UERGS – UNIVERSIDADE ESTADUAL DO RIO GRANDE DO SUL

CURSO SUPERIOR DE TECNOLOGIA EM AUTOMAÇÃO INDUSTRIAL

INFORMÁTICA II

LINGUAGEM DE PROGRAMAÇÃO "C"

PROF. ANDRÉ LAWISCH

NOVO HAMBURGO, MARÇO DE 2002

ÍNDICE

VISÃO GERAL

HISTÓRICO

BCPL ⟶ “B” ⟶ “C”

Simplificação

Implementa funções de baixo nível:
- bit, byte
- endereços

A linguagem “B” foi desenvolvida na década de 70 por Ken Thompson, a partir de uma simplificação da linguagem BCPL (desenvolvida por Martin Richards). A linguagem “C” surgiu de uma implementação da “B” no que se refere a funções de baixo nível. Projetada em 1972, no laboratório da Bell, por Dennis Ritchie, a “C” é tida por uma linguagem de médio nível, pois implementa instruções a nível de bit e endereços (o que só era feito através do assembler) e instruções de alto nível (o que ocorre na maioria das linguagens de programação - COBOL, BASIC).

- Portabilidade: a maioria das implementações comerciais de "C" diferem um pouco da definição original de Kernighan e Ritchie. Isto gerou algumas incompatibilidades entre diferentes implementações da linguagem. Para resolver este problema, o comitê ANSI (American National Standards Institute) iniciou um trabalho de padronização que foi seguido, em grande parte, pelas implementações comerciais;
- Conjunto básico de tipo de dados (char, int, float, double, void);
- Instruções para programação estruturada: laço de controle, decisão;
- Possui um pequeno número de comandos - 32 (BASIC - em torno de 100);
- Protótipos de função: a maioria dos recursos dependentes do computador são relegados as bibliotecas de funções. Toda versão de "C" é acompanhada de um conjunto de bibliotecas de funções que são relativamente padronizadas, permitindo que cada função individual das bibliotecas seja acessada da mesma maneira de uma versão para outra do "C";
- Possui apontadores e aritmética de endereços.

APLICAÇÕES

- Desenvolvimento de aplicações em software básico;
- Controle e interfaceamento.

COMPILADOR X INTERPRETADOR

Para que se possa entender melhor o que é um compilador ou interpretador, são necessários alguns conceitos:

- Linguagem de máquina: conjunto de instruções detalhadas e obscuras que controlam o circuito interno do computador. Poucos programas são escritos nesta linguagem. Ela é muito trabalhosa e a maioria dos computadores possuem seus próprios conjuntos de instruções. Pode-se dizer que, para se programar em linguagem de máquina é necessário um amplo conhecimento do hardware utilizado;
- Linguagem de alto nível: conjunto de instruções mais compatíveis com a linguagem humana e com o modo do ser humano pensar. Estas linguagens, na sua maioria, são de uso genérico e não é necessário ter conhecimento do hardware utilizado.

Uma única instrução em linguagem de alto nível poderá equivaler a várias instruções em linguagem de máquina, mas pode ser executado em diferentes computadores com pouca ou nenhuma alteração. Por essas razões, o uso da linguagem de alto nível oferece algumas vantagens: simplicidade (proximidade da linguagem de alto nível com a linguagem humana), uniformidade (padronização) e portabilidade (independência do hardware utilizado).

Ainda é necessário fazer a seguinte relação:

- Todo programa escrito em linguagem de alto nível é chamado de programa-fonte, e;
- O programa em linguagem de máquina, escrito ou resultante, é chamado de programa-objeto.

Todo o programa-fonte, para ser executado, deve obrigatoriamente, de alguma forma, ser traduzido para programa-objeto. Neste momento é que são utilizados os compiladores e/ou interpretadores.

Interpretador: lê a primeira instrução do programa, faz uma consistência de sua sintaxe e se não houver erro, converte-a para linguagem de máquina para executá-la. Este

processo é realizado de forma seqüencial para as demais instruções até que a última seja executada ou a consistência apresenta algum erro. O interpretador precisa estar presente todas as vezes que vamos executar o nosso programa e o trabalho de checagem da sintaxe deverão ser repetidos.

Compilador: lê a primeira instrução do programa, faz uma consistência de sua sintaxe e se não houver erro, converte-a para linguagem de máquina e, em vez de executá-la, segue para a próxima instrução repetindo o processo até que a última seja alcançada ou a consistência apresenta algum erro. Se não houver erros, o compilador gera um programa em disco com a extensão .OBJ com as instruções já traduzidas. Este programa não pode ser executado até que sejam agregadas a ele rotinas em linguagem de máquina que lhe permitirão a sua execução. Este trabalho é feito por um programa chamado linkeditor, que além de juntar as rotinas necessárias ao programa .OBJ, cria um produto final em disco com a extensão .EXE. Neste momento não é mais necessária a presença do compilador, pois todo o programa já esta traduzido para linguagem de máquina e armazenado em disco, basta rodá-lo.

Desta forma, podemos dizer que o compilador é mais rápido em tempo de execução (instante em que o programa é executado), enquanto que o interpretador é mais rápido em tempo de programação (instante em que o programa é editado). Sabendo-se disso, é mais conveniente desenvolver um novo programa utilizando um interpretador e o compilador é reservado para gerar a versão final do referido programa.

ESTRUTURA DE UM PROGRAMA

Todo programa escrito em linguagem C necessita de uma estrutura mínima para poder ser compilado e não gerar erro. Como C tem poucos comandos, a grande maioria das ações são realizadas através das funções. Desta forma, a primeira declaração deve ser a das bibliotecas de funções (**#include**), seguindo das declarações **#define** e as variáveis globais (públicas). O próximo passo é declarar as funções e, dentre ela devemos destacar a obrigatoriedade da função principal chamada **main()**. As demais funções devem ser declaradas antes ou após a principal. Vamos adotar o padrão de declarar as funções adicionais antes da função principal.

```
/*ESTRUTURA DE UM PROGRAMA EM C*/
/* declaração dos arquivos de cabeçalho e definições */
#include <stdio.h>
#define PI 3.1415

/* definição das variáveis globais */

/*lista de prototipos */
void le_vet(double VET[]);
void imp_vet(double VET[]);

/* funcao para leitura dos valores do vetor */
void le_vet(double *VET)
{
int I;
for(I=0;I<10;I++)
    {
     printf("%d -->",I);
     scanf("%lf",&VET[I]);
     }
}
/* funcao para imprimir o vetor lido */
void imp_vet(double *VET)
{
     int I;
     printf("\n\nVetor digitado:\n");
     for(I=0;I<10;I++)
     printf("\n%lf   ",VET[I]);
}

/* função principal do programa */
void main()
{
   /* define variáveis locais utilizadas no programa */
   double VET[10];
   /*chamada de funções */
   le_vet(VET);
   imp_vet(VET);
}
```

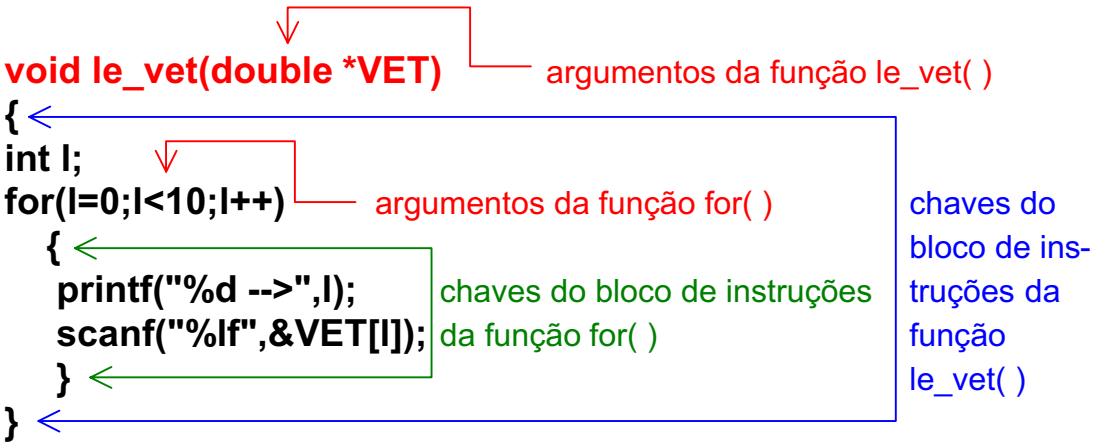
Assim como o programa, as funções, tem uma estrutura padrão. Elas devem conter:

- ***header*** - nome da função seguido por uma lista opcional de argumentos entre parênteses;
- ***lista de declaração de argumentos*** - se o header tiver argumentos;
- ***bloco de instruções*** - constitui o restante da função.

```
void le_vet(double *VET)/* header da função com declaração
                           de argumentos */
{
int I;                      /* declaração das variáveis */
for(I=0;I<10;I++)           /* bloco de instruções */
   {
    printf("%d -->",I);
    scanf("%lf",&VET[I]);
    }
}
```

Os argumentos, também chamados de parâmetros, são símbolos que representam informações passadas entre as funções e o restante do programa.

O bloco de instruções deve estar declarado entre chaves (**{ }**). As chaves podem conter bloco de expressões (**devem terminar por ponto e vírgula**) ou outro bloco de instruções, aninhados um dentro do outro.



Comentários: todo o programa deve ser comentado. Eles são necessários na identificação das características principais do programa ou em explicações sobre a lógica. Os comentários são colocados entre os delimitadores /* e */ ou simplesmente // no início de cada linha comentário.

/* isto é um comentário */
// isto é um comentário

Exemplo de um programa C:

```
/* programa para calcular a área de um círculo */

#include <stdio.h> /* declaração do arquivo de cabeçalho ( biblioteca de função) */
#define PI 3.14159  /* declaração da constante PI e seu valor */

float processa( float r)     /* header da função processa */
{                            /* início da função processa */
        float a;             /* declaração da variável local a */
        a = PI * r * r;      /* declaração da expressão que calcula a área */
        return(a);           /* retorno da função processa */
}                            /* fim da função processa */


main( )                      /* declaração da função principal */
{                            /* início da função principal */
        float area, raio;    /* declaração das variáveis locais */
        printf("Raio = ?");  /* exibe na tela a mensagem que esta entre aspas */
        scanf("%f", &raio);/* lê através do teclado, o valor para variável raio */
        area = processa(raio); /* chama a função processa e transfere para ela o valor */
                               do raio, que é recebido pela função com o nome de r */
        printf("Área = %f", area);  /* exibe na tela a mensagem e o conteúdo da variável
                                       area */
}                        /* fim da função principal e,  por conseqüência do programa */
```

CARACTERÍSTICAS DE UM PROGRAMA

Abaixo são comentadas algumas características importantes de um programa bem escrito. Elas não se aplicam somente a programas escritos em C, mas em qualquer linguagem e servem de referência para o momento em que começarmos a escrever nosso próprios programas.

Integridade:

Diz respeito aos cálculos. De nada adianta escrevermos um belo programa se os cálculos não forem executados corretamente.

Clareza:

Refere-se a legibilidade do programa. Se o programa for escrito de forma clara, fica fácil para qualquer programador (inclusive para o autor) possa entender a lógica utilizada. A clareza é apoiada pelos comentários.

Simplicidade:

"Cada pessoa analisa um problema de forma diferente e há uma tendência de complicar a sua solução. Desta forma deve-se pensar muito na solução desejada, analisando todas as possibilidades, ao invés de utilizar a primeira solução que lhe vier na cabeça. Caso contrário, pode-se perder muito tempo tentando entendê-lo do que criando-o novamente com as alterações desejadas." Esta referência feita no material sobre algoritmos exemplifica como devemos agir para que possamos escrever programas simples, o que facilita a clareza e exatidão.

Eficiência:

Executar no menor tempo o que foi proposto e utilizando o mínimo de recursos disponíveis, sem comprometer a clareza e a simplicidade.

Modularidade:

Programas complexos e extensos podem ser divididos em módulos ou programas menores. Estes módulos ou programas menores são implementados através das funções em C. Estas implementações melhoram a integridade e a clareza do programa e facilita alterações posteriores.

Generalidade:

Programa específico resolve problema específico e sua aplicação se torna restrita. Esta característica refere-se a visão que o programador deve ter quando da hora de projetar a solução de um problema. Quanto maior for a generalidade da solução, maior será sua

aplicação. Normalmente, uma generalização razoável pode ser obtida com um pouco de esforço adicional de programação.

DECLARAÇÃO DE VARIÁVEIS

A declaração de variáveis é uma instrução para reservar uma quantidade de memória necessária para armazenar o tipo especificado. Em C, todas as variáveis devem ser declaradas e, jamais declare uma variável que não será utilizada. Se existir mais de uma variável do mesmo tipo, elas poderão ser declaradas de uma única vez, separando seus nomes por vírgula.

Ex.: int a, b, c;

Tipos de variáveis:

O tipo de variável informa a quantidade de memória, em bytes, que este irá ocupar e a forma como seu conteúdo será armazenado.

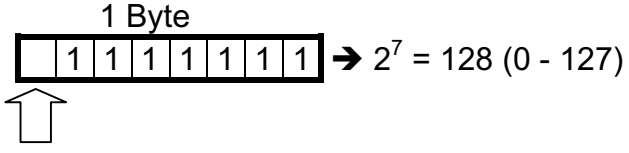
Na linguagem de programação C existem 5 tipos básicos de dados. São eles:

TIPO	BITs	BYTEs	DESCRIÇÃO	DOMÍNIO
Void	0	0	sem argumento e não retorna valor	Sem valor
Char	8	1	caracter isolado	-128 a 127
Int	16	2	valor inteiro	-32.768 a 32.767
Float	32	4	número ponto-flutuante (nº contendo um ponto decimal e/ou um expoente)	3.4E-38 a 3.4E+38
Double	64	8	número ponto-flutuante de dupla precisão (mais algarismos significativos e um expoente que pode ter mais magnitude)	1.7E-308 a 1.7E+308

Variável void:

Variável char:

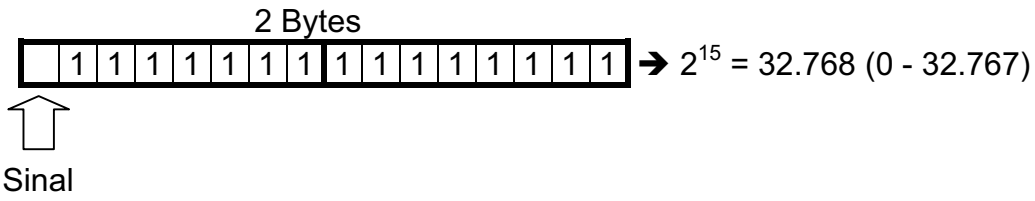
Armazena um caracter em decimal, de acordo com o código da tabela ASCII.



O bit mais significativo é reservado para o sinal. 0 = positivo e 1 = negativo

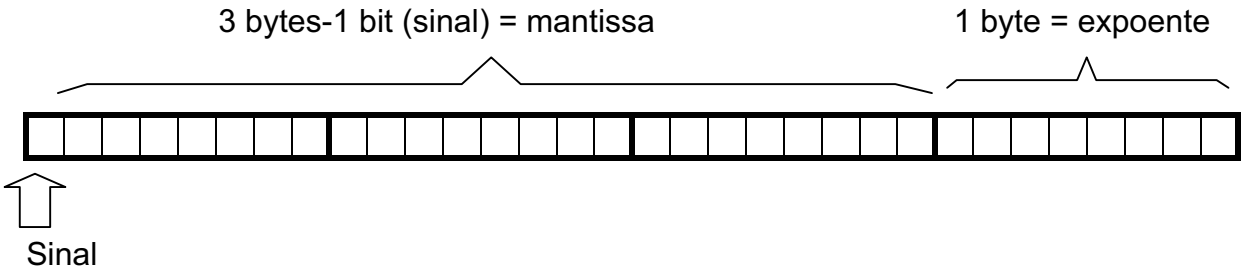
Variável int:

Armazena um valor pertencente ao conjunto dos números inteiros do intervalo de - 32.768 a 32.767.



Variável float:

Armazena um número pertencente ao conjunto dos números reais ou conjunto dos números com ponto flutuante. Os números são armazenados em duas partes na memória. A mantissa representa o número em ponto flutuante e o expoente representa a potência em que o número será aumentado.



Por exemplo, o número 125846 é representado da seguinte forma: .125846e6

.125846 é a mantissa e 6 é o expoente ➔ .125846 x 10^6 ➔ 125846

Variável double:

Tem o mesmo funcionamento da variável float, mas com magnitude maior.

Modificadores de tipo básico de dados:

Estes modificadores são aplicados aos tipos básicos char e int, com exceção do long que pode ser aplicado ao tipo double.

Signed:

Define um tipo básico de dados com sinal, aplicado aos tipos char e int. A declaração signed int é possível, mas redundante, pois o padrão de int é com sinal. Já, a declaração de

signed char poder ter efeito para algumas implementações de C que utilizam o padrão char sem sinal.

signed char ➔ -128 a 127

signed int ➔ -32.768 a 32.767

Unsigned:

Aplicado aos tipos básicos int e char, converte o bit mais significativo, de sinal para dados. Desta forma, serão representados apenas domínios positivos.

unsigned char ➔ 0 a 255

unsigned int ➔ 0 a 65.535

Long e short:

Estes modificadores podem ter um efeito diferente do esperado. Eles mudam o tipo de dados int com relação ao seu tamanho. Tudo depende do tamanho da palavra utilizada pelo computador.

Basicamente, o modoficador short tem o mesmo efeito do int.

short int = int ➔ -32.768 a 32.767

unsigned short int = unsigned int ➔ 0 a 65.535

O modificador long é aplicado aos tipos int, float e double, dependendo da implementação.

long int ➔ -2.147.483.648 a 2.147.483.647 (quatro bytes)

unsigned long int ➔ 0 a 4.294.967.295

long float = double (padrão ANSI)

long double ➔ dez dígitos de precisão (≅ 80 bits)

Nome de identificadores:

Identificadores são os nomes das variáreis, funções, rótulos e outros objetos definidos pelo usuário.

Observações:

- O primeiro caracter deve ser letra e, os demais podem ser letras, números e o caracter especial sublinhado;
- Em C, letras maiúsculas e minúsculas são tratadas diferentemente;
- Não pode ser igual a uma palavra-chave, uma função escrita pelo programador ou as que estão contidas nas bibliotecas C.

Onde são declaradas:

1. Dentro das funções - variável local/privada;
2. Na definição dos parâmetros da função - parâmetros formais;
3. Fora das funções - variável global/pública.

Variáveis locais:

Só existem dentro das funções ou, existem apenas enquanto o bloco de código que as criou está sendo executado (é criada na entrada e destruída na saída do seu bloco)

```
{
        bloco de código
}
```

Exemplo:

```
void f(void)
{
   int t;
   scanf("%d", &t);
   if (t==1)
   {
      char s[80];
      gets(s);
   }
}
```

A variável **s** só existe dentro deste bloco.

Parâmetros formais:

São as variáveis que recebem os argumentos de uma função. Dentro de uma função, elas se comportam como qualquer outra variável local.

Observação: Se for utilizado a declaração dos protótipos da função, você deve ter certeza que os parâmetros formais declarados são do mesmo tipo dos argumentos declarados na chamada da função.

Exemplo:

```
#include <stdio.h>          /* declaração do arquivo de cabeçalho*/

int mul_var(int, int);      /* declaração do protótipo mul_var */

int mul_var(int x, int y)   /* declaração da função mul_var */
{
```

```
        int z;
        z = x * y;
        return(z);          /* retorna o valor de z */
}

void main(void)             /* declaração da função main()*/
{
        int a, b, mul;
        printf("Digite 2 valores:\n");
        scanf("%d  %d", &a, &b);
        mul = mul_var( a, b);       /* chamada da função mul_var*/
        printf("\n %d ", mul);
}
```

Parâmetros formais da função

Argumentos da função, devem ser do mesmo tipo do parâmetro formal

Variáveis globais/públicas:

Estas variáveis são declaradas fora das funções, ou seja, elas são declaradas logo após as declarações #include.

Exemplo:

```
#include <stdio.h>   /* declaração do arquivo de cabeçalho */

int  a,b,c;
char s;

main( )              /* declaração da função principal */
{                    /* início da função principal */

}                    /* fim da função principal */
```

Variáveis

Modificadores ou quantificadores de tipo de acesso:

Os modificadores ou quantificadores **const** e **volatile** controlam como as variáveis podem ser acessadas ou modificadas.

const:

As variáveis declaradas com este tipo de modificador não podem ser alteradas pelo programa, mas pode ser modificada por algo externo ao programa (dispositivo de hardware). A variável const pode receber um valor inicial de uma declaração explicita ou de um recurso dependente de hardware.

Ex.: const int a = 10;

volatile:

Este modificador é utilizado para informar ao compilador que o valor da variável pode ser alterado a qualquer momento, não apenas pelo programa, mas também por interrupções ou por outros efeitos externos.

Ex.: volatile int x;

Modificadores de tipo de classe de armazenamento:

Estes modificadores são utilizados para alterar o modo como um compilador "C" aloca espaço de armazenamento para as variáveis ou para funções.

auto:

Indica que a variável é automática e de escopo local, ou seja, ela existe somente do bloco que a criou e será destruída quando este bloco for abandonado. Quando não for declarado nenhum modificador, presume-se que a variável é do tipo *auto*. Desta forma, este modificador raramente será utilizado.

extern:

Este modificador indica que uma variável ou função é de escopo global e declarada fora do módulo em execução. Esta variáveis são utilizadas em grandes projetos e permitem o compartilhamento destas entre dois ou mais arquivos.

static:

Esta declaração pode modificar tanto variáveis como funções. Ela informa ao compilador que a variável ou função deve ser mantida por toda a duração do programa a partir do ponto de declaração, mesmo quando a execução estiver em outros módulos. Quando o modificador *static* é aplicado a uma variável, o compilador cria armazenamento permanente quase da mesma forma como cria armazenamento para uma variável global. A diferença está no fato de que a variável local static é reconhecida apenas dentro bloco onde foi declarada, mas não perde seu valor entre as chamadas de função.

register:

Uma variável da classe register é armazenada (se possível) nos registradores do processador. Este procedimento é adotado quando queremos melhorar e eficiência e velocidade do programa. Este modificador é reservado para o uso de variáveis de tipo char e int e seus ponteiros.

OPERADORES:

A linguagem de programação C possui um grande número de operadores e, se bem utilizados, permitem recursos poderosos. Abaixo são listados os operadores por precedência.

Operador	Função	Associatividade
( )	membridade	E
[ ]	membridade	E
.	membridade	E
->	membridade	E
-	unário	D
+	unário	D
~	unário	D
!	unário	D
*	unário	D
&	unário	D
++	unário	D
--	unário	D
Sizeof	unário	D
(tipo)	unário	D
*	multiplicativo	E
/	multiplicativo	E
%	multiplicativo	E
+	aditivo	E
-	aditivo	E
<<	bit a bit	E
>>	bit a bit	E
<	relacional	E
>	relacional	E
<=	relacional	E
>=	relacional	E
==	relacional	E
!=	relacional	E
&	bit a bit	E
^	bit a bit	E
|	bit a bit	E
&&	lógico	E
||	lógico	E
?:	condicional	D
=,*=,/=,%=,+=, -=,<<=,>>=, &=,^=,|=	atribuição	D
,	seqüencial	E

Precedência: nada mais é que a hierarquia de avaliação ou a ordem de prioridade de execução.

Associatividade: quando dois ou mais operadores tem a mesma precedência, a associatividade determina a ordem de execução. Há duas regras de associatividade:

1. A cada operador está associada uma direção, e os operadores de um mesmo grupo tem a mesma direção. Esta direção pode ser da esquerda para a direita ou

da direita para a esquerda, dependendo da associatividade dos operadores (veja tabela acima);

2. Os operadores pós-fixados tem maior precedência do que os pré-fixados.

Operadores de membridade:

Parênteses ( ): são usados em chamadas de função.

Colchetes [ ]: envolvem os índices de vetores e matrizes (e algumas vezes ponteiros) para indicar um elemento individual do vetor ou matriz. Também são utilizados para declarar o vetor ou matriz.

Ex.: int nota [5], int notas [5] [5], scanf("%d",&nota[0])

Ponto . : permite o acesso a um "campo" ou membro de uma estrutura.

Ex.: c.nota = nota[0]

Seta ->: (um hífen e um sinal de maior que) é usado para referir-se a um membro de uma estrutura através de um ponteiro para a estrutura.

Ex.: ptr -> nota = nota[1]

Operadores unários:

Os operadores unários são aqueles que exigem apenas um operando e têm alto grau de precedência.

Menos unário - : muda o sinal de seu operando. Operandos sem sinal produzem resultados sem sinal.

Mais unário + : retorna o valor do operando.

Complemento de um ~ : inverte todos os bits da variável afetada. 0´s tornam-se 1´s e 1´s tornam-se 0´s.

Negação unária ! : executa e negação lógica de seu operando. Se o operando for verdadeiro, retorna falso e vice-versa. Ex.: teste = !flag

Conteúdo indireto * : recupera um valor apontado pelo ponteiro ou armazena um valor na variável apontada pelo ponteiro.

Ex.: nome = *p "a variável nome receberá o conteúdo apontado por p

*p = nome "a variável apontada por p receberá o conteúdo da variável nome.

Endereço & : recupera o endereço de uma variável ou função.

Ex.: notaaluno = &nota "a variável notaaluno receberá o endereço da variável nota.

Incremento/Decremento ++/-- : utilizados para incrementar e/ou decrementar em uma unidade o valor de uma variável. Eles podem ser utilizados antes (prefixo) ou após (posfixo) a variável, dando resultados ligeiramente diferentes.

Ex.: Y = X ++ "Y recebe o valor de X e depois X é incrementado

Y = X -- "Y recebe o valor de X e depois X é decrementado

Y = ++ X "X é incrementado e depois Y recebe o valor de X

Y = -- X "X é decrementado e depois Y recebe o valor de X

Obs: No caso dos ponteiros, os operadores de incremento e decremento funcionam de forma diferente. Os ponteiros não são necessariamente incrementados e decrementados em uma unidade, mas pelo tamanho do tipo apontado pelo ponteiro.

Sizeof() : retorna o tamanho de um item.

Ex.: int dado, sizeof(dado)

Operadores aritméticos:

Multiplicação * : multiplica dois operandos.

Divisão / : divide dois operandos.

Módulo % : retorna o resto inteiro da divisão.

Adição + : soma dois operandos.

Subtração - : subtrai um operando de outro.

Operadores Bit a Bit:

Os operadores de bit só podem ser utilizados com valores do tipo caracter ou inteiro.

Deslocamento à esquerda << : é utilizado para deslocar os bits **n** casas à esquerda dentro do byte.

Ex.: a = 15	0	0	0	0	1	1	1	1
a << 1	0	0	0	1	1	1	1	0
a << 4	1	1	1	0	0	0	0	0

Deslocamento à direita >> : é utilizado para deslocar os bits **n** casas à direita dentro do byte.

Ex.: a = 56	0	0	1	1	1	0	0	0
a >>1	0	0	0	1	1	1	0	0
a >>4	0	0	0	0	0	0	0	1

E bit a bit & : retorna bit 1 se ambos comparados forem 1, caso contrário, retorna 0.

Ex.: a = 56	0	0	1	1	1	0	0	0
b = 15	0	0	0	0	1	1	1	1
&	0	0	0	0	1	0	0	0

OU bit a bit | : retorna bit 0 se ambos comparados forem 0, caso contrário, retorna 1.

Ex.: a = 56	0	0	1	1	1	0	0	0
b = 15	0	0	0	0	1	1	1	1
|	0	0	1	1	1	1	1	1

XOU bit a bit ^ : retorna bit 0 se ambos comparados forem iguais, caso contrário, retorna 1.

Ex.: a = 56	0	0	1	1	1	0	0	0
b = 15	0	0	0	0	1	1	1	1
^	0	0	1	1	0	1	1	1

Operadores relacionais:

As expressões compostas com operadores relacionais retornam 0 em caso de expressões falsas e 1 se forem verdadeiras.

Menor que < : compara dois valores e retorna 1 quando o primeiro valor for menor que o segundo, caso contrário retorna 0.

Maior que > : compara dois valores e retorna 1 quando o primeiro valor for maior que o segundo, caso contrário retorna 0.

Menor que ou igual a <= : compara dois valores e retorna 1 quando o primeiro valor for menor que ou igual ao segundo, caso contrário retorna 0.

Maior que ou igual a >= : compara dois valores e retorna 1 quando o primeiro valor for maior que ou igual ao segundo, caso contrário retorna 0.

Igual a == : compara dois valores e retorna 1 quando os dois valores forem iguais, caso contrário retorna 0.

Diferente de != : compara dois valores e retorna 1 quando os dois valores forem diferentes (não iguais), caso contrário retorna 0.

Operadores lógicos:

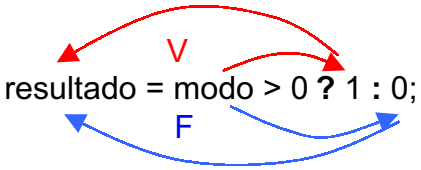
Os operadores lógicos comparam duas expressões e retornam o valor 0 (falso) ou 1 (verdadeiro). O operador E lógico tem precedência sobre o OU lógico.

E lógico && : retorna 1 quando as duas expressões forem 1, caso contrário retorna 0.

OU lógico || : retorna 0 quando as duas expressões forem 0, caso contrário retorna 1.

Operador condicional:

O operador condicional (ou ternário) consiste de um ponto de interrogação (?) e um dois pontos (:) usados juntos e requer três operandos.



"a variável resultado receberá valor 1 se modo for maior que 0, caso contrário receberá 0.

Operadores de atribuição:

Os operadores binários (atuam sobre dois operandos) de atribuição podem ser vistos como operadores "combinados" e são usados para reduzir as expressões. Considerando a expressão: soma = soma + quant, a variável soma esta repetida. O C permite reduzir a expressão utilizando o operador de atribuição mais igual (+=): soma += quant.

= o sinal de igual produz uma simples atribuição. Pi = 3,14159.

***=** o símbolo de multiplicação e o sinal de igual são usados para expressar a multiplicação dos operandos (esquerdo e direito) e o resultado é atribuído ao operando a esquerda deste operador.

/= o símbolo de divisão e o sinal de igual são usados para expressar a divisão dos operandos (esquerdo e direito) e o resultado é atribuído ao operando a esquerda deste operador.

%= o símbolo de percentagem e o sinal de igual são usados para expressar a operação de módulo dos operandos (esquerdo e direito) e o resultado é atribuído ao operando a esquerda deste operador.

+= o símbolo de adição e o sinal de igual são usados para expressar a adição dos operandos (esquerdo e direito) e o resultado é atribuído ao operando a esquerda deste operador.

-= o símbolo de subtração e o sinal de igual são usados para expressar a subtração dos operandos (esquerdo e direito) e o resultado é atribuído ao operando a esquerda deste operador.

<<= o símbolo de deslocamento à esquerda e o sinal de igual são usados para expressar o deslocamento à esquerda do operando da esquerda pela quantidade expressa pelo operando da direita e o resultado é atribuído ao operando a esquerda deste operador.

>>= o símbolo de deslocamento à direita e o sinal de igual são usados para expressar o deslocamento à direita do operando da esquerda pela quantidade expressa pelo operando da direita e o resultado é atribuído ao operando a esquerda deste operador.

&= o operador de atribuição &= produz uma interseção (E) dos operandos da esquerda com o operandos da direita e atribui o resultado ao operando da esquerda.

^= o operador de atribuição ^= produz um OU exclusivo dos operandos da esquerda com o operandos da direita e atribui o resultado ao operando da esquerda.

|= o operador de atribuição |= produz um OU dos operandos da esquerda com o operandos da direita e atribui o resultado ao operando da esquerda.

Operadores de seqüencialização:

O operador de seqüencialização, a virgula (,), indica uma seqüência de instruções executadas da esquerda para a direita.

Ex.: for(i=1,j=1;i<10;i++,j++)

int i,j,c,l;

COMANDOS DE CONTROLE DA LINGUAGEM C

Comandos de seleção:

É utilizado quando o programador quiser subordinar a uma condição, um determinado número de comandos (instruções, funções ou outros comandos). O comando de seleção não necessita da declaração #include, pois é uma palavra reservada.

É utilizada para executar um teste lógico e, então, tomar uma entre duas atitudes possíveis, dependendo do resultado do teste (verdadeiro ou falso).

A linguagem de programação C trata **como verdadeiro, todo valor diferente de zero** e como **falso, todo valor igual a zero**.

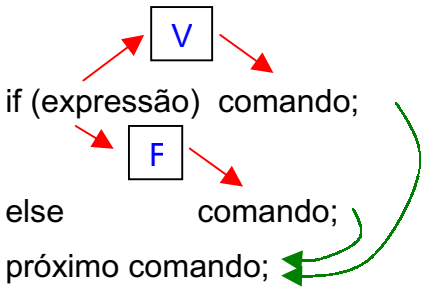
If:

Sintaxe: **if (expressão)** comando;

else comando;

onde, comando poder ser um único, um bloco de comandos e, neste caso, usa-se os delimitadores de bloco de comandos ({ }) ou nenhum comando. A cláusula else é opcional.

Fundamento: se a expressão for verdadeira, será(ão) executado(s) o(s) comando(s) declarado(s) após a expressão. Se a cláusula else for utilizada, ela só será executada se a expressão for falsa.



Exemplos:

```
If (x<0) printf("%f", x);

If (flag)
{
   printf("número da conta: %d", numconta);
   credito=0;
}
```

if aninhados:

Ocorre quando um comando if é um objeto de um outro if ou else.

Sintaxe exemplo:

```
if ( i )
{
   if (  k  ) comando1;
   if (  j  ) comando2;
   else  comando3;
```

```
}
else comando4;
```

if – else – if:

Sintaxe exemplo:

```
if (expressão)  comando;
else    if (expressão)  comando;
        else    if (expressão) comando;
                else    if (expressão)  comando;
                        else  comando;
```

Neste caso, as expressões são avaliadas de cima para baixo. Assim que uma expressão for diferente de zero (verdadeira), o comando associado a ela é executado e a estrutura encerrada. Se nenhuma expressão for verdadeira, será executado o comando associado ao último else e, se ele não for declarado, nenhum comando será executado.

Operador ternário:

O operador ternário é similar ao comando if. É chamado de ternário por que ele necessita de três operandos, como mostra a forma geral:

Sintaxe: **exp1 ? exp2 : exp3**

onde, exp1, exp2 e exp3 são expressões.

Exemplo:

Usando o operador ternário	Usando comando if
x = 10; y = x > 9 ? 50 : 100;	if (x = 10) y = 50; else y = 100;

Os dois exemplos acima fazem a mesma coisa, tem o mesmo efeito, mas o operador ternário é mais eficiente.

Recapitulando: Em C, uma expressão igual a zero é falsa e diferente de zero é verdadeira.

```
if  (b)  printf("%d \n", a/b);
if (b!=0)  printf("%d \n", a/b);
```

switch:

O comando de seleção **switch** é utilizado em caso de seleção múltipla. Ele testa o valor de uma expressão contra uma lista de constantes inteiras ou caracter. Quando o valor é igual, os comandos associados àquela constante são executados.

```
switch(expressão)
{
```

```
        case constante1:
                comandos;
                break;
        case constante2:
                comandos;
                break;
                .
        default:
                comandos;
    }
```

O comando **break** causa a saída imediata do **switch**. Se não existir um comando **break** seguindo os comandos de um **case** , o programa segue executando os comandos dos **cases** abaixo.

Exemplo:

```
void main()
{
    int dia;
    clrscr();
    printf("Digite um dia da semana: ");
    scanf("%d",&dia);
    switch(dia)
    {
        case 1: printf("\nDomingo\n");
                break;
        case 2: printf("\nSegunda\n");
                break;
                .
        case 7: printf("\nSábado\n");
                break;
        default: printf("\nNúmero inválido!!!\n");
    }
}
```

Comandos de repetição:

O comando de repetição é utilizado nos casos em que queremos repetir determinados comandos.

Laço for

Sintaxe: **for(inicialização;condição;incremento)** comando;

for(inicialização;condição;incremento)

```
{
comando;
}
```

onde:

Inicialização: é geralmente uma atribuição que determina o valor inicial da variável de controle. Esta declaração pode ser feita antes do laço e, desta forma, não necessita ser declarada no escopo do comando.

Condição: é uma expressão que determina o final do laço ou o valor final da variável de controle.

Incremento: define como a variável de controle do laço varia cada vez que o laço é repetido.

Exemplo:

```
int x;
for(x=1;x<=100;x++) printf("%d",x);

for(x=100;x!=65;x-=5)
{
        z=x*x;
        printf("o quadrado de %d é %f",x,z);
}
```

O laço infinito:

```
for( ; ; ) printf("este laço é infinito\n");

ch="\0";
for( ; ; )
{
        ch=getchar();        Obs.:
        if (ch=="A") break; ← o comando break causa o término imediato do laço
}
printf("Você digitou A");
```

Laço while

Sintaxe: **while(condição)** comando;

```
while(condição)
{
        comandos;
}
```

A condição pode ser qualquer expressão, e verdadeiro é qualquer valor não-zero. Quando a condição é falsa, o controle do programa passa para o primeiro comando após o bloco de código do comando **while.**

Dentro de um laço, pode ser usado o comando **break.** Ele causa a saída imediata do laço. Já o comando **continue,** força a repetição (retorna ao início do while e testa a condição).

Exemplo:

```
void main()
{
        int t=1, dia;
        while(t)
        {
                clrscr();
                printf("Digite um dia da semana, ou <99> para encerrar: ");
                scanf("%d",&dia);
                switch(dia)
                {
                        case 1: printf("\nDomingo\n");
                                break;
                                .
                        case 7: printf("\nSábado\n");
                                break;
                        case 99: printf("\nEncerrando\n");
                                t=0;
                                break;
                        default: printf("\nNúmero inválido!!!\n");
                } /* fim switch*/
        } /* fim while */
} /* fim main() */
```

Laço do-while

Ao contrário dos laços for e while, que testam as condições no início, o do-while testa a condição no final. Este laço sempre será executado pelo menos uma vez e repete até que a condição se torne falsa.

Sintaxe:

```
do
{
        comando
```

}while(condição);

Exemplo:

```
do
{
        scanf("%d",&num);
}
```

Comandos de desvio:

São utilizados para realizar desvios dentro de um programa.

return:

Este comando é utilizado para retornar de uma função. Quando for invocada a chamada de uma função, o módulo principal do programa transfere a execução para a função (salto do módulo principal para a função) e, ao término da função, o comando return devolve e execução para o módulo principal (salto de retorno). O comando return não precisa ser o último comando declarado dentro da função e, necessariamente não precisa ser apenas um, mas quando ele for executado automaticamente haverá o retorno, devolvendo ou não valor.

Exemplo:

Módulo principal

```
void main()
        {
                char a;
                a=minusculo()[1];
                printf("%c \n", a);
        }
```

Função

```
char minusculo()
        {
                ch=getch();
                if((ch>='A')&&(ch<='Z'))
                        ch+='a'+'A';
                return(ch);
        }
```

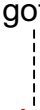
[1] chamada da função minusculo.

goto:

Apesar das linguagens possuírem estruturas de controle eficientes, também é disponibilizado o comando de desvio. Deve-se ter um cuidado muito grande, pois a sua utilização em demasia ou de forma incorreta tende a tornar os programas ilegíveis.

A sua utilização é simples, basta declarar um rótulo (identificador aceito pela linguagem C seguido de dois pontos) e, de algum ponto do programa estabelecer o desvio através da declaração goto rótulo;

Sintaxe: goto rótulo;



rótulo:

Exemplo:

Usando goto

```
x=1;
retorno:
   x++;
   if(x<100)  goto retorno;
```

Usando outra estrutura de controle

```
x=1;
while(x,100)
   x++;
```

```
for(x=1;x,100;x++)
```

break:

O comando break tem duas finalidades. A primeira é de terminar um case num comando switch e a segunda é de forçar o término imediato de um laço de repetição.

Exemplos:

```
for(t=0;t<100;t++)
{
   print("%d \n",t);
   if(t==10)  break;
}
```

```
do
{
   .....
   if(condição1) break;
}while(condição2);
```

exit():

Assim como o comando break força o término de um laço, a função exit() força o término de execução de um programa, retornando ao sistema operacional.

Sintaxe:

void exit(*código_de_erro*);

onde *código_de_erro* é utilizado para indicar o estado de retorno. Quando for zero, indica término normal de execução e outros valores são utilizados para indicar algum tipo de erro (retornado para a variável ERRORLEVEL em arquivos "batch" do DOS).

VETORES E MATRIZES

Vetor do tipo int:

Imagine a seguinte situação: vamos elaborar um programa para ordenar 3 ou 4 valores. Com a declaração de 3 ou 4 variáveis mais alguns comandos e o nosso programa está pronto. Agora, imaginem que devemos ordenar 50 valores. Podemos utilizar os mesmos comandos?

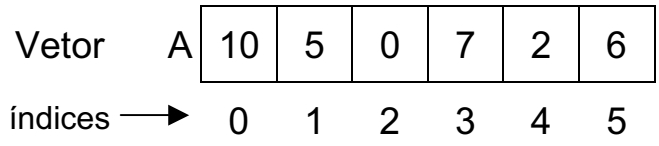
Vamos exemplificar para ficar mais clara esta situação:

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

```
        int a, b, c, x, y, z;
        printf("Digite 3 valores");
        scanf("%d", &a);
        scanf("%d", &b);
        scanf("%d", &c);
        x=a;
        if (b>x)
        {
                x=b;
                y=a;
        }
        if (c>x)
        {
                x=c;
                y=b;
                z=a;
        }
        if (y>z) printf("%d %d %d", z, y, x);
        else printf("%d %d %d", y, z, x);
}
```

Este exemplo ordena 3 valores. Se quiséssemos ordenar 50 valores, esta proposta seria, no mínimo 47 vezes maior, pois cada uma dos 50 valores deve ser lido e comparado com os demais.

Uma forma de resolver este problema é utilizar vetor para armazenar os valores e posteriormente ordená-los. O vetor, nada mais é que um conjunto dividido em n partes, e cada uma destas partes é referenciada por um índice, conforme figura abaixo:



Desta forma, o nome da variável não muda, muda somente o índice que faz referência à posição dentro do conjunto. Como ficaria o programa utilizandoum vetor?

```
#include <stdio.h>
main()
{
        int a[3], x, z, y=1;
        printf("Digite 3 valores");
        for(x=0;x<3;x++)
            scanf("%d", &a[x]);
        while (y==1)
        {
```

```
                y=0;
                for(x=0;x<2;x++)
                    if(a[x]>a[x+1])
                    {
                        z=a[x];
                        a[x]=a[x+1];
                        a[x+1]=z;
                        y=1;
                    }
        }
        for(x=0;x<3;x++)
            printf("%d", a[x]);
}
```

O exemplo declarado acima, faz a ordenação de vetores em ordem crescente e, não interessa qual o tamanho do vetor, no máximo alguns parâmetros serão modificados, mas nenhuma linha seria acrescentada. Este exemplo trata de vetor, ou seja, um conjunto distribuído numa linha com várias colunas.

Vetor do tipo caracter:

```
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

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 funçao 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);
}
```

A linguagem C suporta um grande número de funções que manipulam strings. As mais comuns são:

strcpy(), strcat(), strlen(), strcmp(), sprintf()

As funções de manipulação de strings estão declaradas no arquivo de cabeçalho "string.h".

Função strcpy():

A função strcpy() é usada para copiar o conteúdo da string "de" para a string "para". Forma geral:

```
strcpy(para,de);
```

```
/* exemplo de uso de strcpy() */
#include <stdio.h>
#include<string.h>
main()
```

```
{
  char str [80];
  strcpy(str, "Alo");
  printf("%s", str);
}
```

Função strcat():

A função strcat() anexa (concatena) "s2" em " s1"; "s2" não é modificada. As duas strings devem ser terminadas com nulo e o resultado também será terminado com um nulo.

strcat(s1,s2);

```
#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);
}
```

Função strcmp():

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 numero negativo é retornado.

strcmp(s1,s2);

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>
#include <string.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.

```
/* lê 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;
  }
}
```

Função strlen():

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

strlen(s); onde s é uma string.

O exemplo a seguir, imprime 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));
}
```

Função sprintf():

A função sprintf() permite que se simule um "printf" em uma string. É usada para concatenar valores provenientes de variáveis de diferentes tipos dentro de uma string.

sprintf(s,"formato",<lista de variáveis>); onde s é a string que receberá o "printf"

```
/* exemplo utilizando sprintf*/
#include<stdio.h>
#include<string.h>
main()
{
  char frase[80];
  sprintf(frase, "Novo Hamburgo, %d de %s de %d \n", 6, "agosto", 1999)
  printf("%s", frase)
}
```

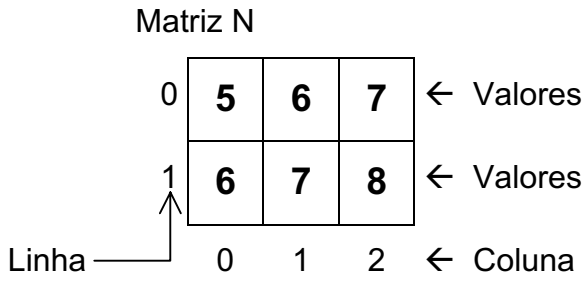
Exercícios:

1. Escrever um programa que lê uma string e a escreve em ordem inversa.
2. Escrever um programa que lê duas strings e informa o tamanho, a igualdade entre elas e no final escrever as strings concatenadas.
3. Escrever um programa que lê uma string s[30] e escreve cada palavra desta string numa nova linha.
4. Escrever um programa que lê uma string e a escreve em maiúsculo. Ver função toupper.
5. Posso escrever o seguinte código?

```
#include <stdio.h>
#include <string.h>
main()
{
 char str[80];
 strcpy(str, "Alo Tom");
 printf(str);
}
```

Matrizes:

Também existe as matrizes, um conjunto de várias linhas e várias colunas, conforme o exemplo abaixo:



Neste caso, esta variável possui dois índices, a linha e a coluna e deve ser declarada da seguinte forma:

```
#include <stdio.h>
main()
{
        int a[3][3], x, y;
        printf("Digite 9 valores");
        for(x=0;x<3;x++)
            for(y=0;y<3;y++)
                scanf("%d", &a[x][y]);
}
```

Exercícios:

1. Faça um programa que ordena uma matriz de 5x5 em ordem decrescente, de maneira que o maior valor se localize na posição [0][0] e o menor na [4][4]. O programa deve escrever a matriz original e a ordenada.
2. Fazer um programa que cria e leia valores para uma matriz 7x7. Escrever a matriz lida, bem como, a soma dos elementos da diagonal principal, a soma dos elementos posicionados acima e abaixo desta.
3. Fazer um programa que cria e leia valores para uma matriz 7x7. Escrever a matriz lida, bem como, a soma dos elementos da diagonal secundária, a soma dos elementos posicionados acima e abaixo desta.

4. Fazer um programa que cria e leia valores para uma matriz 7x7. Escrever a matriz lida, bem como, a soma dos elementos posicionados acima da diagonal principal e secundária, a soma dos elementos posicionados abaixo da diagonal principal e secundária, a soma dos elementos posicionados acima da diagonal principal e abaixo da diagonal secundária e a soma dos elementos posicionados abaixo da diagonal principal e acima da diagonal secundária.

$$a_{7x7} \rightarrow \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} & a_{06} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{60} & a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$$

E/S PELO CONSOLE

FUNÇÃO printf():

A função printf() é utilizada para escrever na tela. O protótipo desta função esta descrito no arquivo de cabeçalho **<stdio.h>.** Desta forma, para utilizar esta função, este arquivo deve ser declarado como **#include <stdio.h>**.

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

A string de controle é dividida em dois tipo. O primeiro tipo são os caracteres que serão escritos na tela e o segundo contém os formatos que definem a maneira pela qual os argumentos serão mostrados.

Código Significado

%d	Exibe um inteiro no formato decimal com sinal
%u	Exibe um inteiro no formato decimal sem sinal
%f	Exibe um tipo float no formato decimal
%c	Exibe um caractere
%s	Exibe uma string
%o	Exibe um número octal sem sinal
%x	Exibe um número hexadecimal sem sinal (letras minúsculas)
%X	Exibe um número hexadecimal sem sinal (letras maiúsculas)

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, 110.789 é ponto flutuante.

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

printf(“%c, %d”,65,”C”);
exibe:
A, 67

FUNÇÃO scanf():

A função scanf() é utilizada para leitura de dados pelo teclado. O protótipo desta função esta descrito no arquivo de cabeçalho **<stdio.h>.** Desta forma, para usar esta função, este arquivo deve ser declarado como **#include <stdio.h>**.

scanf("string de controle", lista de argumentos);

A string de controle determina como os valores são lidos para as variáveis apontadas na lista de argumentos.

Todas as leituras devem ser finalizadas por <ENTER>.

A lista de argumento deve conter os endereços das variáveis a que devem ser atribuídos os valores.

Código	Significado
%d	Lê um inteiro no formato decimal com sinal
%f	Lê um tipo float no formato decimal
%c	Lê um caractere
%s	Lê uma string
%o	Lê um número octal sem sinal
%x	Lê um número hexadecimal sem sinal (letras minúsculas)

Exemplos:

```
#include <stdio.h>
main()
{
        int idade;
        printf(“Digite um número:  ”);
        scanf(“%d”, &idade);
        printf ("\nA minha idade é  %d", idade) ;
}
```

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

Funções getch() e getche()

Função getchar()

Função putchar()

Instruções	Descrição
getchar()	lê um caracter do teclado aguardando <Enter>
getche()	lê um caracter do teclado e prossegue
getch()	lê um caracter sem eco na tela e prossegue
putchar()	escreve um caracter na tela
gets()	lê uma string do teclado
puts()	escreve uma string na tela

ENTRADA E SAÍDA DE DISCO - ARQUIVOS

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 formos trabalhar com mais de um arquivo simultaneamente) 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 para abrir um fluxo e retorna o ponteiro de arquivo associado ao arquivo em questão.

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

onde **modo** é uma string contendo o estado desejado para abertura.

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

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.

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.

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

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

Entretanto, você verá freqüentemente escrito assim:

```
FILE  *ofp;
if((ofp=fopen("test","w"))==NULL)
{
  puts("não 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 é o caractere a ser escrito.

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().

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.

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

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.

Oservação: 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()
{
    FILE *fp;
    char ch;
    if((fp=fopen("teste.txt","wt"))==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
        if(EOF==putc(ch, fp))           buffer até o newline. */
        {
            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 exibe-os na tela.*/

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *fp;
    char ch;
    if((fp=fopen("teste.txt","rt"))==NULL)
     { /* experimentar "rb"/"rt" e testar newline */
        puts("O arquivo não 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);
```

Header: stdio.h

Observação: 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.

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 são 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 não são 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("Não posso abrir arquivo!");
            exit(1);
      }

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

      fclose(out);
}
```

Apagando arquivos: remove()

int remove(char *nome_arquivo);

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

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("Não 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 armazenamentopode ser (e freqüentemente 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) 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("Não 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 */
            }
      }
      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("Não posso abrir arquivo!");
            exit(1);
      }

      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;

}
```

/*escreve uma estrutura no arquivo*/

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    struct
    {
        char titulo[30];
        int regnum;
        double preco;
    } livro;
    char numstr[81];
    FILE *fp;
    if((fp=fopen("livro.dat","wb"))==NULL)
    {
        puts("Não posso abrir arquivo!");
        exit(1);
    }
    do
    {
        printf("\n Digite o titulo:     ");
        gets(livro.titulo);
        printf("\n Digite o registro: ");
        gets(numstr);
        livro.regnum = atoi(numstr);
        printf("\n Digite o preco:    ");
        gets(numstr);
        livro.preco = atof(numstr);
        fwrite(&livro,sizeof(livro),1,fp);
        printf("\n Adiciona outro livro (s/n) ? ");
    } while(getche()== ´s´);
    fclose(fp);
}
```

/*le uma estrutura do arquivo*/

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    struct
    {
        char titulo[30];
        int regnum;
        double preco;
    } livro;
    FILE *fp;
    if((fp=fopen("livro.dat","rb"))==NULL)
    {
        puts("Não posso abrir arquivo!");
        exit(1);
    }
    while(fread(&livro,sizeof(livro),1,fp)==1)
    {
```

```
            printf("\n Titulo:      %s \n", livro.titulo);
            printf("\n Registro: %03d \n",livro.regnum);
            printf("\n Preco:      %.2f \n",livro.preco);
        }
        fclose(fp);
}
```

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	SEEK_SET	0
posição corrente	SEEK_CUR	1
fim do arquivo	SEEK_END	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 test:

```
FILE *fp;
char ch;

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

Os Fluxos-Padrões:

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.

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.

FUNÇÕES GRÁFICAS

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. As funções gráficas estão descritas no arquivo de cabeçalho **<graphics.h>**. Abaixo, apresentamos de forma genérica, todas as funções deste arquivo de cabeçalho.

arc bar bar3d circle

cleardevice	clearviewport	closegraph	detectgraph
drawpoly	ellipse	fillellipse	fillpoly
floodfill	getarccoords	getaspectratio	getbkcolor
getcolor	getdefaultpalette	getdrivername	getfillpattern
getfillsettings	getgraphmode	getimage	getlinesettings
getmaxcolor	getmaxmode	getmaxx	getmaxy
getmodename	getmoderange	getpalette	getpalettesize
getpixel	gettextsettings	getviewsettings	getx
gety	graphdefaults	grapherrormsg	_graphfreemem
_graphgetmem	graphresult	imagesize	initgraph
installuserdriver	installuserfont	line	linerel
lineto	moverel	moveto	outtext
outtextxy	pieslice	putimage	putpixel
rectangle	registerbgidriver	registerbgifont	restorecrtmode
sector	setactivepage	setallpalette	setaspectratio
setbkcolor	setcolor	setfillpattern	setfillstyle
setgraphbufsize	setgraphmode	setlinestyle	setpalette
setrgbpalette	settextjustify	settextstyle	setusercharsize
setviewport	setvisualpage	setwritemode	textheight
textwidth			

Inicializando a Placa de Vídeo:

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);

Onde:
dirver ➔ aponta para um número inteiro na memória correspondente ao driver gráfico que será carregado, conforme tabela abaixo:

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

Obs1: sem um driver gráfico carregado na memória, as funções gráficas não vão funcionar.

Obs2: quando se usa **DETECT**, a função 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.

modo ➔ aponta para o inteiro que especifica um dos modos de vídeo disponível no driver gráfico. Veja tabela na página seguinte.

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.

Obs.: Os drivers gráficos estão contidos nos arquivos *.BGI, que devem estar disponíveis no sistema.

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	EGAMONOHI	3	640x350
HERC	HERCMONOHI	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

Inicializando o modo gráfico sem usar o DETECT:

Para fazer com que o sistema seja inicializado no modo gráfico CGA 4 cores 320x200 pontos de resolução, use:

```
#include<graphics.h>
int driver, modo;
driver=CGA;
modo=CGAC0;
initgraph(&driver,&modo,""); //assume o arquivo CGA.BGI no diretório corrente
```

Neste caso, você esta definindo o tipo de driver e modo, mas se a placa não suportar, poderá ter problemas.

Inicializando o modo gráfico utilizando o DETECT:

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

void main(void)
{
  int driver = DETECT, mode, errorcode;  //declara driver como DETECT
  initgraph(&driver, &mode, "");              //inicializa modo gráfico
  line(0, 0, getmaxx(), getmaxy());// desenha uma linha na tela
  getch();      // suspende a execução do prog. até que seja pressionada uma tecla
  closegraph();        // encerra o modo gráfico
}
```

Quando utilizamos o driver DETECT, é necessário que o driver correspondente à placa de vídeo (normalmente carrega o driver EGAVGA.BGI) esteja disponível no diretório

onde será feita a compilação do programa (veja no ambiente BorlandC → Options, Directories e Output Directories) ou deve ser informado o path, na função initgraph, para que o arquivo seja localizado.

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).

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
                    subseqüentemente chamada */

setbkcolor(cor_de_fundo); /* dá a cor de fundo do traçado de uma
                             função de desenho subseqüentemente chamada */
```

onde cor e cor_de_fundo são dados pelas tabelas abaixo:

Cor	Inteiro Equivalente
EGA_BLACK (preto)	0
EGA_BLUE (azul)	1
EGA_GREEN (verde)	2
EGA_CYAN (ciano)	3
EGA_RED (vermelho)	4
EGA_MAGENTA (margenta)	5
EGA_BROWN (marrom)	20
EGA_LIGHTGREY (cinza claro)	7
EGA_DARKGREY (cinza escuro)	56
EGA_LIGHTBLUE (azul claro)	57
EGA_LIGHTGREEN (verde claro)	58
EGA_LIGHTCYAN (ciano claro)	59
EGA_LIGHTRED (vermelho claro)	60
EGA_LIGHTMAGENTA (marg. Claro)	61
EGA_YELLOW (amarelo)	62

EGA_WHITE (branco)	63

Cor de Fundo	**Inteiro Equivalente**
BLACK (preto)	0
BLUE (azul)	1
GREEN (verde)	2
CYAN (ciano)	3
RED (vermelho)	4
MAGENTA (margenta)	5
BROWN (marrom)	6
LIGHTGREY (cinza claro)	7
DARKGREY (cinza escuro)	8
LIGHTBLUE (azul claro)	9
LIGHTGREEN (verde claro)	10
LIGHTCYAN (ciano claro)	11
LIGHTRED (vermelho claro)	12
LIGHTMAGENTA (margenta claro)	13
YELLOW (amarelo)	14
WHITE (branco)	15

Funções gráficas:

As funções gráficas fundamentais são aquelas que desenham um ponto, uma linha, um círculo e um retângulo. Essas funções são chamadas de putpixel(), line(), circle() e rectangle() respectivamente. Os seus protótipos são mostrados aqui:

```
void far putpixel(int x, int y, int cor);
void far line(int começo_x, int começo_y, int fim_x, int fim_y);
void far circle(int x, int y, int raio);
void far rectangle(int esquerda, int acima, int direita, int abaixo)
```

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 (começo_x, começo_y) até (fim_x, fim_y), na cor corrente estabelecida por setcolor().

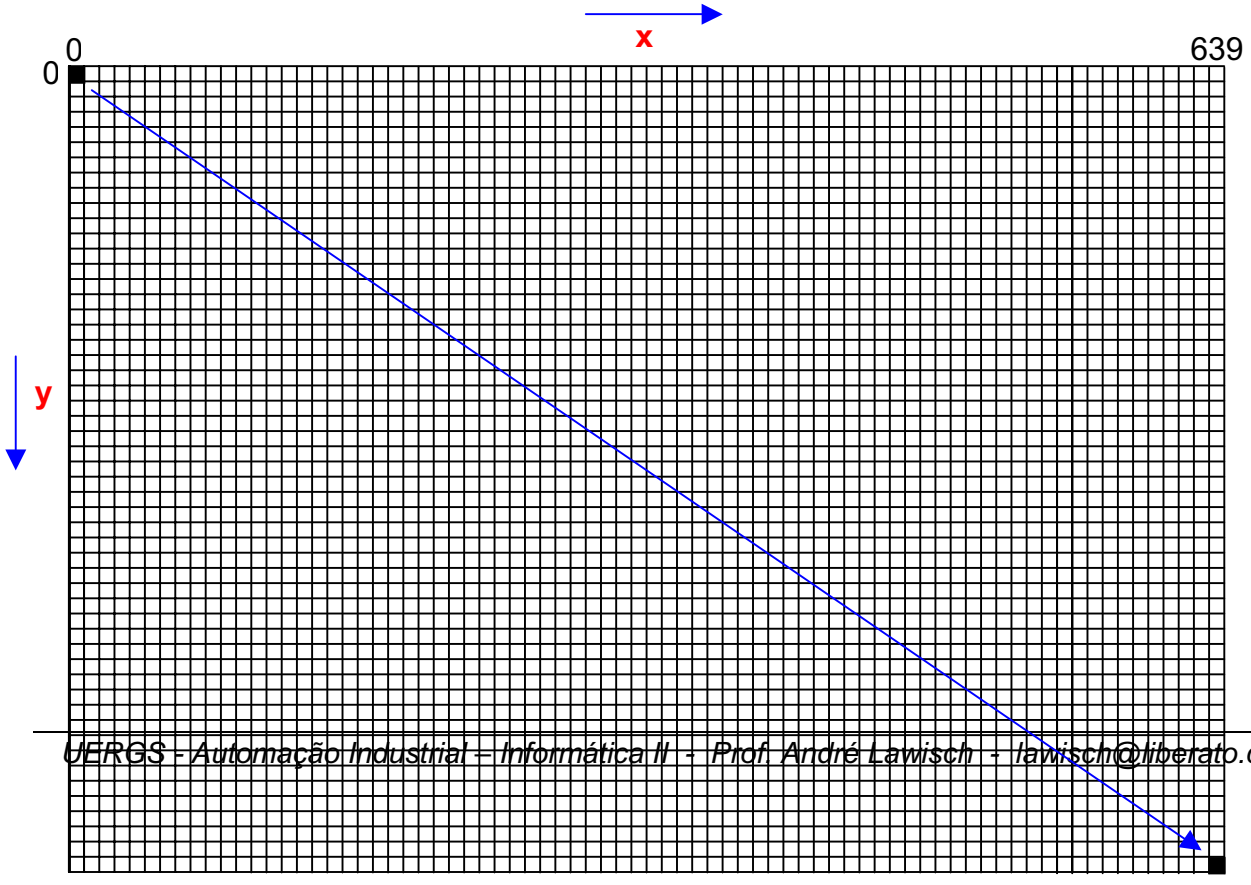
A função circle() desenha um círculo de raio igual a raio, na cor estabelecida por setcolor(), com centro determinado por (x,y).

A função rectangle desenha uma caixa definida pelas coordenadas (esquerda, acima) e (direita, abaixo) na cor de desenho corrente.

A cor de desenho padrão é branco e o fundo preto.

Se qualquer das coordenadas estiver fora dos limites, será exibida somente a parte (se houver) do círculo dentro do limite.

Considerando que foi detectado o driver VGA e modo VGAHI, teremos a resolução de 640x480, conforme exemplo na próxima pagina:



479

639,479

/* Programa exemplo */

```
#include<graphics.h>        //arquivo de cabeçalho gráfico
#include<conio.h>           //arquivo de cabeçalho
main()                      // função principal
{
int driver, modo;           // variáveis locais
register int i;             //
driver=DETECT;
initgraph(&driver, &modo,"");       // inicializa modo gráfico
setcolor(EGA_LIGHTGREEN);   // seta cor do primeiro plano
line(0,0,200,150);      // desenha uma linha das coord. 0,0 a 200,150 na cor setada
setcolor(EGA_LIGHTRED);         // seta cor do primeiro plano
line(50,100,200,125);       // desenha uma linha das coord. 50,100 a 200,150 na cor setada
for(i=0;i<319;i+=10)
        putpixel(i,100,EGA_YELLOW); //Desenha alguns pontos na cor
                                    EGA_YELLOW
setcolor(EGA_LIGHTBLUE);        // seta cor do primeiro plano
circle(50,50,35);           //desenha um círculo com centro em 50,50 e 35 de raio
setcolor(EGA_LIGHTMAGENTA); // seta cor do primeiro plano
circle(100,160,100); //desenha um círculo com centro em 100,160 e 100 de raio
sleep(1);                       // espera 1 segundo
cleardevice();                  // limpa a tela gráfica
setbkcolor(LIGHTBLUE);   //cor de fundo azul claro
setcolor(EGA_LIGHTRED);         // seta cor do primeiro plano
outtextxy(getmaxx()/2,getmaxy()/2,"Isto é um teste");   // imprime a string
                        iniciando no meio da tela
getch();
closegraph();
}
```

Funções associadas ao modo gráfico:

```
void far cleardevice(void);   //limpa a tela gráfica
int far getmaxx(void); //retorna o valor máximo de X na tela gráfica
int far getmaxy(void); //retorna o valor máximo de Y na tela gráfica
int far getx(void); //retorna a posição corrente da coordenada X
int far gety(void); //retorna a posição corrente da coordenada Y
void far moveto(int x, int y); //move a posição corrente (cursor) para X,Y
void far outtextxy(int x, int y, char far *textstring); //mostra a textstring nas
                                                    coordenadas x,y do modo gráfico
void far settextstyle(int font, int direction,); //seta o estilo do texto
onde:
```

int font é um valor referente ao tipo de letra que pode ser utilizado;
DEFAULT_FONT 8x8 bit-mapped font
TRIPLEX_FONT Stroked font
SMALL_FONT Stroked font
SANS_SERIF_FONT Stroked font
GOTHIC_FONT Stroked font

int direction define a direção do texto;
HORIZ_DIR Left to right
VERT_DIR Bottom to top

int charsize define o tamanho do caracter na tela;
1 display chars in 8-by-8 box onscreen
2 display chars in 16-by-16 box onscreen
...
10 display chars in 80-by-80 box onscreen

void far settextjustify(int horiz, int vert); //seta o alinhamento horizontal e vertical do texto, conforme tabela abaixo

Param	Name	Val	How justified
horiz	LEFT_TEXT	(0)	Left <
	CENTER_TEXT	(1)	> center text <
	RIGHT_TEXT	(2)	> right
vert	BOTTOM_TEXT	(0)	from bottom up
	CENTER_TEXT	(1)	center text
	TOP_TEXT	(2)	from top down

void far setlinestyle(int linestyle, unsigned upattern, int thickness);
onde:
int linestyle define o estilo da linha
"SOLID_LINE", linha sólida
"DOTTED_LINE", linha pontilhada
"CENTER_LINE", linha centralizada
"DASHED_LINE", linha não contínua
"USERBIT_LINE" linha com o estilo definido pelo usuário

unsigned upattern define o molde
se, no estilo, for declarado a linha com estilo definido pelo usuário, este parâmetro deverá conter uma seqüência de bit´s 0s e 1s que representará o padrão de pixeis traçados e apagados, usados para traçar a linha.

int thickness define a espessura
"NORM_WIDTH" largura de um pixel
"THICK_WIDTH" largura de três pixeis

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 arc(int x, int y, int stangle, int endangle, int radius); // Esta função é utilizada para desenhar um arco na tela. As coordenadas (x,y) representam as coordenadas do centro da figura, radius é o raio e stangle e endangle especificam os ângulos inicial e final em graus do arco.

void far bar(int left, int top, int right, int bottom); // Esta função desenha uma barra na tela, preenchendo uma área retangular especificada com a cor e o padrão atuais de preenchimento. Os dois primeiros parâmetros declarados no protótipo da função fornecem as coordenadas do canto superior esquerdo da barra e os dois últimos, o canto inferior direito.

void far bar3d(int left, int top, int right, int bottom, int depth, int topflag);
// Esta função desenha uma linha de contorno para uma barra em 3-D, usando a cor e o estilo atuais de linha, preenchendo em seguida a área delimitada com a cor e o padrão atuais de preenchimento. Os dois primeiros parâmetros declarados no protótipo da função fornecem as coordenadas do canto superior esquerdo da barra e os próximos dois, o canto inferior direito, o parâmetro **depth** especifica a profundidade em pixeis e se **topflag** for igual a 1, será desenhado o topo da caixa e 0 para não desenhar.

void far ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);
// Esta função desenha um arco elíptico na tela, usando a cor e o estilo atuais de linha. As coordenadas (x,y) representam as coordenadas do centro da figura, radius é o raio, stangle e endangle especificam os ângulos inicial e final em graus do arco e os parâmetros xradius e yradius fornecem o comprimento do eixos horizontal e vertical.

void far setviewport(int esquerda, int acima, int direita, int abaixo, int clipflag);
// Esta função define a atual porta de visualização na tela gráfica. As coordenadas esquerda, acima, direita e abaixo determinam o retângulo que, na tela, é a porta de visualização. Se o parâmetro **clipflag** for diferente de zero, os desenhos serão aparados no limite da porta de visualização (Isto evita consumo inútil da memória de vídeo)e 0 para o contrário.

void far getviewsettings(struct viewporttype far *info); // Esta função retorna as informações sobre a atual porta gráfica de visualização. A função atribui valores aos campos da estrutura **viewporttype**, cujo endereço é passado como um parâmetro. A estrutura viewporttype é definida em graphics.h como mostrado aqui:

```
struct viewporttype
{
int left, top, right, bottom;
int clipflag;
}
```

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 ao corrente e fazer o ajuste necessário em suas variáveis internas para que a área gráfica caia na tela.

```
/*Demonstraçao de portas de visualizaçao*/
#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 dimensões 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;
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

SCHILDT, Herbert. C, completo e total – 3ª edição revisada e atualizada. São Paulo - Makron Books, 1996.

MIZRAHI, Victorine Viviane. Treinamento em linguagem C – módulo 1 e 2. São Paulo - McGraw-Hill, 1990.

GOTTFRIED, Byron S. Programando em C. – São Paulo - McGraw-Hill, 1993.

CASTRO, Fernando César Comparsi de. Apostila: Curso de linguagem C aplicada à Engenharia.

PRATES, Rubens. Guia de consulta rápida – Linguagem C. Novatec Editora.

PLANTZ, Alan C. C:quick reference. Rio de Janeiro – Campus, 1989. Tradução de Fernando Cabral.

MANZANO, José Augusto N. Garcia. Estudo dirigido de Linguagem C. São Paulo – Érica – 1997.

KELLY-BOOTLE, Stan. Dominando o Turbo C. Rio de Janeiro – Editora Ciência Moderna – 1989. Tradução Eduardo Alberto Barbosa.

MENDONÇA, Alexandre e ZELENOVSKY, Ricardo. PC: Um Guia Prático de Hardware e Interfaceamento. Rio de Janeiro – MZ Editora Ltda – 1999.

ANEXO 1

Programa exemplo – Utilização do Mouse

```
#include <stdio.h>
#include <dos.h>
#include <graphics.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
```

```
/* Estruturas extraídas do arquivo de cabeçalho dos.h

struct WORDREGS {
   unsigned int    ax, bx, cx, dx, si, di, cflag, flags;
};

struct BYTEREGS {
   unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
};
```

```
                union   REGS    {
                   struct  WORDREGS x;
                   struct  BYTEREGS h;
                };

                int int86(int intno,
                        union REGS *inregs,
                        union REGS *outregs);
                */

//-------------------------------------------------------------------------------
union REGS inregs;
int cx, bx, ax, dx;
//-------------------------------------------------------------------------------
void inicio_mouse()
{
   int iret;
   inregs.x.ax=0x0000;
   iret=int86(0x33,&inregs,&inregs);
   if(iret==0)
   {
        printf("Driver de mouse nÆo encontrado na mem¢ria.\n");
        exit(3);
   }
}



//-------------------------------------------------------------------------------
void aparece_mouse()
{
   inregs.x.ax=0x0001;
   int86(0x33,&inregs,&inregs);
}


//-------------------------------------------------------------------------------
void apaga_mouse()
{
   inregs.x.ax=0x0002;
   int86(0x33,&inregs,&inregs);
}

//-------------------------------------------------------------------------------
void le_mouse()
{
   inregs.x.ax=0x0003;
   int86(0x33,&inregs,&inregs);
```

```
    bx   =  inregs.x.bx;
    cx   =  inregs.x.cx;
    dx   =  inregs.x.dx;
}

//---------------------------------------------------------------------------
int modo_grafico()
{
   int gdriver = DETECT, gmode, errorcode;
   initgraph(&gdriver, &gmode, "");
   errorcode = graphresult();
   if (errorcode != grOk)
   {
      printf("Graphics error: %s\n", grapherrormsg(errorcode));
      printf("Pressione qualquer para encerrar:");
      getch();
      return(1);
   }
}
//---------------------------------------------------------------------------
void moldura()
{
   int loop;
   for(loop=1;loop<16;loop++)
   {
      setcolor(loop);
      rectangle(loop, loop, getmaxx()-loop, getmaxy()-loop);
   }
}
//---------------------------------------------------------------------------
void escreve(int b, int c, int d)
{
       gotoxy(18,10);
       printf("o botão pressionado   --> variável bx=%3d", b);
       gotoxy(18,12);
       printf("coordenada horizontal --> variável cx=%3d", c);
       gotoxy(18,14);
       printf("coordenada vertical   --> variável dx=%3d", d);
}

//---------------------------------------------------------------------------
void botao(int x,int y,int alt, int comp)
{
          setcolor(63);
          line(x,y,x,y-alt);
          line(x,y-alt,x+comp,y-alt);
          setcolor(07);
          line(x+comp,y-alt,x+comp,y);
          line(x+comp,y,x,y);
```

```
}
//--------------------------------------------------------------------------------
void botao_inv(int x,int y,int alt, int comp)
{
        setcolor(07);
        line(x,y,x,y-alt);
        line(x,y-alt,x+comp,y-alt);
        setcolor(63);
        line(x+comp,y-alt,x+comp,y);
        line(x+comp,y,x,y);
}
//--------------------------------------------------------------------------------
void testa(int bot, int x1, int y1, int alt1, int comp1)
{
        if((bx==1)&&(cx>=x1)&&(cx<=x1+comp1)&&(dx>=y1-alt1)&&(dx<=y1))
        {
                apaga_mouse();
                botao_inv(x1,y1,alt1,comp1);
                aparece_mouse();
        }
        if((bx==2)&&(cx>=x1)&&(cx<=x1+comp1)&&(dx>=y1-alt1)&&(dx<=y1))
        {
                apaga_mouse();
                botao(x1,y1,alt1,comp1);
                aparece_mouse();
        }
}


//================================================
void main(void)
{
  int comprimento=140, altura=20;
  if(modo_grafico()) exit (1);
  setbkcolor(8);
  moldura();
  botao(250,320,altura,comprimento);
  inicio_mouse();
  aparece_mouse();
  while(1)
  {
          le_mouse();
          escreve(bx, cx, dx);
          if (bx==3)
          {
          apaga_mouse();
          break;
          }
          testa(bx,cx,dx,altura,comprimento);
```

```
   }
   getch();
   closegraph();
}
//=============================================
```

ANEXO 2

A porta paralela

A porta paralela , também chamada de porta da impressora, esta projetada para permitir a conexão do PC com impressoras paralelas, mas também pode ser utilizada como porta genérica, constituída de 3 registradores, entrada/saída. Esta porta, também chamada de LPT1, é encontrada no endereço base 378h, normalmente. É possível que um computador possa ter duas portas paralelas configuradas. Esta segunda, chamada de LPT2, possui endereço base 278h.

Existe uma placa com adaptador de vídeo monocromático e porta paralela, chamada de MD&PA com endereço base de 3BCh. Apesar de ser antiga, seu endereço ficou consagrado e ainda é utilizado em alguns computadores. Sendo assim, é recomendado que antes de utilizar o computador, verifique o endereço da porta na tela de inicialização ou pelo painel de controle, sistema, ..., do windows. Veja no quadro abaixo, os diversos endereços de I/O das portas paralelas.

LPT1	LPT2	MD&PA	Nome
378h	278h	3BCh	Dados
379h	279h	3BDh	Estado
37Ah	27Ah	3BEh	Controle

Descrição da Porta Paralela SPP

Sinal	Nome	Pinagem 25p/36p*	E/S	Descrição
/STROBE	Strobe	01/01	S	Indica se os dados estão prontos ou não para serem transmitidos. (0 = Dados prontos para serem transmitidos e 1 = Dados não prontos para serem transmitidos)
/ACK	Acknowledge	10/10	E	Indica que a impressora está preparada para receber dados.
BUSY	Busy	11/11	E	Indica que a impressora não está preparada para receber dados.
PE	Paper Empty	12/12	E	Indica que a impressora está sem papel para a impressão.
SELECT	Select	13/13	E	Indica que a impressora está no estado "on line", pronta para receber informação.
/AUTO FD XT	Auto Feed	14/14	S	O papel avança para o começo da próxima linha.
/ERROR	Error	15/32	E	Indica quando ocorre algum tipo de erro (término do papel, impressora desativada).
/INIT	Init	16/31	S	Reinicializa a impressora e limpa o buffer de impressão.
/SELECT INPUT	Select Input	17/36	S	Os dados só podem ser transferidos para a impressora quando esta linha estiver em nível lógico baixo.
D0 a D7	D0 a D7	2 a 9/ 2 a 9	S	Dados.
GND	Ground	18 a 25/ 19 a 30	S	Terra.

***** O primeiro número refere-se ao pino do conector de 25 pinos (DB-25). O segundo número refere-se ao pino do conector de 36 pinos do cabo da impressora (Centronics 36 pinos).

Endereçamento da porta paralela SPP

Endereço base - 378h (Registrador de dados)

Por onde os dados sairão. Colocando-se um número de 8 bits nesse endereço fará com que o dado seja exteriorizado pela porta paralela.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Dado	Data 7	Data 6	Data 5	Data 4	Data 3	Data 2	Data 1	Data 0
Pino	9	8	7	6	5	4	3	2

Estes pinos deverão ser capazes de fornecer/drenar 2,6/24mA. Uma instrução OUT escreve diretamente nos pinos do conector. Quando o bit for 1, resulta em nível TTL alto na saída.

Endereço de status - 379h (Registrador de estado)

Serve para ler o estado das linhas de entrada, como Acknowledge e Busy. Colocando-se uma instrução de leitura de dados para ler esse endereço retornaremos o estado do registrador de status, um número de 8 bits que é dividido da seguinte forma:

	*bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Dado	*BUSY	ACK	PE	SELECT	ERROR	X	X	X
Pino	11	10	12	13	15	-	-	-

* Bit com inversão.

Endereço de controle - 37Ah (Registrador de controle)

Serve para habilitarmos as linhas de saída de controle, como Init e Auto Feed. Para isto, basta escrevermos um número de 8 bits neste endereço, no formato apresentado a seguir. Além disso, através do bit 4, mascaramos a IRQ 7. Com este bit em "1", a interrupção pode ocorrer.

	bit 7	bit 6	bit 5	bit 4	*bit 3	bit 2	*bit 1	*bit 0
Dado	X	X	X	IRQ 7	*SELECT INPUT	INIT	*AUTO FD XT	*STROBE
Pino	-	-	-	-	17	16	14	1

* Bits com inversão.