
193	ı	194	ı	195	ı	196	ı	197	ı	198	ı	199	ı	200	ı	201	ı	202	ı
203	ı	204	ı	205	ı	206	ı	207	ı	208	ı	209	ı	210	ı	211	ı	212	ı
213	ı	214	ı	215	ı	216	ı	217	ı	218	ı	219	ı	220	ı	221	ı	222	ı
223	ı	224	ı	225	ı	226	ı	227	ı	228	ı	229	ı	230	ı	231	ı	232	ı
233	ı	234	ı	235	ı	236	ı	237	ı	238	ı	239	ı	240	ı	241	ı	242	ı
243	ı	244	ı	245	ı	246	ı	247	ı	248	ı	249	ı	250	ı	251	ı	252	ı
253	ı	254	ı	255	ı														

/* Gerador da tabela ASCII */

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

```
main()
{
```



```

textcolor(YELLOW); /* prox. saída de texto em YELLOW */
gotoxy(5,2);cputs("Iniciando em (5,2) desta janela. ");

cwindow(10,20,40,25,edge1,LIGHTMAGENTA,MAGENTA);/*janela: fundo MAGENTA, borda
LIGHTMAGENTA tipo edge1 */
textcolor(LIGHTRED);/* prox saída texto em LIGHTRED */
gotoxy(6,1);cputs("CONTAGEM REGRESSIVA:");
for(i=10;i>=0;i--){
gotoxy(15,3);cprintf("%02d",i); /* saída para a janela ativa! */
sleep(1);
}

cwindow(5,15,35,20,edge3,LIGHTRED,RED);/*janela: fundo RED, borda LIGHTRED tipo edge3 */
textcolor(WHITE); /* prox saída texto em WHITE */
gotoxy(13,3);cputs("BUM!");
sleep(1);

cwindow(50,5,75,10,edge1,LIGHTBLUE|BLINK,BLUE);/* fundo BLUE, borda LIGHTBLUE piscando tipo
edge1 */
textcolor(YELLOW|BLINK);/* prox saída texto em YELLOW piscando */
gotoxy(4,2);
cprintf("Digite sua idade:");
scanf("%d",&i); /* pega dado da janela ativa */
gotoxy(4,3);
textcolor(WHITE);/* prox saída texto em WHITE */
cprintf("Voce tem %d anos!",i);
sleep(2);

system("cls"); /* limpa e apaga a tela */
}

void cwindow(lft,up,rgt,dwn,edge,edgecolor,backgrcolor)
int lft,up,rgt,dwn;
int edgecolor,backgrcolor;
char *edge;

/*
<lft,up>
e0 e5 e1
e4
e3 e2
<rgt,dwn>
*/

{
register int i,j;
char line[80]="";
char tmp[2];
union REGS inregs,outregs;
struct text_info ti;

window(lft+1,up+1,rgt-1,dwn-1); /* define area da janela */
textbackground(backgrcolor); /* define cor de fundo */
clrscr(); /* limpa tela e pinta fundo */

```

```

window(1,1,80,25);          /* redefine: janela = janela total */

/* coloca borda */
if(edge!=NULL){

    /* define linhas horizontais da borda */
    tmp[0]=edge[5];
    tmp[1]='\0';
    j= lft+1;
    while(j<=rgt-1){
        strncat(line,tmp,1);
        j++;    }

    /* imprime lados horizontais da borda */

    textcolor(edgecolor);

    gotoxy(lft+1,up);
    cprintf("%s",line);

    gotoxy(lft+1,dwn);
    cprintf("%s",line);

    /* imprime lados verticais da borda */

    i=up+1;
    while(i<=dwn-1) {
        gotoxy(lft,i);
        putchar(edge[4]);
        gotoxy(rgt,i);
        putchar(edge[4]);
        i++;
    }

    /* coloca caracteres nos 4 cantos da janela */

    gotoxy(lft,up);putchar(edge[0]);
    gotoxy(rgt,up);putchar(edge[1]);
    gotoxy(lft,dwn);putchar(edge[3]);

    /* truque p/ evitar rolagem do caso tela = tela total (80x25) */
    /* quando escrevendo o caracter do canto inferior direito da janela */

    gotoxy(rgt,dwn);
    gettextinfo(&ti); /* pega text info e bota na struct ti */

    inregs.h.ah=0x09; /* int 10h, funcao 09h: escreve char na pos. cursor */
                    /* c/ atributo especificado e NAO avança cursor */
                    /* That's the trick! */

    inregs.h.al=edge[2]; /* char a ser escrito */
    inregs.h.bh=0; /* pagina ativa */

```

```

inregs.h.bl=ti.attribute; /* atributo definido na struct ti */
inregs.x.cx=1; /* fator de repetição */
int86(0x10,&inregs,&outregs);

    }

    window(lft+1,up+1,rgt-1,dwn-1); /* reduz em uma unidade as dimensoes */
                                   /* da janela ativa p/ evitar transbordo */
                                   /* alem da borda definida quando a linha */
                                   /* a ser impressa excede tamanho da janela */
}

```

REGS (union) is defined in <DOS.H>

```

union REGS {
    struct WORDREGS x;
    struct BYTEREGS h;
};

```

BYTEREGS and WORDREGS are defined in <DOS.H>

Structures for storing byte and word registers

```

struct BYTEREGS {
    unsigned char  al, ah, bl, bh;
    unsigned char  cl, ch, dl, dh;
};

struct WORDREGS {
    unsigned int  ax, bx, cx, dx;
    unsigned int  si, di, cflag, flags;
};

```

int86 is defined in <DOS.H>

```

int int86(int intno, union REGS *inregs, union REGS *outregs);

```

text_info is defined in <CONIO.H>

Current text window information.
Used by the gettextinfo function.

```

struct text_info {
    unsigned char winleft;    left window coordinate
    unsigned char wintop;    top window coordinate
    unsigned char winright;  right window coordinate
};

```

```

unsigned char winbottom;    bottom window coordinate
unsigned char attribute;    text attribute
unsigned char normattr;    normal attribute
unsigned char currmode;    current video mode:
                           BW40, BW80, C40, C80, or C4350
unsigned char screenheight; text screen's height
unsigned char screenwidth;  text screen's width
unsigned char curx;         x-coordinate in current window
unsigned char cury;         y-coordinate in current window
};

```

17) APÊNDICE III: PROGRAMA DEMONSTRATIVO DE CONTROLE DO MOUSE EM MODO TEXTO

```

/* Programa simples p/ identificação sonora de ícones c/ Mouse IBM */
/* Por Eng. Fernando César C. de Castro - CREARS 48711 */
/* 13/12/1993 */

#include<dos.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>
#include<graphics.h>
#include<alloc.h>
#include<string.h>

#define NOT_MOVED 0
#define RIGHT 1
#define LEFT 2
#define UP 3
#define DOWN 4

#define WLFT 1
#define WUP 1
#define WDWN WUP+HEIG
#define HEIG 2 /* altura da janela internamente sem gap */
#define HINIT 5 /* pos. horiz. 1a. janela desenhada na linha */
#define SPACE 4 /* largura de cada janela sem string e sem o gap */
                /* e tambem espaço entre janelas - 1 */
#define HGAP 2 /* aprox. gap entre caracter e lados horiz da janela (dep. par ou impar) */
#define VGAP 2 /* aprox gap entre lados vert. da janela (dep. par ou impar) */

void storewincoord();
void wedge();
void cwindow();
void cmouses();
void mouse_position();
void mode();
void cursor_on();
void cursor_off();
void mouse_reset();

```

```

typedef struct wincoord {
int lf;
int up;
int rg;
int dw;
        } wico;

main()
{

wico wc[42];
char *notes[]={ "Dó", "Ré", "Mi", "Fá", "Sol", "Lá", "Si" };
int
freq[]={65,73,82,87,98,110,123,131,147,165,175,196,220,247,262,294,330,349,392,440,494,523,587,659,69
8,784,880,988,1047,1175,1319,1397,1568,1760,1976,2093,2349,2637,2794,3136,3520,3951};
int register i;
int x,y;
int wup,wdown,wlft,wrgt,nwin;
int tcolor,bcolor,ecolor;
int noit;

system("cls");
textmode(0x03); /* 16 color text 80x25, virtual screen 640x200, mouse cursor 8x8 pixel */
                /* ver tabela abaixo */

mouse_reset(); /* init mouse */

wup=WUP;
wdown=WDWN;
wlft=WLFT+HINIT;
bcolor=BLUE;
ecolor=LIGHTBLUE;
tcolor=WHITE;
nwin=0;noit=0;

while(nwin<42){ /* 6 octaves with 7 pitch */

if(nwin!=0){ wlft=wrgt+SPACE;} /* pos inicial prox janela na linha horiz de janelas */
wrgt=wlft+strlen(notes[noit])+2*HGAP;

if(wrgt>80){
        wup=wup+HEIG+VGAP;
        wdown=wdown+HEIG+VGAP;
        wlft=WLFT+HINIT;
        wrgt=wlft+strlen(notes[noit])+2*HGAP;
        }

storewincoord(wlft,wup,wrgt,wdown,&wc[nwin]); /* stores window coordinates */
cwindow(wlft,wup,wrgt,wdown,bcolor,ecolor);
window(wlft+1,wup+1,wrgt-1,wdown-1);
textcolor(tcolor);
gotoxy((wrgt-wlft)/2,(wdown-wup)/2);
cputs(notes[noit]);

```

```

bcolor++;ecolor++;

if(abs(ecolor-bcolor)==8){bcolor++;} /* avoid edge color = background color */
if(bcolor%8==0||bcolor==BLACK){bcolor++;} /* avoid background color BLACK */

nwin++;noit++;
if(nwin%7==0){noit=0;} /* if remainder zero -> end of octave */
    }

window(1,1,80,25); /* back to original window */
textcolor(YELLOW);
cursor_on();

    do{
mouse_position(&x,&y); /* show mouse position */
gotoxy(1,25);cprintf("Virtual Coords: %03d %03d",x,y);

if(!leftb_pressed()){nosound();}

else    {
for(i=0;i<42;i++){
if(x<=wc[i].rg&&x>=wc[i].lf&&y>=wc[i].up&&y<=wc[i].dw){ sound(freq[i]);break;}
    }
    }

/* looping up till right button pressed */
    } while(!rightb_pressed());

system("cls");
cursor_off();
textmode(LASTMODE);
return 0;
}

/* selects videomode - nao usada -> igual a textmode() */
void mode(mode_code)
int mode_code;
{
union REGS r;
r.h.al = mode_code;
r.h.ah = 0;
int86(0x10,&r, &r);
}

/*****
/*  mouse interfacing functions  */
*****/

/* turn-off mouse cursor */
void cursor_off()
{
int fnum;
fnum=2; /* cursor off */
cmouses(&fnum,&fnum,&fnum,&fnum);

```

```

}

/* turn-on mouse cursor */
void cursor_on()
{
int fnum;
fnum=1; /* cursor on */
cmouses(&fnum,&fnum,&fnum,&fnum);
}
/* returns V if rightbutton pressed, F otherwise */
rightb_pressed()
{
int fnum,arg2,arg3,arg4;
fnum=3; /* get button position and status */
cmouses(&fnum,&arg2,&arg3,&arg4);
return arg2&2;
}

/* returns V if leftbutton pressed, F otherwise */
leftb_pressed()
{
int fnum,arg2,arg3,arg4;
fnum=3; /* get button position and status */
cmouses(&fnum,&arg2,&arg3,&arg4);
return arg2&1;
}

/* return mouse cursor coordinates */
void mouse_position(x,y)
int *x,*y;
{
int fnum,arg2,arg3,arg4;
fnum=3; /* get button position and status */
cmouses(&fnum,&arg2,&arg3,&arg4);
*x=arg3;
*y=arg4;
}

/* init mouse */
void mouse_reset()
{
int fnum,arg2,arg3,arg4;
fnum=0; /* init mouse */
cmouses(&fnum,&arg2,&arg3,&arg4);
if(fnum!=-1){puts("No mouse driver installed!");exit(1);}
/*if(arg2!=2){puts("Two buttons mouse required!");exit(1);}*/
}

void cmouses(fnum,arg2,arg3,arg4)
int *fnum,*arg2,*arg3,*arg4;
{
union REGS regs;

regs.x.ax = *fnum;
regs.x.bx = *arg2;

```

```

regs.x.cx = *arg3;
regs.x.dx = *arg4;
int86(0x33, &regs, &regs);
*fnum=regs.x.ax;
*arg2=regs.x.bx;
*arg3=regs.x.cx;
*arg4=regs.x.dx;
}
/*****
/*  miscellaneous functions                               */
*****/

void cwindow(int lft,int up,int rgt,int dwn,int back,int edge)
{
register int i,j;
window(lft,up,rgt,dwn);
textcolor(back);
textbackground(back);
for(j=1;j<=dwn-up+1;j++){
gotoxy(1,j);
for(i=0;i<=rgt-lft;i++){putch(' ');}
}
textcolor(edge);
wedge(lft,up,rgt,dwn);
}

void wedge(int sx,int sy,int ex,int ey)
{
    register int i;

    gotoxy(1,1);putch('┌');
    gotoxy(ex-sx+1,1);putch('┐');
    gotoxy(1,ey-sy+1);putch('└');
    window(ex,ey,ex+1,ey);gotoxy(1,1);putch('┘');
    window(sx,sy,ex,ey); /* back to original window */

    for(i=2;i<=ex-sx;i++){gotoxy(i,1);putch(' ');}
    for(i=2;i<=ey-sy;i++){gotoxy(1,i);putch('||');}
    for(i=2;i<=ex-sx;i++){gotoxy(i,ey-sy+1);putch('=');}
    for(i=2;i<=ey-sy;i++){gotoxy(ex-sx+1,i);putch('||');}
}

void storewincoord(wlft,wup,wrgt,wdwn,sptr)
int wlft,wup,wdwn;
wico *sptr;
{
/* text screen (1,1,80,25); virtual screen (0,0,640,200); mouse cursor 8x8 */
/* xt=1 -> xv=0 e xt=80 -> xv=640-8=632*/
/* yt=1 -> yv=0 e yt=25 -> xv=200-8=192*/
/* 640-8 & 200-8 -> cursor mouse = 8x8 pixels */
/* coordinates transformation: virtual = 8*text - 8 */
sptr->lf=8*wlft-8;
sptr->up=8*wup-8;
sptr->rg=8*wrgt-8;
sptr->dw=8*wdwn-8;
}

```

```
}
```

18) APÊNDICE IV: PROGRAMA DEMONSTRATIVO DE CONTROLE DO MOUSE EM MODO GRÁFICO

```
/* Programa de desenho simples p/ Mouse IBM e monitor VGA */

/* EGAVGA.bgi deve estar no diretorio corrente */

/* Por Eng. Fernando César C. de Castro CREARS 48711 */
/* 15/12/1992 */

#include<dos.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>
#include<graphics.h>
#include<alloc.h>

#define ICONRIGHT 26
#define ICONDOWN 10

#define NOT_MOVED 0
#define RIGHT 1
#define LEFT 2
#define UP 3
#define DOWN 4

void cmouses();
void mouse_position();
void mouse_motion();
void cursor_on();
void cursor_off();
void mouse_reset();

main()
{
char deltax, deltay;
int x,y,xa,ya;
int graphdriver,graphmode,ncolor;
char
*idcolor[]={ "Pr", "Az", "Vd", "Ci", "Vm", "Mg", "CzC", "Mr", "CzE", "AzC", "VdC", "CiC", "VmC", "MgC", "Am",
"Br" };
```

```

int
color[]={EGA_BLACK,EGA_BLUE,EGA_GREEN,EGA_CYAN,EGA_RED,EGA_MAGENTA,EGA_LIG
HTGRAY,EGA_BROWN,EGA_DARKGRAY,EGA_LIGHTBLUE,EGA_LIGHTGREEN,EGA_LIGHTCY
AN,EGA_LIGHTRED,EGA_LIGHTMAGENTA,EGA_YELLOW,EGA_WHITE};

directvideo=1;
graphdriver=VGA;
graphmode=VGAHI;

initgraph(&graphdriver,&graphmode,"");
ncolor=WHITE;
setcolor(color[ncolor]);
setlinestyle(SOLID_LINE,0,NORM_WIDTH);

mouse_reset(); /* init mouse */
cursor_on();
outtextxy(2,2,idcolor[ncolor]); /* writes idcolor on icon */
rectangle(0,0,ICONRIGHT,ICONDOWN); /* draw icon rectangle */
setlinestyle(SOLID_LINE,0,THICK_WIDTH);

    do{
mouse_position(&x,&y); /* get mouse position */
if(!leftb_pressed()){ xa=x;ya=y;moveto(x,y);} /* tracks mouse cursor if leftb not pressed */

mouse_motion(&deltax,&deltay); /* get mouse motion */

/* if right button pressed outside icon: */
if(rightb_pressed()&&(x>=ICONRIGHT||y>=ICONDOWN)){ cursor_off();cleardevice();cursor_on();goto
Licon;}

/* if right button pressed with cursor inside icon: */
if(rightb_pressed()&&x<ICONRIGHT&&y<ICONDOWN){
ncolor++;
if(color[ncolor]>EGA_WHITE)ncolor=0;
Licon:
setcolor(EGA_BLACK);
cursor_off();
outtextxy(2,2,"___"); /* clean icon area with ascii 219 */

if(color[ncolor]==EGA_BLACK||color[ncolor]==EGA_DARKGRAY)setcolor(EGA_WHITE);
else{ setcolor(color[ncolor]);} /* avoid darkgray and black on icon */

outtextxy(2,2,idcolor[ncolor]); /* writes idcolor on icon */

setlinestyle(SOLID_LINE,0,NORM_WIDTH);
rectangle(0,0,ICONRIGHT,ICONDOWN); /* draw icon rectangle */
setlinestyle(SOLID_LINE,0,THICK_WIDTH);

cursor_on();
setcolor(color[ncolor]); /* set new color */
delay(200); /* wait 0.2 seconds */

    }

```

```

/* draws if left button is pressed: */
if(leftb_pressed()&&(deltax||deltay)){ cursor_off();linere1(x-xa,y-ya);cursor_on();xa=x;ya=y;}

        } while(!(leftb_pressed() && rightb_pressed()));
/* looping up till both buttons pressed together */

cleardevice();
closegraph();
cursor_off();
return 0;
}

/*****
/*  mouse interfacing functions  */
*****/

/* turn-off mouse cursor */
void cursor_off()
{
int fnum;
fnum=2; /* cursor off */
cmouses(&fnum,&fnum,&fnum,&fnum);
}

/* turn-on mouse cursor */
void cursor_on()
{
int fnum;
fnum=1; /* cursor on */
cmouses(&fnum,&fnum,&fnum,&fnum);
}

/* returns V if rightbutton pressed, F otherwise */
rightb_pressed()
{
int fnum,arg2,arg3,arg4;
fnum=3; /* get button position and status */
cmouses(&fnum,&arg2,&arg3,&arg4);
return arg2&2;
}

/* returns V if leftbutton pressed, F otherwise */
leftb_pressed()
{
int fnum,arg2,arg3,arg4;
fnum=3; /* get button position and status */
cmouses(&fnum,&arg2,&arg3,&arg4);
return arg2&1;
}

/* return mouse cursor coordinates */
void mouse_position(x,y)
int *x,*y;

```

```

{
int fnum,arg2,arg3,arg4;
fnum=3; /* get button position and status */
cmouses(&fnum,&arg2,&arg3,&arg4);
*x=arg3;
*y=arg4;
}
/* return path direction */
void mouse_motion(deltax,deltay)
char *deltax,*deltay;
{
int fnum,arg2,arg3,arg4;
fnum=11; /* get movement direction */
cmouses(&fnum,&arg2,&arg3,&arg4);

if(arg3>0){ *deltax=RIGHT;}
else if(arg3<0) { *deltax=LEFT;}
else { *deltax=NOT_MOVED;}

if(arg4>0){ *deltay=DOWN;}
else if(arg4<0) { *deltay=UP;}
else { *deltay=NOT_MOVED;}

}

/* init mouse */
void mouse_reset()
{
int fnum,arg2,arg3,arg4;
fnum=0; /* init mouse */
cmouses(&fnum,&arg2,&arg3,&arg4);
if(fnum!=-1){puts("No mouse driver installed!");exit(1);}
/*if(arg2!=2){puts("Two buttons mouse required!");exit(1);}*/
}

void cmouses(fnum,arg2,arg3,arg4)
int *fnum,*arg2,*arg3,*arg4;

{
union REGS regs;

regs.x.ax = *fnum;
regs.x.bx = *arg2;
regs.x.cx = *arg3;
regs.x.dx = *arg4;
int86(0x33, &regs, &regs);
*fnum=regs.x.ax;
*arg2=regs.x.bx;
*arg3=regs.x.cx;
*arg4=regs.x.dx;

}

```

19) APÊNDICE V: PROGRAMA DEMONSTRATIVO DE USO DE ESTRUTURAS PARA DETERMINAÇÃO DO CENTRO GEOMETRICO DE UM POLIEDRO SÓLIDO

```
/*Calcula Centro Geometrico - Versao 1*/
#include <stdio.h>

struct coord{
float x;
float y;
float z;
};

void main()
{
int i,teste;
FILE *fp;
struct coord vert[10];
float *cg;

fp=fopen("vertices.dat","rt");
if(!fp){puts("nao posso abrir arquivo");exit(1);}

for(i=0;i<10;i++){
teste=fscanf(fp,"%f%f%f",&vert[i].x,&vert[i].y,&vert[i].z);
if(teste==-1){puts("achei EOF!");exit(1);}
}
cg=med(vert); /* o nome de qualquer área de armazenamento é o endereço */
/* inicial da área na memória!!! */
printf("%f %f %f",cg[0],cg[1],cg[2]);
}

float *med(V)
struct coord *V;
{
int i;
float *cg;

cg=(float *)malloc(3*sizeof(float));
if(!cg){puts("memoria insuficiente");exit(1);}
cg[0]=cg[1]=cg[2]=0;

for(i=0;i<10;i++)
{
cg[0]=cg[0]+V[i].x; /* V é um ponteiro para uma MATRIZ DE ESTRUTURAS e nao */
cg[1]=cg[1]+V[i].y; /* para uma VARIÁVEL DE ESTRUTURA !!! */
cg[2]=cg[2]+V[i].z;
}

cg[0]=cg[0]/10;
cg[1]=cg[1]/10;
cg[2]=cg[2]/10;
```

```

return cg;
}
/*Calcula Centro Geometrico - Versao 2 */
#include <stdio.h>

struct coord{
float x[10];
float y[10];
float z[10];
};

float *med();

void main()
{
int i, teste;
FILE *fp;
struct coord vert;
float *cg;

fp=fopen("vertices.dat", "rt");
if(!fp){puts("nao posso abrir arquivo");exit(1);}

for(i=0; i<10; i++){
teste=fscanf(fp, "%f%f%f", &vert.x[i], &vert.y[i], &vert.z[i]);
if(teste==-1){puts("achei EOF!");exit(1);}
}

cg=med(&vert); /* vert é uma VARIAVEL DE ESTRUTURA com matrizes */
/* como membros e nao uma MATRIZ DE ESTRUTURAS!!! */

printf("%f %f %f", cg[0], cg[1], cg[2]);
}
float *med(V)
struct coord *V;
{
int i;
float *cg;

cg=(float *)malloc(3*sizeof(float));
if(!cg){puts("memoria insuficiente");exit(1);}
cg[0]=cg[1]=cg[2]=0;

for(i=0; i<10; i++)
{
cg[0]=cg[0]+V->x[i]; /* V é um PONTEIRO PARA UMA VARIAVEL DE ESTRUTURA */
cg[1]=cg[1]+V->y[i]; /* e x é um membro da VARIAVEL DE ESTRUTURA apontada */
cg[2]=cg[2]+V->z[i]; /* por V o qual é uma MATRIZ !!! */
}

cg[0]=cg[0]/10;
cg[1]=cg[1]/10;
cg[2]=cg[2]/10;
}

```

```
return cg;
}
```

O arquivo VERTICES.DAT deve conter as coordenadas dos vértices do poliedro (no caso, 10 vértices) no formato numérico de 3 colunas, cada uma representando respectivamente x,y,e z. Por exemplo:

```
0  7  9
1  6  8
2  5  7
3  4  6
4  3  5
5  2  4
6  1  3
7  0  2
8  8  1
9  9  0
```

20) APÊNDICE VI: PROGRAMA DEMONSTRATIVO DO USO DE PONTEIROS “FAR” PARA ESCRITA DIRETA NA MEMORIA DE VIDEO

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

```
int main(void)
{
    int gd, gm, i;
    unsigned int far *screen;

    detectgraph(&gd, &gm);
    if (gd == HERCMONO)
        screen = (unsigned int *) MK_FP(0xB000, 0);
    else
        screen = (unsigned int *) MK_FP(0xB800, 0);
    for (i=0; i<26; i++)
        screen[i] = 0x0700 + ('a' + i);
    return 0;
}
```

21) APÊNDICE VII: PROGRAMA DEMONSTRATIVO DO USO DE ALOCAÇÃO DINÂMICA DE MEMÓRIA PARA OPERAÇÕES SIMPLES COM MATRIZES BIDIMENSIONAIS

```
/*Programa demonstra operacoes de soma , multiplicao e I/O de matrizes*/
/* Utilizando técnicas de alocação dinamica de memoria */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <conio.h>
#include <time.h>
#include <dos.h>

void matmult(unsigned,unsigned,unsigned,double **,double **,double **);
void matsum(unsigned,unsigned,double **,double **,double **);
void matfprint(unsigned, unsigned, double **, char *);
void matprint(unsigned,unsigned,double **);
double **matinit(unsigned,unsigned);

void main(void)
{
double **A;
double **B;
double **C;
unsigned int ai,aj,bi,bj;
register unsigned int i,j;
double dtemp;
char op;

puts("\nEste programa realiza A [op] B, sendo A e B matrizes e [op] as operações + ou *.");
do{
printf("Operação[*/+]?");
op=getche();
printf("\n");
if(toupper(op)=='F'){puts("Fim.");exit(0);}
}while(op!='*'&&op!='+');

LBL01:
printf("Número de linhas da matriz A?");
scanf("%u",&ai);
printf("Número de colunas da matriz A?");
scanf("%u",&aj);

A=matinit(ai,aj);
for(i=0;i<=ai-1;i++){
printf("Entre linha %d da matriz A:\n",i+1);
for(j=0;j<=aj-1;j++){
printf("A[%u,%u]? ",i+1,j+1);
scanf("%lG",&dtemp);
A[i][j]=dtemp;
}
}

printf("Número de linhas da matriz B?");
```

```

scanf("%u",&bi);
printf("Número de colunas da matriz B?");
scanf("%u",&bj);
if(op=='*'&&aj!=bi){
putch(7);/* ascii BEL */
puts("\nNúmero de colunas de A deve ser igual ao número de linhas de B!");
goto LBL01;
}
if(op=='+'&&(ai!=bi||aj!=bj)){
putch(7);/* ascii BEL */
puts("\nDimensoes de A e B devem ser idênticas!");
goto LBL01;
}

```

```

B=matinit(bi,bj);
for(i=0;i<=bi-1;i++){
printf("Entre linha %d da matriz B:\n",i+1);
for(j=0;j<=bj-1;j++){
printf("B[%u,%u]? ",i+1,j+1);
scanf("%lG",&dtemp);
B[i][j]=dtemp;
}
}

```

```

if(op=='*'){
C=matinit(ai,bj);
matmult(ai,aj,bj,A,B,C);
matprint(ai,bj,C);
matfprint(ai,bj,C,"matprod.dat");
}
else{
C=matinit(ai,aj);
matsum(ai,aj,A,B,C);
matprint(ai,aj,C);
matfprint(ai,aj,C,"matsoma.dat");
}
}

```

```

double **matinit(l,c)
unsigned int l,c;
{
double **a;
register unsigned int i;

a=(double **)malloc(l*sizeof(double));
if(!a) {LMEM:puts("Memória insuficiente!");exit(1);}
for(i=0;i<=l-1;i++){
a[i]=(double *)malloc(c*sizeof(double));
if(!a[i]) goto LMEM;
}

return a;
}
void matprint(l,c,mat)

```

```

unsigned int l,c;
double **mat;
{
register unsigned int i,j;

for(i=0;i<=l-1;i++){
for(j=0;j<=c-1;j++){
printf("%5.2lG\t",mat[i][j]);
}
printf("\n");
}

void matfprint(l,c,mat,arq)
unsigned int l,c;
double **mat;
char arq[80];
{
register unsigned int i,j;
FILE *out;

if((out=fopen(arq,"w"))==NULL)
{printf("Impossivel abrir arquivo %s!",arq);exit(1);}

for(i=0;i<=l-1;i++){
for(j=0;j<=c-1;j++){
fprintf(out,"%5.2lG\t",mat[i][j]);
}
fprintf(out,"\n");
}
fclose(out);
}

void matmult(unsigned l1,unsigned c1,unsigned c2,double ** M1,double **M2,double **MP)
{
register unsigned int i,j,k; /* k-> c1=l2 */

for(i=0;i<=l1-1;i++){ /* i e j indexam MP */
for(j=0;j<=c2-1;j++){

MP[i][j]=0; /* inicializa (zera) MP[i][j] */

for(k=0;k<=c1-1;k++){
MP[i][j]+=M1[i][k]*M2[k][j];
}

}
}

void matsum(unsigned l,unsigned c,double **M1,double **M2,double **MS)

```

```
{  
register unsigned int i,j;  
  
for(i=0;i<=l-1;i++){  
for(j=0;j<=c-1;j++){  
  
MS[i][j]=M1[i][j]+M2[i][j];  
        }  
    }  
  
}
```

21) APÊNDICE VIII: PROGRAMA EXEMPLO PARA GERAÇÃO DE GRAFICOS CARTESIANOS NA TELA A PARTIR DE UM ARQUIVO NUMERICO.

```

/* lê um arquivo numérico de nome a especificar no formato de 2 colunas */
/* e plota na tela */

/* Compilar no modelo de memória SMALL */

/* Necessita ser compilado e linkado através de arquivo de projeto o qual referencia */
/* este arquivo e os arquivos LITT.OBJ e EGAVGA.OBJ obtidos com o utilitário */
/* BGIOBJ.EXE do diretório /BGI */

#include<dos.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<math.h>
#include<conio.h>
#include<alloc.h>
#include<graphics.h>

#define MONITOR VGA
#define MODO VGAHI
#define MAXX 639 /* largura da tela em pixels */
#define MAXY 479 /* altura da tela em pixels */
#define MME 5 /* Metade do tamanho dos Marcadores de Escala */
#define BUFSIZE 4096 /* Tamanho do buffer de arquivos */

void main(argc,argv)
int argc;
char *argv[];
{
FILE *in;
float *x;
float *y;
float delta,accum, topx, topy, botx, boty, Ax, Bx, Ay, By;
int i, teste,numlido,th,tw;
int gdriver, gmode, errorcode;
char num[80];

if(argc!=2){puts("Digite o nome do arquivo!");exit(1);}

if((in=fopen(argv[1],"rt"))==NULL){ /* sempre texto c/ fscanf() e fprintf() */
puts("Nao posso abrir arquivo!");
exit(1);
}

if (setvbuf(in, NULL, _IOFBF, BUFSIZE) != 0) /* cria e conecta buffer a 'in' */
puts("Nao posso estabelecer buffer para arquivo a ser lido!");

x=(float *)malloc(sizeof(float));
if(!x){LBLMEM:puts("Memória insuficiente!");exit(1);}
y=(float *)malloc(sizeof(float));
if(!y){goto LBLMEM;}

i=0;

while(1){

```

```

teste=fscanf(in,"%f%f",x+i,y+i);
if(teste==-1)break; /* testa EOF: fscanf() retorna -1 quando encontra-o */
i++;

x=realloc(x,((unsigned)(i+1))*sizeof(float));
if(!x){LBLALOC:puts("Memória insuficiente!\nNao posso ler mais dados!");exit(1);}
y=realloc(y,((unsigned)(i+1))*sizeof(float));
if(!y){goto LBLALOC;}
}
fclose(in);
numlido=i-1;

/* deteta top e bottom de x e y */
topx=topy=botx=boty=0;
for(i=0;i<=numlido;i++){

if(x[i]>topx){topx=x[i);}
if(x[i]<botx){botx=x[i);}

if(y[i]>topy){topy=y[i);}
if(y[i]<boty){boty=y[i);}
}

gdriver=MONITOR;
gmode=MODO;

/* inicializa modo grafico */
errorcode = registerbgidriver(EGAVGA_driver); /* registra driver EGAVGA */

/* reporta qualquer erro de registro */
if (errorcode < 0)goto L_EGR;

errorcode = registerbgifont(small_font); /* registra fonte 'Small_Font' */

/* reporta qualquer erro de registro */
if (errorcode < 0)goto L_EGR;

initgraph(&gdriver, &gmode, "");

/* le resultado da inicializacao */

errorcode = graphresult();

if (errorcode != grOk) /* ocorreu um erro */
{
L_EGR:
printf("Erro gráfico: %s\n", grapherrormsg(errorcode));
printf("Pressione qualquer tecla para parar.");
getch();
exit(1);/* retorna p/ DOS com codigo de erro 1(nao erro = 0)*/
}

/*

```

Transformação Linear CoordReal ---> CoordVideo:

$$\text{CoordVideo} = A * \text{CoordReal} + B$$

Sejam topx e topy os valores máximos das coordenadas reais x e y e sejam botx e boty os valores mínimos das coordenadas reais x e y. Sejam MAXX e MAXY respectivamente os valores máximos das coordenadas de vídeo (MAXX=639 e MAXY=479 p/ monitor VGA em modo VGAHI).

Para a coordenada Vertical Y temos o seguinte sistema de equações:

$$\begin{aligned} 0 &= A_y * \text{topy} + B_y \\ \text{MAXY} &= A_y * \text{boty} + B_y \end{aligned}$$

Resolvendo p/ A_y e B_y temos:

$$\begin{aligned} A_y &= \text{MAXY} / (\text{boty} - \text{topy}) \\ B_y &= \text{MAXY} / (1 - \text{boty} / \text{topy}) \end{aligned}$$

Para a coordenada horizontal X temos o seguinte sistema de equações:

$$\begin{aligned} 0 &= A_x * \text{botx} + B_x \\ \text{MAXX} &= A_x * \text{topx} + B_x \end{aligned}$$

Resolvendo p/ A_x e B_x temos:

$$\begin{aligned} A_x &= \text{MAXX} / (\text{topx} - \text{botx}) \\ B_x &= \text{MAXX} / (1 - \text{topx} / \text{botx}) \end{aligned}$$

*/

/* Calcula Ax,Bx,Ay e By */

$$\begin{aligned} A_x &= \text{MAXX} / (\text{topx} - \text{botx}); \\ A_y &= \text{MAXY} / (\text{boty} - \text{topy}); \end{aligned}$$

if(botx==0){Bx=0;} /* Para evitar divisao por zero */
else{Bx=MAXX/(1-topx/botx);}

if(topy==0){By=0;} /* Para evitar divisao por zero */
else{By=MAXY/(1-boty/topy);}

/* Faz a transformação de coordenadas */

```
for(i=0;i<=numlido;i++){  
x[i]=Ax*x[i]+Bx;  
y[i]=Ay*y[i]+By;  
}
```

/* plota na tela */

setttextstyle(SMALL_FONT, HORIZ_DIR, 4);

setcolor(EGA_LIGHTBLUE);

```

for(i=0;i<=numlido-1;i++){
line((int)x[i],(int)y[i],(int)x[i+1],(int)y[i+1]);
}

/* desenha linha vertical e horizontal da escala */

setcolor(EGA_YELLOW);
line(MAXX/2,0,MAXX/2,MAXY);

setcolor(EGA_LIGHTRED);
line(0,MAXY/2,MAXX,MAXY/2);

/* desenha 11 marcadores numéricos na escala vertical */

setcolor(EGA_YELLOW);

th=textheight("W");
tw=textwidth("W");
delta=(topy-boty)/10;

for(i=0,accum=topy;i<MAXY;i+=(MAXY+1)/10,accum-=delta){ /* coloca os 10 1ºs marcadores */
    sprintf(num,"%4.2G",accum);
    if(i!=(MAXY+1)/2){outtextxy(MAXX/2+MME+tw/2,i,num);
        line(MAXX/2-MME,i,MAXX/2+MME,i);}
    else{outtextxy(MAXX/2+MME+tw/2,i-2*th,num);} /* para evitar superposição c/ eixo horizontal */

line(MAXX/2-MME,MAXY,MAXX/2+MME,MAXY); /* coloca o 11º último marcador */
sprintf(num,"%4.2G",accum);
outtextxy(MAXX/2+MME+tw,MAXY-th,num);

/* desenha 11 marcadores numéricos na escala horizontal */

setcolor(EGA_LIGHTRED);

delta=(topx-botx)/10;

for(i=0,accum=botx;i<MAXX;i+=(MAXX+1)/10,accum+=delta){ /* coloca os 10 1ºs marcadores */
    sprintf(num,"%-4.2G",accum); /* justifica a esquerda */
    if(i!=(MAXX+1)/2){
        line(i,MAXY/2-MME,i,MAXY/2+MME);
        if(i!=0){outtextxy(i,MAXY/2+MME+th,num);}
        else{outtextxy(i,MAXY/2+MME+th,num);}
    }
    else{outtextxy(i+tw/2,MAXY/2+MME+th,num);} /* para evitar superposição c/ vertical */
    line(i-1,MAXY/2-MME,i-1,MAXY/2+MME);
}
}
line(MAXX,MAXY/2-MME,MAXX,MAXY/2+MME); /* coloca o 11º último marcador */
sprintf(num,"%4.2G",accum);
outtextxy(MAXX-textwidth(num),MAXY/2+MME+th,num);

getch();
cleardevice();
closegraph();
}

```

22) REFERÊNCIAS BIBLIOGRAFICAS:

[1] - Turbo C - Guia do Usuário
Herbert Schildt
Ed. Mc Graw Hill

[2] - Turbo C Avançado
Herbert Schildt
Ed. Mc Graw Hill

[3] - Dominando o Turbo C
Stan Kelly-Bootle
Ed. Ciência Moderna

[4] - MSDOS Avançado - Guia Microsoft p/ programação em C
Ray Duncan
Makron Books