INSTITUTO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE SANTA CATARINA
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
DISCIPLINA: INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO C

INTRODUÇÃO À

LINGUAGEM C

PLANO DE AULA

3ª. EDIÇÃO

PROFA. FERNANDA ISABEL MARQUES ARGOUD

FLORIANÓPOLIS, ABRIL DE 2009.

SUMÁRIO

1 - INTRODUÇÃO

Uma das linguagens mais utilizadas por técnicos e pesquisadores é a linguagem C. Isto ocorre principalmente pela versatilidade e pela complexidade da linguagem, que permitem a criação de programas muito sofisticados.

1.1 - HISTÓRICO

A primeira versão da linguagem foi desenvolvida por dois pesquisadores da Bell Laboratories, Brian Kernighan e Dennis Ritchie. A empresa necessitava de uma linguagem especificamente para escrever o sistema operacional UNIX, mas C revelou-se tão eficiente e "transportável" para outros sistemas operacionais, sistemas e hardwares que seu uso alastrou-se rapidamente. Esta primeira versão, chamada "K&R" sofreu algumas modificações com o tempo, para adaptar-se a computadores com mais de 8 bits e assim nasceu a versão "ANSI C", considerada um padrão da linguagem. Algum tempo depois, com a moda de programação orientada a objetos, nasceu a versão C++ que não mais segue a programação linear. Várias empresas criaram seus próprios compiladores C e assim apareceram o MS C (Microsoft), o Turbo C, Borland C, etc.

1.2 - CARACTERÍSTICAS DA LINGUAGEM

"C é uma linguagem compilada, estruturada e de baixo nível."

Linguagem **compilada** porque, após ser escrita num editor de textos qualquer (que siga o padrão ASCII), precisa ser decodificada, compilada (cada módulo separadamente) e **linkada** para obter-se um programa executável. Certos softwares como o Turbo C e o Borland C permitem que se edite, compile e linke os programas em C dentro de um mesmo ambiente (chamado de "IDE", ou "*Integrated Development Environment*"), o que facilita muito a manipulação.

É uma linguagem **estruturada** porque segue o padrão de endentação, tal como em Pascal e Fortran por exemplo, com alinhamentos dos blocos lógicos cada vez mais à direita, quanto mais "interno" ao bloco for o comando, e com execução linear, sem utilização de *goto*'s, *break*'s, etc.

Finalmente é uma linguagem de **baixo nível** por permitir acesso às camadas lógicas mais baixas da máquina. Isto é, por aproximar-se bastante da linguagem de máquina, Assembler, que apesar de bastante rudimentar tem a capacidade de acessar diretamente a memória, o hardware do computador, como registradores, portas, posições da RAM, etc. Com isto, ganha-se muito em rapidez de execução e em poder para utilizar completamente os recursos do computador. É importante salientar que apesar de ser possível utilizar-se funções muito complexas de baixo nível em C, um programador não interessado nisto terá uma linguagem estruturada como qualquer outra de alto nível.

Outra característica importante é que C é uma linguagem de "estilo livre", sem formatação rigorosa como Fortran e Basic. Em Basic, cada linha contém um comando e cada comando ocupa somente uma linha (às vezes, há até numeração das linhas). Em Fortran, os arquivos de saída contém espaços reservados para cada string, valor de caractere ou espaço em branco que deverá ser impresso. Nada disto ocorre em C. Desde que a sintaxe correta seja seguida, não há maiores restrições na linguagem. Ou melhor, quase não há.

☝ O programador não pode esquecer que o compilador C diferencia caracteres minúsculos de maiúsculos. Por exemplo, as variáveis "numero" e "Numero" são consideradas diferentes uma da outra na linguagem C.

Um avanço significativo que C possibilitou, foi escrever-se um programa numa linguagem de alto nível (que C não deixa de ser) e ter, após a compilação, um código gerado diretamente em Assembler. Qual a vantagem nisto? Quem trabalha com circuitos contendo microprocessadores ou microcontroladores sabe! Até pouco tempo seria necessário escrever páginas de código em Assembler para funções simples. Agora, com o uso de compiladores como o Keil - C ou Avocet, o programador escreve o código em C e o compilador encarrega-se de transformá-lo em Assembler.

Costuma-se dizer que o C é uma linguagem extremamente "portável". Ou seja, foi desenvolvido para UNIX, mas roda muito bem em DOS. Além disto, um programa escrito em C, para uma estação de trabalho provavelmente rodará num PC ou num computador médio; ou mesmo passará de um IBM-PC para um Machintosh. Isto ocorre porque C não é rígido, não tem funções pré-definidas de I/O (aliás, de nenhum tipo) para cada máquina e adapta-se a qualquer hardware. Funções específicas (como entrada e saída) para cada máquina devem ser escritas pelo próprio usuário e certamente estas não rodariam num hardware diferente. Uma opção para o programador mais prático (ou preguiçoso) é procurar nas dezenas de arquivos de biblioteca ".h" (por exemplo, "stdio.h") uma função que se encaixe nas suas necessidades. É importante salientar que também estas funções (como por exemplo, printf(), que imprime saídas formatadas na tela) foram desenvolvidas por usuários e não pertencem à linguagem original. Na verdade, quase tudo em C é definido pelo usuário, daí sua complexidade e ao mesmo tempo, seu poder.

2 - PROGRAMA-EXEMPLO

O programa abaixo serve para ilustrar a estrutura de um programa em C.

```
/*****************************************************************/
/*** PROGRAMA C que demonstra como usar várias primitivas gráficas   ***/
/***    com monitor VGA                                            ***/
/*****************************************************************/
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <conio.h>

#define TRUE 1
#define MAX 200
#define NAO  'n'
#define SIM    's'

void circulo(int, int);
void barra(int);
void elipse(int, int) ;

void main(void)
{
      int gdriver = DETECT, gmode, errorcode, i;
      int midx, midy;
      char ch;

     initgraph(&gdriver, &gmode, " " );
     errorcode = graphresult();
     if (errorcode != grOk)
     {        printf("Erro de Função Gráfica:  %s \n", grapherrormsg (errorcode));
              printf("Aperte uma tecla para parar: ");
              getch();
              exit(1);
     }
    setbkcolor(BLACK);
    for(i = 10; ch != NAO; i +=100 )
    {
        circulo(i, i);                          // desenha um círculo vazio
        barra(i + 50);                          // desenha uma barra cheia
```

Diretivas do pré-processador: inclusão de bibliotecas

Diretivas do pré-processador: definição de constantes

Protótipos de funções

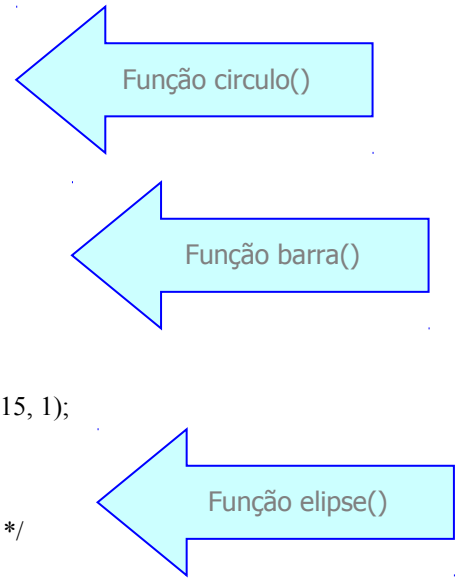
Função Principal

```
            elipse(i +200, i + 300);                    // desenha uma elipse cheia
            printf("Deseja continuar? (s/n)");
            ch = getchar();
        }
        closegraph();
}

 void circulo(int x, int y)
  /* desenha um pequeno círculo  */
 {  setcolor(BLUE);
      circle(x, y,40);
  }

void barra(int posicao)
/* desenha uma barra tridimensional */
{   setcolor(CYAN);
    setfillstyle(LINE_FILL, CYAN);
    bar3d(posicao, 20, posicao + 50, 90, 15, 1);
}

void elipse(int x, int y)
/*  desenha uma elipse, traça e preenche */
{   setcolor(MAGENTA);
    setfillstyle(SLASH_FILL, MAGENTA);
    fillellipse(x, y, 40, 70);
}
```



Observe atentamente o programa. Este inicia com comentários sobre o nome do programa e o que ele faz. Em C, comentários sempre estão entre os símbolos "/*" e "*/", ou após o símbolo "//". Ou seja, tudo que estiver entre "/* ... */" é completamente ignorado pelo compilador e tudo que estiver, na mesma linha, após o "//" também o é.

Em seguida, temos uma série de comandos, chamados "diretivas", que não pertencem à linguagem C propriamente, sempre começando com o símbolo "#". Estes comandos fazem parte do que chamamos de "Pré-Processador". O pré-processador, como o nome diz, é um programa que examina o programa-fonte em C antes deste ser compilado e executa certas modificações nele, baseado nas diretivas. As diretivas mais comuns são "#include" e "#define". A primeira, indica ao compilador quais arquivos de *header* (os "*.h") serão utilizados pelo programa. A segunda, define constantes e macros para facilitar a visualização do programa. Para maiores informações, vide Apêndice C.

A área seguinte é a região de declaração dos "protótipos de funções". Isto é necessário, em alguns compiladores C, para indicar ao compilador quais e qual o formato das funções que existem no programa. Por exemplo, o protótipo "void circulo(int, int)" diz ao compilador que dentro deste código ele encontrará uma função chamada *circulo*, que recebe dois argumentos do tipo *int* (inteiros) e não retorna valor algum (*void*) à expressão chamadora.

As outras áreas são todas funções. A primeira é a função principal do programa, *main*(). A função ***main*() é sempre a primeira a ser executada num programa C**, não importa onde esteja localizada no código. Neste programa foi colocada em primeiro lugar por convenção. Note que uma função inicia-se sempre com o nome desta (seu tipo e argumentos) e em seguida temos o seu "corpo", sempre delimitado pelos caracteres "{" e "}". Tudo que estiver entre os símbolos de abre e fecha-chaves faz parte do corpo de uma função. Com exceção da função *main*(), que existe obrigariamente em qualquer programa C, todas as outras funções foram previamente declaradas em protótipos.

Não só as funções, mas também blocos de comandos são delimitados por "{" e "}". Note o corpo do comando "*for*" do programa.

As variáveis em C geralmente são declaradas no início dos blocos, em alguns compiladores por convenção e em outros por obrigação. Contudo, a rigor, as variáveis podem ser declaradas em qualquer ponto do programa (dentro do escopo necessário, claro) desde que antes de serem utilizadas.

Finalizando, note que a maioria dos comandos C terminam com o caractere ";" que é um análogo do "End" utilizado em outras linguagens, como Fortran e Pascal.

Nos capítulos seguintes todos os pontos discutidos acima serão explorados e o Apêndice C tem uma lista das diretivas mais comumente utilizadas.

3 - TIPOS DE VARIÁVEIS E CONSTANTES

3.1 - CONSTANTES

Uma constante tem valor fixo e inalterável. Em C uma constante caractere é escrita entre aspas simples (“ ' “ e “ ' “); uma cadeia de caracteres, entre aspas duplas (“ " “ e “ " “); e constantes numéricas como o número propriamente dito.

Exemplos de declarações de constantes:

```
const char const_caract ='c';
#define NOME "meu primeiro programa  "
#define VALOR 8
```

Constantes podem ser dos tipos:

- **Constantes numéricas:** inteiros, octais, hexa, longas e ponto flutuante (reais).
  Ex:

```
45E-8 (exponencial)      32 (inteiro)          034 (octal)              0xFE(hexa)
2e3 (exponencial)32L  (longa)          567893 (longa implícito)     2.3  (double)
```

Como podemos notar, constantes inteiras não possuem ponto decimal; constantes octais devem ser precedidas por um '0'; constantes hexa, por um '0x'; constantes longas devem ser seguidas por um 'L', mas quando se trata de um número muito grande o compilador já entende que é um inteiro longo; e constantes *double* ou *float* têm ponto decimal flutuante.

- **Constantes de Caractere**: podem ser representadas por seu código ASCII, ou entre aspas, ' '.
Ex:

```
 ' A ' = 65    (em ASCII)
```

- **Sequências de Escape** - São códigos de tarefas a serem executadas na compilação, representados por caracteres que não podem ser impressos.

Sequência de escape	**Significado**
\a	Caractere Bell (ANSI C)
\b	Caractere de retrocesso (backspace)
\n	Caractere de nova linha
\r	Caractere de retorno de carro
\t	Caractere de tabulação horizontal
\\	Caractere de contra-barra
\'	Caractere de aspas simples
\"	Caractere de aspas duplas
\?	Caractere de ponto-de-interrogação
\###	Código ASCII em octal de caractere
\x###	Código ASCII em hexadecimal de caractere
\0	Caractere nulo

Obs: os símbolos “**#**” correspondem a dígitos de 0 a 7, p/ a base octal e de 0 a F, p/ a base hexadecimal!

Ex: A instrução
printf(“\n\t\t Hoje \202 dia da Can\x87\xc60 da \nAm\x82rica\?”);

vai imprimir:

```
        Hoje é dia da Canção da
América?
```

Ex2: Teste o programinha:

```
#include  <stdio.h>
void main(void)
{
    printf("A\nB\nC");
    printf("\n");
    printf("A\tB\tC");
    printf("\n");
    printf("AB\rC");
    printf("\n");
    printf("AB\b\bC");
    printf("\n");
    printf("Beep\aBeep\aBeep\a");
    printf("\n");
    printf("\A\B\C");                    /* O que acontece aqui ????  */
    printf("\n");
    printf(" Os comandos do Dos estão no C:\\DOS ");
    printf("\n");
    printf("Can\x87\xc6o da Am\x82rica – Milton Nascimento\n");
    printf("\xc9\xcd\xcd\xcd\xbb\n\xc8\xcd\xcd\xcd\xbc");
    printf("\n");
    printf("\"Cuidado!\" não fume ");
}
```

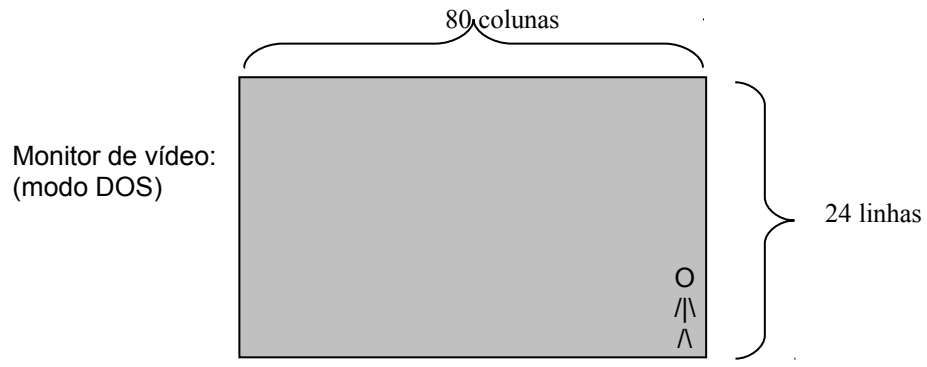
- **Strings** - Conjunto, série ou sequência de caracteres seguido do caractere ' \0 ' e delimitado por " ".

Ex:

```
char * texto= "ABC";       // ou char  texto[4]  ={ 'A' , 'B' , 'C' , '\0'};
```

Exercícios:

Usando os caracteres do teclado, as sequências de escape, a função printf() e a função gotoxy(*int x*, *int y*), que posiciona o cursor na coluna *x* e na linha *y* do monitor, escreva o programa que imprime um boneco no canto inferior direito da tela de saída. Ex:



3.2 - TIPOS DE VARIÁVEIS

As variáveis são o aspecto fundamental de qualquer linguagem de computador. Uma variável em C é um espaço de memória reservado para armazenar um certo tipo de dado e tendo um nome para referenciar seu conteúdo. Ao contrário das constantes, uma variável tem seu valor mutável, daí o nome "variável".

Declaração de uma variável: ocorre quando a variável é "apresentada" ao compilador. O usuário declara que vai criar uma variável de um certo tipo para utilizá-la em seu programa. A sintaxe da declaração de uma variável é:

```
tipo_var      nome_var  ;
```

onde *tipo_var* é o tipo de variável criada e *nome_var*, o nome ou os nomes (separados por vírgulas) das próprias.

Ex: int num ;

e assim está declarada a variável "num" inteira.

Ex2: float var1, var 2, var 3;

declara as variáveis "var1" , "var2" e "var3", ao mesmo tempo, como sendo do tipo float.

Definição de uma variável: ocorre quando a variável já declarada recebe um valor, uma atribuição. A definição da variável pode ocorrer na mesma linha da declaração, mas sempre depois desta e denominamos isto de "inicialização da variável". A sintaxe da definição de variáveis é:

```
nome_var = valor ;
```

onde *nome_var* é o nome (ou nomes, separados por símbolos de igual) da variável e *valor* é o valor atribuído à mesma.

Ex: num = 5; ou num1 = num 2 = num3 = 0;

e assim o valor 5 (inteiro) é atribuído à variável "num" e o mesmo valor, 0, é atribuído a três variáveis ao mesmo tempo.

Ex2: char x = 'b';

neste caso a inicialização da variável "x" como tendo o valor do caractere 'b' ocorreu logo após a declaração.

Tipos de Variáveis: O tipo de uma variável informa a quantidade de memória, em bytes que esta irá ocupar. São eles:

Tipo	Tamanho	Escala (para *word* de 8 bits, no Turbo C)
unsigned **char**	1 word	0 a 255
char	1 word	-128 a 127
enum *	2 words	-32.768 a 32.767
unsigned **int**	2 words	0 a 65.535
short **int**	2 words	-32.768 a 32.767
int	2 words	-32.768 a 32.767
unsigned long	4 words	0 a 4.294.967.295
long	4 words	-2.147.483.648 a 2.147.483.647
float	4 words	$3{,}4 \cdot 10^{-38}$ a $3{,}4 \cdot 10^{38}$
double	8 words	$1{,}7 \cdot 10^{-308}$ a $1{,}7 \cdot 10^{308}$
long **double**	10 words	$3{,}4 \cdot 10^{-4932}$ a $1{,}1 \cdot 10^{+4932}$
void	0	sem valor
ponteiro	1-2 words	endereço de memória

Obs: indica ao compilador que nenhuma memória deve ser alocada

Os tipos básicos estão em negrito, os outros tipos são chamados de **modificadores de tipos** e servem para alterar o tamanho de um tipo básico. Por exemplo, em alguns computadores, como o IBM-370, o modificador "short" faz com que o tipo "int" fique com a metade do tamanho, 8 bits. O tamanho dos tipos varia bastante de máquina para máquina e de compilador, para compilador.

Inteiros com e sem sinal são interpretados de maneira diferente pelo compilador. O bit de ordem superior, bit 15, de um número inteiro com sinal é sempre '0', quando o inteiro é positivo e '1' quando o número é negativo. Se usarmos o modificador "unsigned" o compilador vai ignorar o bit de sinal, tratando-o como um bit a mais para números positivos.

Ex:

```
void main(void)
{
   unsigned int j = 65000;
   int i = j ;
   printf(" %d  % u \n", i, j);
}
```

O resultado será (na base binária: 1111.1101.1110.1000):

```
-536    65000
```

Variáveis também são modificadas por **Classes de Armazenamento**: **auto**, **static**, **register** e **extern**. Isto será visto mais tarde, quando estudarmos o escopo das variáveis.

☝* *O tipo "enum" é um acréscimo recente ao C. É definido como um conjunto de constantes enumeradas. Cada constante é associada a um valor inteiro.*

Ex: *enum tipo_sinaltransito { vermelho, amarelo, verde};*
enum tipo_sinaltransito sinal;

, pelo qual sinal *só pode ter um dos três valores:* vermelho*, que tem índice 0,* amarelo*, de índice 1 e* verde*, 2.*

Exercícios:

1 - Identifique o tipo das seguintes constantes:
a)'\r' _____ b) 2130 _____ c) -123 _____ d) 33.28_____ e) 0x42 _____
f) 0101 _____ g) 2.0e30_____h) '\xDC'_____i) '\"'_____ j) '\\' _____
k) 'F' _____ l) 0 _____ m) '\0'_____

2 - O que é uma variável, na linguagem C?

3 - Quais os 5 tipos básicos de variáveis em C?

4 - O tipo *float* ocupa o mesmo espaço que _____ variáveis do tipo *char*.

5 - Escreva um programa que contenha uma única instrução e imprima na tela:

```
Esta é a linha um.
    Esta é a linha dois.
        um
            dois
                tres
```

Obs: para escrever a letra ‘é’ utilize o código \202, da tabela ASCII extendida.

4 - OPERADORES

A linguagem C é rica em operadores, em torno de 40. Alguns são mais usados que outros, como é o caso dos operadores aritméticos que executam operações aritméticas.

Os tipos de operadores são: Aritméticos, Relacionais, Lógicos, Bit a Bit, de Atribuição, de Atribuição Reduzida, Pré e Pós-Fixados e Condicionais.

Operadores Aritméticos - Representam as operações aritméticas básicas de soma, subtração, divisão e multiplicação; além dos operadores unários (operam apenas sobre um operando) de sinal negativo e positivo. São eles:

Binários	
+	Soma
-	Subtração
*****	Multiplicação
/	Divisão
%	Módulo (devolve o resto da divisão inteira)
Unário	
-	Menos unário
+	Mais unário

Ex:

Expressão	Tem o valor
5 + i	5 somado ao valor da variável i
22.3 * f	22,3 multiplicado pelo valor da variável f
k/3	o valor de k dividido por 3 *
x-y	o valor de x menos o valor de y
22/3	7 (como é divisão de inteiros o resultado é truncado)
22%3	1 (este operador devolve o resto da divisão inteira)
-a	-1 multiplicado ao valor da variável a

Ex2:

```
#include <stdio.h>
void main(void)
{
   int ftemp, ctemp;
   printf("Digite a temperatura em graus Celsius: ");
   scanf("%d", &ctemp);
   ftemp = 9./5 * ctemp   + 32;  // porque este ponto aqui???
   printf("Temperatura em graus Fahrenheit é  %d", ftemp);
}
```

Resultado:

```
Digite a temperatura em graus Celsius: 21
Temperatura em graus Fahrenheit é 69
```

Operadores Relacionais - São assim chamados porque são utilizados para comparar, relacionar dois operandos. São eles:

>	Maior
>=	Maior ou Igual
<	Menor
<=	Menor ou Igual
==	Igualdade

!=	Diferença

O resultado da comparação será sempre igual a 0 (Falso) ou 1 (Verdadeiro).

Ex:

Expressão	Tem o valor
5< 3	0
3 < 5	1
5 == 5	1
3 == 5	0
i <= 3	0, se i>3 e 1, caso contrário

Ex2:

```
#include <stdio.h>
void main(void)
{   int verdad, falso;
    verdad = (15 < 20);
    falso =   (15 == 20);
    printf("Verdadeiro= %d, falso= %d \n", verdad, falso);
}
```

Note que o operador relacional "Igual a" é representado por dois sinais de igual. Se for usado apenas um sinal, o compilador entenderá como uma atribuição e não como comparação.

Ex:

x = = 2 está comparando se x é ou não igual a 2

x=2 está atribuindo o valor 2 à variável x (expressão verdadeira, por definição)

Operadores Lógicos - São chamados de "lógicos" porque seguem a Lógica Booleana de operação com bits. A diferença básica é que a rigor, a Álgebra Booleana só utiliza dois algarismos: o "0" e o "1", o "não" e o "sim", o "falso" e o "verdadeiro", o "espaço" e a "marca", etc. E em C, considera-se o número 0 como "falso" e todos os outros números como "verdadeiros". Os operadores lógicos são:

&&	AND
||	OR
!	NOT

A operação "E" (ou "AND") representada pelo símbolo "**&&**", exige que todos os operandos sejam verdadeiros para que sua saída seja verdadeira.

A operação "OU" (ou "OR") representada pelo símbolo "**||**", exige que ao menos um dos operandos seja verdadeiro para que sua saída seja verdadeira.

A operação "NÃO" (ou "NOT") representada pelo símbolo "**!**", inverte o operando. Se for falso, sua saída é verdadeira e vice-versa.

Abaixo temos as Tabelas-Verdade da Lógica Booleana:

operando1	operando2	AND	OR
falso	falso	falso	falso
falso	verdadeiro	falso	verdadeiro
verdadeiro	falso	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro	verdadeiro

operando	NOT
falso	verdadeiro
verdadeiro	falso

Ex:

Expressão	Tem o valor
5 || 3	1
5 || 0	1
5 && 3	1

5 && 0	0
0 || 0	0
i || j	0, se i e j forem 0 e 1, caso contrário
!5	0
!0	1

Operadores Bit a Bit - Realizam as mesmas operações que os lógicos, só que bit a bit do número. Operam apenas em números inteiros, em sua forma binária (tal como estão armazenados na memória), casa binária, por cada binária, por isto o nome.

São eles:

&	AND
|	OR
^	XOR
<<	deslocamento à esquerda
>>	deslocamento à direita
~	complemento de um (unário)

A operação "OU-EXCLUSIVO" (ou "XOR") representada pelo símbolo "^", exige que ou um ou outro dos operandos seja verdadeiro para que sua saída seja verdadeira, nunca todos ao mesmo tempo.

A operação de "deslocamento à esquerda" de bits $x<<y$ literalmente desloca os bits do número binário x, y vezes para a esquerda. Isto equivale a multiplicar um número binário x por 2, a cada deslocamento. Ou, em outras palavras: $x \times 2^y$. Os espaços criados no deslocamento são preenchidos com 0's.

A operação de "deslocamento à direita" de bits $x>>y$ literalmente desloca os bits do número binário x, y vezes para a direita. Isto equivale a dividir um número binário x por 2, a cada deslocamento. Ou, em outras palavras: $x / 2^y$. Os espaços criados no deslocamento são preenchidos com 0's.

A operação de "complemento de um" inverte todos os bits do número binário. Os que são "0" passam a ser "1" e vice-versa, e o correspondente valor binário é utilizado.

operando1	operando2	XOR
falso	falso	falso
falso	verdadeiro	verdadeiro
verdadeiro	falso	verdadeiro
verdadeiro	verdadeiro	falso

Ex:

Expressão	Em binários	Tem o valor .
1 | 2	0000.0001 | 0000.0010	0000.0011 = 3
0xFF & 0x0F	1111.1111 & 0000.1111	0000.1111 = 0x0F
0x0D << 2	0000.1101 << 2	00110100 = 0x34
0x1C >> 1	0001.1100 >> 1	0000.1110 = 0x0E
~0x03	compl(0000.0011)	1111.1100 = 0xFC
3 ^ 2	0000.0011 ^ 0000.0010	0000.0001 = 1

Operador de Atribuição - Em C, o sinal de igual não tem a interpretação dada em matemática. O que acontece é que o resultado ou valor do operando do lado direito é copiado, atribuído para a variável ou endereço, o operando do lado esquerdo. O operador de atribuição é:

= **Igual a**

Ex:

Expressão	Operação
i = 3	coloca o valor 3 em i
i = 3 + 4	coloca o valor 7 em i
i = (k=4)	coloca o valor 4 em k e depois de k para i
i =(k=4) + 3	coloca o valor 4 em k,a adição é feita e o valor 7 é colocado em i
3 = i	operação inválida! a variável deve estar do lado esquerdo

Operadores de Atribuição Reduzida - Compactam operações quaisquer seguidas de operação de atribuição e tornam o código mais rápido pois a variável utilizada só é procurada uma vez na memória.

Formato:
operação **=**

Ex:

Expressão	**É equivalente a** .
a += 2	a = a+2
j <<= 3	j = j << 3
q/= 7 + 2	q = q / (7+2)

Operadores Pré e Pós-Fixados - Realizam incremento ou decremento do valor de uma variável antes de seu valor ser utilizado, no caso de operador pré-fixado, ou depois de seu valor ser utilizado, no caso de operador pós-fixado. A vantagem de se utilizar estes operadores, e não o tradicional "variável = variável + 1;" é que além da praticidade e da compactação do código, torna o programa muito mais rápido.

++ incrementa de 1 seu operando
-- decrementa de 1 seu operando

Ex: Suponhamos a variável i = 5:

Expressão	**Valor expressão**	**Valor de i depois** .
5 + i++	= 10	6
5 + i--	= 10	4
--i + 5	= 9	4
++i + 5	= 11	6

Operador Condicional - Substitui com vantagens o loop: "Se *expressão1* é verdadeira Então *expressão2,* Senão *expressão3*". Sua sintaxe é:
exp1* ? *exp2* : *exp3

Ex:

Expressão	**Valor** .
5? 1 : 2	1
i? i+j : k+j	valor de i+j, se i não é zero e k+j, caso contrário
(m>7)? 3:4	3, se m maior que 7 e 4, caso contrário
c=(a>b)? a: b	devolve o maior valor, entre a e b, à variável c
d=(a>b)? ((a>c)? a:c): ((b>c)? b:c))	devolve o maior valor, o maior entre a, b e c, para d
e=(a>b)?((a>c)?((a>d)?a:d):((c>d)?c:d)):((b>c)?((b>d)?b:d):((c>d)?c:d));	maior entre a,b,c e d

Exemplos:

```
/*****************************************************/
/****      EXEMPLO 1:  Programa que mistura tipos int e char **/
/*****************************************************/
#include <stdio.h>
void main(void)
{  char c = 'a', ans;
   printf("O valor de c+3 = %c", c + 3);
   ans = c % 3;
   printf( "\n\nResto da divisão inteira = %d\n", ans);
}
```

```
/*******************************************************/
/***          EXEMPLO 2: Programa sobre o operador aritmético % */
/*******************************************************/
#include <stdio.h>
void main(void)
{   printf( "\n13 resto 3 = %d", 13 % 3);
    printf("\n-13 resto 3 = %d", -13 % 3);
    printf("\n13 resto -3 = %d", 13 % -3);
    printf("\n-13 resto -3 = %d", -13 % -3);
}
```

```
/*********************************************************/
/****        EXEMPLO 3: Programa para demonstrar operadores  ****/
/****        relacionais - Números primos                    ****/
/*********************************************************/
#include <stdio.h>
#include <math.h>
 void gera_primos(int limite);

void main(void)
{     int maximo;
      printf("\n Gerar numeros primos ate ?");
      scanf( "%d", &maximo);
      gera_primos (maximo);
      printf("\n");
}
void gera_primos (int limite)
{  int divisor;
   int candidato;
   int r=1;
   if (limite >=7)
   {    if (limite%2 == 0)       /* O limite superior é par */
           limite--;
        for(candidato = 3; candidato <= limite; candidato +=2)
        {       divisor = 3;
                while (divisor <= sqrt(candidato) && (r=candidato % divisor)!= 0)
                     divisor +=2;
                if (r !=0)
                   printf("%8d", candidato);                  /* numeros primos */
        }
    }
}
```

```
/*************************************************************/
/******     EXEMPLO 4: Programa para exibir o padrao de bits de um  */
/*****       inteiro sem sinal                                       */
/*************************************************************/
#include  <stdio.h>
void mostra_bits( unsigned especimen);

void main(void)
{  unsigned valor;
   mostra_bits (0);
   mostra_bits (5);
   mostra_bits (13);
   mostra_bits (117);
```

```
    mostra_bits (132);
    printf("\n\n Entre um numero: ");
    scanf("%u", &valor);
    printf("\n");
    mostra_bits(valor);
}

void mostra_bits(unsigned especimen)
{  const int pos_max = 15;
    int posicao_bit;
    printf("\n\b O número  %d na base binária \202: \n", especimen);
    for (posicao_bit = pos_max; posicao_bit >=0; posicao_bit--)
        printf("%d", especimen >> posicao_bit & 1);
}
```

Exercícios:

Implemente o programa que lê valores para duas variáveis inteiras, X e Y, e depois, dependendo da escolha do usuário, implementa uma das operações a seguir e imprime a resposta: (a) X AND Y; (b) X (OU-EXC bit-a-bit) Y; (c) X deslocado Y vezes para a direita.

5 - CONVERSÕES DE TIPOS

Apesar do tipo de cada variável e constante ser definido no início das funções, eventualmente uma variável ou constante precisará ser convertida para outro tipo, no meio do programa.

Uma conversão pode ocorrer porque definimos um operando como sendo de um tipo e depois precisamos compará-lo ou realizar alguma operação aritmética com este e outro operando de outro tipo. Ou então porque o tipo do operando não foi suficiente para armazená-lo durante todo o correr do programa. O fato é que conversões de tipos de variáveis e constantes acontecem e podem ser:

Conversão Automática - Ocorre quando tipos diferentes aparecem numa mesma expressão. A regra geral é converter o tipo "menor" para o tipo "maior".

Ex:

```
    int  x=5;
     float y = 2.4;
...      soma = x + y;        /* Nesta linha o inteiro x é convertido para o valor real 5.0 */
```

Conversão Forçada - Também conhecida como conversão "cast". Ocorre quando um tipo específico for necessário e o operando não foi definido como tal. A sintaxe é:

```
(tipo)  expressão
```

onde *tipo* é o tipo para o qual será convertido o operando e *expressão* é uma expressão que contém a variável ou constante a ser convertida.

Ex:

```
    float r = 3.5;
    int i;
....  i = (int) r;        /* Nesta linha o valor 3 (truncamento do real r) foi atribuído a i */"
```

Conversões de tipos são perigosas, pois além de poderem gerar um código não-portátil (pois o tamanho dos tipos varia de máquina para máquina), podem criar problemas de armazenamento dos operandos. Por exemplo, quando um número real muito grande, cuja representação da parte inteira exceda 16 bits (por exemplo 3.24e14), for convertido para inteiro, o resultado será imprevisível, pois não se pode determinar o que será feito com a parte inteira que não couber nos 16 bits de um int.

As regras para conversões automáticas são:

Tipo **Char** e **Short** - São convertidos automaticamente para **Int**. Utiliza-se o código ASCII do caractere. Se o caractere contém o bit MSB em 1, o inteiro resultante poderá ser negativo, ou não, dependendo da máquina. Se o caractere for **unsigned**, o inteiro será positivo e os bits de maior ordem serão preenchidos com zeros.

Tipo **Float** - É convertido para **Double**.

Conversão de Operandos Aritméticos - o tipo "maior", dentre os tipos da expressão será o do resultado. A regra é: double > long > unsigned > int.

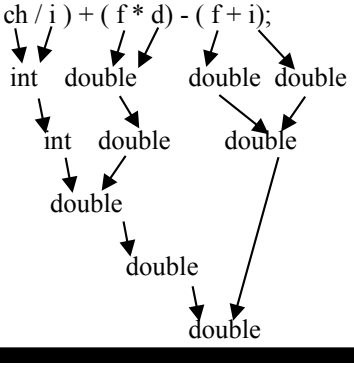
Conversão de Operandos de Atribuição - o lado direito será convertido para o tipo do lado esquerdo. A regra é:

Operando à direita	Operando à esquerda	Conversão
Double	Float	com arredondamento
Float	Int	trunca parte frac., se o número não couber em int: resultado indeterm.
Long	Int	elimina os bits MSB
Int	Char	elimina os bits MSB

Exemplo 1 – diferentes tipos numa mesma expressão:

```
char ch;
int i;
float f;
double d;

result = ( ch / i ) + ( f * d) - ( f + i);
```



Exemplo 2:

```
#include <stdio.h>
void main(void)
{    int i;
     for (i = 1; i <= 100; ++i)
     {   printf("%d/3 é: %f \n", i, (float) i/3);
         if(!(i%20))   getch();  //mostra vinte linhas e espera usuário teclar para continuar
      }
}
```

Exemplo 3:

A expressão: (float) x/2 , converte x e, por consequência, o 2 para float
Já a expressão: (float) (x/2) , converte o resultado inteiro de x/2 para float

6 - PRECEDÊNCIA

Os operadores obedecem uma certa ordem de precedência. Isto é: operações de maior precedência são realizadas antes das de menor precedência. O operador de mais alta precedência é o *abre* e *fecha-parênteses* ("()"), portanto, tudo que estiver entre eles será realizado primeiro. Por exemplo:

```
primeiro  *=   segundo  <= terceiro
```

Primeiramente o segundo operando é comparado ao terceiro e depois a atribuição ao primeiro é feita, porque o operador "**<=**" tem uma precedência maior que o "*=". Esta expressão é equivalente a:

```
primeiro *= (segundo <= terceiro )
```

Observe que a expressão (segundo **<=** terceiro) tem valor zero ou um.
No caso da expressão:

```
primeiro = segundo -= terceiro
```

os operadores "**=**" e "**-=**" têm igual precedência, mas o segundo operando é avaliado antes do primeiro porque os compiladores C costumam avaliar expressões condicionais e de atribuição da direita para a esquerda.

Existem 15 classes de precedência, ou seja, todos os operadores pertencentes à mesma classe têm igual precedência. Neste caso, valem as regras de associatividade, para determinar-se quais operações serão realizadas primeiro.

A tabela abaixo mostra as classes de precedência dos operadores, em ordem decrescente:

Operador	Nome do Operador	Precedência	Associatividade
( )	Chamada de função	1	esq. p/ dir.
[ ]	Elemento Matriz		
->	Ponteiro para Membro Estrutura		
.	Membro de Estrutura		
!	Negação Lógica	2	dir. p/ esq.
~	Complemento de um		
++	Incremento		
--	Decremento		
-	Menos unário		
(tipo)	Cast		
*****	Ponteiro		
&	Endereço		
sizeof	Tamanho do Objeto		
*****	Multiplicação	3	esq. p/ dir.
/	Divisão		
%	Resto da Divisão		
+	Adição	4	esq. p/ dir.
-	Subtração		
<<	Deslocamento à Esquerda	5	esq. p/ dir.
>>	Deslocamento à Direita		
<	Menor Que	6	esq. p/ dir.
<=	Menor ou Igual a		
>	Maior Que		
>=	Maior ou Igual a		
==	Igualdade	7	esq. p/ dir.
!=	Desigualdade		

&	AND, Bit a Bit	8	esq. p/ dir.
^	XOR, Bit a Bit	9	esq. p/ dir.
|	OR, Bit a Bit	10	esq. p/ dir.
&&	AND Lógico	11	esq. p/ dir.
||	OR Lógico	12	esq. p/ dir.
?:	Condicional	13	dir. p/ esq.
=	Atribuição	14	dir. p/ esq.
op=	Atribuição Reduzida		
,	Vírgula	15	esq. p/ dir.

7 - ORDEM DE AVALIAÇÃO

A ordem de avaliação de uma expressão indica se a operação será avaliada da direita pra esquerda ou o contrário.

Os operadores "&&", "||", e "," sempre são avaliados da esquerda para a direita. Fora este caso, a ordem de avaliação dependerá do compilador.

Ex:

```
        int i=5;
...         y = (++i) + (--i);      ...
```

Neste caso, y poderá receber o valor 5+4= 9, 6+5 = 11 ou até 10(???)!!! Dependerá do compilador.

8 - COMANDOS DE CONTROLE DE FLUXO

Considera-se comando válido em C, qualquer expressão válida, seguida por um ponto-e-vírgula (;), ou expressão entre chaves ({}).

Ex:

```
    a = 5;
```

Neste capítulo, entretanto, trataremos de comandos de controle de fluxo.

Pode-se dividir os comandos de controle de fluxo do C, em três categorias: instruções condicionais (**if** e **switch**), comandos de controle de loop (**while**, **for** e **do-while**) e instrução de desvio incondicional (**goto**).

8.1 - O COMANDO IF

A forma geral da declaração **if** é:

```
if(condição_de_teste)
    comando;
```

A interpretação é: "Se a *condição_de_teste* for verdadeira (não-zero), executa *comando*". Caso contrário, a execução é transferida para a instrução seguinte ao comando if.

Ex:

```
if (x==5)  y=3;
```

Se *comando* contiver mais de uma instrução, o bloco deve ser colocado entre chaves ({}):

```
if(condição_de_teste)
{       comando1;
        comando2;
        comando3;
          ...
}
```

8.2 - O COMANDO IF-ELSE

A forma geral da declaração **if-else** é:

```
if(condição_de_teste)
        comando1;
else    comando2;
```

A interpretação é: "Se a *condição_de_teste* for verdadeira (não-zero), executa *comando1*, Senão, executa *comando2*".

Ex:

```
if ( x < 6)
     y = 1;
else
     y = 2;
```

Os comandos **if** e **if-else** podem ser aninhados!!!

Isto ocorre quando uma série de testes sucessivos tem que ser feita, para fazer-se a escolha da instrução a ser executada. A sintaxe pode ser tal como:

```
if(condição1)
        if(condição2)
                comando1;  //instrução executada quando condição1 e condição2 forem V
        else
                comando2;  //instrução executada quando condição1 for V e condição2 for F
else

        if(condição3)
                comando3;  //instrução executada quando condição1 for F e condição3 for V
        else
                comando4; //instrução executada quando condição1 e condição3 forem F
```

Exercício: Um professor de educação física especificou a tabela abaixo que define o tempo de treinamento para alunos do sexo feminino e masculino, jovens ou adultos. Escreva o programa que implementa a tabela abaixo, isto é, lê a idade e sexo do usuário e devolve o tempo de treinamento recomendado.

	Sexo:	
Idade:	**Feminino**	**Masculino**
≤ 30 anos	t = 15’	t = 45’
> 30 anos	t = 25’	t = 60’

Mas há que se ter cuidado com o que é interno, o que é externo e a qual if pertence cada else e com a ***endentação***!!!!!

Ex:

O algoritmo:
```
if ( i > 2)
    if ( j == 3)
        y = 4;
    else  y = 5;
```
é diferente de:
```
if ( i > 2)
{       if( j == 3)
            y = 4;
} else y = 5;
```
e de:
```
if (i > 2);
if ( j == 3)
    y = 4;
else y = 5;
```
e de: ...
```
if (i > 2)
{       if( j == 3);
            y = 4;
} else y = 5;
```

No primeiro caso, o else refere-se ao if mais interno e no segundo caso, ao if externo, pelo uso das chaves. No terceiro caso e no quarto casos, o ponto-e-vírgula "terminou" a instrução if, antes que esta executasse qualquer comando interno.

Obs:

```
if (a > b)   c = a;
else         c = b;
```
É equivalente a: `c = ( a > b ) ? a : b ;`

Obs2: Em expressões *condição_de_teste* não-relacionais deve-se tomar cuidado:

`if (i == 3)  y = 5;` → se i for igual a 3, y igual a 5

`if (i = 3)   y = 5;` → se i=3 for não zero , y igual a 5 (i=3 é TRUE).

Obs3:

`if ( i != 0)  y = 3;` é equivalente a `if (i) y = 3;`

Exercício:

Crie um programa de adivinhação, utilizando os comandos if e if-else. O programa pede um número ao usuário, verifica se este é igual ao número mágico (previamente definido) e imprime " ** Certo **", caso a pessoa tenha acertado ou " ** O número mágico é maior que .... ** " o número que a pessoa digitou, ou ainda " ** O número mágico é menor que ... ** " o número digitado.

Exemplos

```
/*************************************************************************/
/***              Programa 1:  Exemplo de Conversão Automática          **/
/*************************************************************************/
#include <stdio.h>
void main(void)
{
     char ch;
     int i;
     float fl;

    fl = i = ch = 'A';                                  // o caractere 'A' é armazenado
    printf("ch = %c, i = %d, fl = %2.2f \n", ch, i, fl);   // como 65 (em i) e 65.00 (em fl)
    ch = ch + 1;                                        // converte ch para int, soma 1 e reconverte para char
    i = fl + 2 * ch;     // converte ch para int, multiplica 2, transforma em float, soma a fl e  converte para int
    fl = 2.0 * ch + i;                  // converte ch em float, multiplica por 2.0, i é convertido em float  e
    printf("ch = %c, i = %d, fl = %2.2f \n", ch, i, fl);    //somado a ch, o resultado é armazenado em fl
    ch = 5212205.17;                                    // o valor excede 8 bits, portanto é truncado e o código
    printf ("Agora ch = %c\n", ch);                     //ASCII é armazenado em ch
}
```

```
/*************************************************************************/
/*** Programa 2: Demonstrar Conversão Forçada de Dados     (Cast)      ******/
/*************************************************************************/
#include <stdio.h>
void main(void)
{       int valor1, valor2;
        valor1 = 1.6 + 1.7;
    valor2 = (int) 1.6 + (int) 1.7;
```

```
   printf("valor1 = %d\n", valor1);
   printf("valor2 = %d\n", valor2);
}
```

```
/*********************************************************************/
/***  Programa 3:  Demonstrar Estrutura IF - ELSE                  ****/
/*********************************************************************/
/*   Este programa localiza a percentagem de dias abaixo gelados!!    */
#include <stdio.h>
#define ESCALA "celsius"
#define GELANDO 0
int main(void)
{
   float temperatura;
   int gelando = 0;
   int dias =0;

   printf("Entre com a relacao das temperatura dos dias gelados.\n");
   printf("Use escala %s, e entre s para sair.\n", ESCALA);
   while (scanf("%f", &temperatura) == 1)
   {    dias++;
        if(temperatura < GELANDO)
           gelando++;
   }
   if (dias!=0)
      printf("Do total de %d dias: %.1f\% foram abaixo de zero.\n", dias, 100.0*(float) gelando/dias);
   else
      printf("Nao foi fornecido nenhum dado!\n");
  return 0;
}
```

8.3 - O COMANDO WHILE

A forma geral da declaração *while* é:

```
while( expressão_de_teste)
        comando;
```

A interpretação é: "Enquanto *expressão_de_teste* for verdadeira; execute *comando*". No momento em que *expressão_de_teste* deixa de ser não-zero, a execução continua na linha de comando seguinte ao laço while.

Se houver vários comandos internos ao loop while, estes devem estar entre chaves ({ }).

Ex:

```
        i=0;
...     while(i < 10)
        {   a = b * 2;
            chama_função( );
            i++;
        }
```

É importante salientar que se a expressão_de_teste não for verdadeira já no primeiro teste do laço este não será executado nenhuma vez e que o comando while é mais apropriado para laços onde o número de interações não é conhecido de antemão. Por exemplo, como saber quantas vezes o usuário vai digitar caracteres para um número ou uma string de entrada?

Ex:

```
#include  <stdio.h>
  #include <conio.h>
   void main(void)
   { int cont=0;
    printf("Digite uma frase: \n");
    while(getche()!= 13)        /* O caractere com ASCII igual a 13 é a tecla enter (return) */
       cont++;
    printf("\nO numero de caracteres é %d", cont); }
```

Ex2:

```
 i = 0;                                          i = 5;
... while (i<5)          É equivalente a        ... while (i)
      i++; ...                                        i--;  ...
```

8.4 - O COMANDO DO-WHILE

A forma geral da declaração *do-while* é:

```
do
   comando;
 while(expressão_de_teste);
```

A interpretação é: "Faça *comando* enquanto *expressão_de_teste* for verdadeira".O comando *do-while* faz quase o mesmo que o *while*, com a diferença que no primeiro, o loop é executado pelo menos uma vez, já que o teste da expressão é feito no final da interação. Ou seja, se *expressão_de_teste* for falsa já na primeira interação, *comando* é executado uma vez e em seguida a execução continua fora do loop, na próxima linha de comando. Caso *expressão_de_teste* seja verdadeira, *comando* será executado até que esta se torne falsa.

Exercício:

Adapte o programa do número mágico, para uso com estrutura *do-while*.

8.5 - O COMANDO FOR

A forma geral da declaração *for* é:

```
for(inicialização;teste;incremento)
     comando;
```

Em sua forma mais simples, *inicialização* é uma instrução de atribuição (p.e.: i = 0) e é sempre executada uma única vez antes do laço ser inicializado.

O *teste* é uma instrução condicional que controla o laço. *Comando* será executado até que *teste* seja falso.

A expressão de *incremento* (ou decremento) define a maneira como a variável de controle do laço será alterada a cada interação.

Ex:

```
  for( i = 0; i < 5; i++)          →  Para i de 0 até 4:
     j++;                                 incrementa j a cada interação e incrementa i
  for(i = 5; i > 0; i--)           →  Para i de 5 até 0:
      j = j * 2;                          novo valor de j é j * 2  e decrementa i
  for(;;)  { ... }                 →  Loop eterno
  for(i = 0; i< 5; i++)              É equivalente a        for (i = 0; i< 5; j++, i++);"
     j++;
```

Exercícios

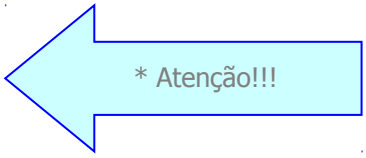
1 - Faça um programa que imprima os números de 1 a 10, utilizando:
a) comando while,
b) comando do-while,
c) comando for.

2 - Faça um programa que imprima os números de 0 a 9, de 2 em 2, utilizando o comando for.
3 - Faça um programa que imprima o fatorial de um número solicitado ao usuário, utilizando o comando while.

8.8 - O COMANDO SWITCH

A forma geral da declaração *switch* é:

```
  switch (exp_int)
{   case rot1:
                cmd1
    case rot2:
                cmd2
    ...
    default:
                cmdn
}
```



* Os comandos *cmd1*, *cmd2*, etc e a declaração *default* são opcionais no bloco.

Este comando testa o valor da expressão inteira *exp_int*, comparando-a com *rot1*, *rot2*, etc, até encontrar um rótulo que se iguale. Quando encontra, começa a executar de cima para baixo os comandos *cmd1*, *cmd2*, etc, até o final do bloco. Se não encontra, executa o comando do bloco *default*, *cmdn*.

Ex:
```
switch (i)
{   case 1:  j = j + 5;
    case 2:
    case 3:  j = j + 3;   }
```

Valor de i	**Comandos executados**
1	j = j + 5; e j = j + 3;
2 ou 3	j = j + 3;
qualquer outro	nenhum.

Utiliza-se a instrução *break* para que apenas o comando referente a cada rótulo seja executado.
Ex:
```
switch ( i )
{ case 1:   j = j + 5;
            break;
  case 2:
  case 3:   j = j + 3;
            break;
  default:  j = j+ 1;    }
```

Valor de i	**Comandos executados**
1	j = j + 5;
2 ou 3	j = j + 3;
qualquer outro	j = j + 1;

Exercícios

```
/*************************************************************************/
/****       Programa que imprime números de 1 a 10 utilizando laço while:          *****/
/*************************************************************************/
#include <stdio.h>
void main(void)
{  int contador = 1;
  while (contador <= 10)
  {   printf("%d\n", contador);
      contador++;
  }
}
```

```
/*************************************************************************/
/****     Programa que imprime números de 1 a 10 utilizando laço do-while          ****/
/*************************************************************************/
#include <stdio.h>
void main(void)
{   int contador = 1;
   do
   {  printf("%d\n", contador);
     contador++;
   }
   while (contador <= 10);
}
```

```
/*************************************************************************/
/***    Programa que imprime números de 1 a 10 utilizando laço for                *****/
/*************************************************************************/
#include <stdio.h>
void main(void)
{    int contador;
    for( contador = 1; contador <= 10; contador++)
       printf("%d\n", contador);
}
```

```
/*************************************************************************/
/****       Programa que imprime números de 0 a 9, de 2 em 2                    *****/
/*************************************************************************/
#include <stdio.h>
void main(void)
{    int i;
     for ( i = 0; i <= 9; i+=2)
        printf("%d\n", i);
}
```

```
/*************************************************************************/
/*****    Programa que calcula o fatorial de um número                      *******/
/*************************************************************************/
#include <stdio.h>
#include <conio.h>
void main(void)
```

```
{    int numero, j;
     char ch;
     double fat=1;

     for(;;)
     { fat=1;
      printf("Entre com um número positivo:  \n");
      scanf("%d", &numero);
      if(numero== 0 || numero == 1)
         printf("O fatorial de %d é: %.0f\n", numero, fat);
      else
      {   j = numero;
          while(j)
          {   fat  *=  j;
              j--;
           }
          printf("O fatorial de %d é: %.0f\n", numero, fat);
      }
     }
}
```

```
/*************************************************************************/
/***     Programa que gera a tabuada de 2 a 9                          ******/
/*************************************************************************/
#include <stdio.h>
void main(void)
{   int i, j, k;
    printf("\n");
    for (k = 0; k <= 1; k++)
    {   printf("\n");
        for( i = 1; i < 5; i++)
           printf("TABUADA DO %3d  ", i+4*k+1);
        printf("\n");
        for( i = 1; i <= 9; i++)
        {    for( j = 2 + 4 *k; j <= 5 + 4*k; j++)
                   printf("%3d x %1d = %3d\t", j, i, j*i);
             printf("\r");
         }
    }
}
```

```
/*************************************************************************/
/****  Programa da feira  de frutas                                  ****/
/*************************************************************************/
#include <stdio.h>
#include <conio.h>
void main(void)
{  int i, opcao;
     printf("\n");
     for(i = 1; i <= 53; i++)
       printf("*");
     printf("\n*****\t\tPROGRAMA DA FEIRA!!\t\t*****\n");
     for(i = 1; i <= 53; i++)
       printf("*");
     printf("\n\n\t\t Escolha sua opção: \n");
     printf("\n\n\t\t(1) Uva: \n");
```

```
        printf("\n\t\t(2) Maça; \n");
        printf("\n\t\t(3) Banana; \n");
        printf("\n\t\t(4) Laranja; \n");
        scanf("%d",&opcao);
        switch(opcao)
        {  case 1:  printf("O cacho de uvas custa R$1.00");
                    break;
           case 2:  printf("A unidade de maças custa R$0.50");
                    break;
           case 3:  printf("O kilo de bananas custa R$0.70");
                    break;
           case 4:  printf("A duzia de laranjas custa R$0.90");
                    break;
           default: printf("Desculpe, mas não temos esta fruta!!");
        }
}
```

Exercícios

1) Um certo Centro Acadêmico está tentando realizar um plebiscito para escolha do logotipo do curso, dentre 3 propostas. Faça o programa que implementa a "urna eletrônica", a qual vai contabilizar os votos dos estudantes e professores em cada logotipo, os votos brancos e nulos; calcular os percentuais de votos válidos em cada logotipo e imprimir qual foi o logotipo vencedor.

2) Faça um programa-calculadora de 4 funções, ou seja, o usuário entra com dois números e estes são somados, subtraídos, multiplicados ou divididos.

9 - FUNÇÕES

Uma função é uma unidade de código de programa autônoma projetada para cumprir uma tarefa particular. Funções permitem grandes tarela, fas de computação em tarefas menores e permitem às pessoas trabalharem sobre o que outras já fizeram, ao invés de partir do nada.

A linguagem C em si, não possui funções pré-definidas. Todas as funções utilizadas em C foram projetadas pelos próprios usuários e algumas mais usadas já foram incorporadas às bibliotecas de alguns compiladores. Um exemplo de função em C é **printf()**, que realiza saídas dos programas sem que o usuário precise preocupar-se como isto é feito, pois alguém já fez isto e vendeu sua idéia aos outros usuários.

A principal razão da existência de funções é impedir que o programador tenha de escrever o mesmo código repetidas vezes.

As funções em C são utilizadas como *funções* (retornam valores; podem ser chamadas de dentro de uma expressão e não recebem parâmetros) e *subrotinas* (não retornam valores; são chamadas por um comando CALL e recebem parâmetros) das outras linguagens. No entanto, não pode haver aninhamento de uma função dentro de outras funções. Cada bloco de um programa em C é uma e somente uma função.

Sintaxe:

```
tipo  nome_da_função(declaração de parâmetros formais)
{   declaração de variáveis
    comandos
}
```

Onde:

tipo - tipo do valor de retorno da função. Se uma função não retornar nenhum valor deve-se usar o tipo "void", pois, por *default*, as funções em C/C++ retornam um inteiro. Ex: "void main(void)".

nome_da_função - nome da função. Como qualquer identificador em C, o nome não pode ser uma palavra reservada da linguagem (a não ser no caso da função *main()*), pode ser composto por letras, números e o caractere de sublinhado (também chamado *underscore:* "_"), mas deve iniciar com uma letra ou com o *underscore*.

declaração de parâmetros formais - neste campo são declarados os parâmetros que a função recebe. Se a função não receber nenhum parâmetro, em alguns compiladores exige-se a utilização de "void", em outros, basta a omissão (quando então os parâmetros são assumidos como inteiros). Os nomes dos parâmetros devem ser separados por vírgulas. Ex: " int sqrt(x, y)".

Em alguns compiladores (como é o caso do Turbo C), a declaração das variáveis utilizadas na função deve, obrigatoriamente, preceder quaisquer comandos da função. Deve vir logo depois do caractere de abre-chaves ("{"). Existem outros compiladores que aceitam esta declaração em qualquer linha da função, desde que precedendo a utilização das mesmas.

comandos - Além dos comandos do corpo da função, este bloco pode conter o comando **return** que finaliza a execução da função e retorna o valor para a expressão que a chamou. Caso não haja "return", este será assumido quando o compilador encontrar o caractere de fecha-chaves ("}") e o valor retornado será indefinido.

Ex:

```
somaum ( int  numentra)
{ int numsai;
  numsai = numentra + 1;
  return numsai;
}
```

Exercício:
Escreva a função que recebe, calcula e devolve a média de 4 valores.

9.1 - CHAMADA DA FUNÇÃO

Vimos até agora, como é a sintaxe da execução do corpo de uma função chamada em um expressão. Mas qual é a sintaxe da chamada de uma função? Seja numa expressão, ou não, a sintaxe é:

nome_da_função(*argumentos*);

Na chamada de função em C não se utiliza CALL. Note que o que diferencia a chamada de uma função, da declaração da mesma é a utilização do ponto-e-vírgula (";"). Os *argumentos* são valores passados para a função. Quando não houver argumentos a serem passados, deixa-se este espaço em branco.

Ex:

```
somaum(5)                          → Retorna o valor 6
maior = acha_num_maior(4,7,2,5);     → Retorna o valor 7 para a variável maior.
```

9.2 - PARÂMETROS E ARGUMENTOS

Argumento é o valor passado para uma função.
Parâmetro (Formal) é a variável que recebe valor do argumento.
Ex:

```
#include <stdio.h>
mult(int,int);
void main(void)
 {   int a=4, b=5;
     printf(" O valor da multiplicação de %d por %d é  %d\n", a, b, mult(a,b));
 }
```

argumentos

```
mult(int x, int y)
{  int resultado;
   resultado = x * y;
   return resultado;
}
```

parâmetros formais

Normalmente, C utiliza passagem de parâmetros "**por valor**" para funções. Isto é, os argumentos recebem cópias dos valores das variáveis na expressão. Temos isto ilustrado no exemplo acima. Quando a função *mult()*, recebe os argumentos *a* e *b*, na verdade apenas cópias dos valores de *a* e *b* são enviados para a função *mult()*. Em resumo, as variáveis *a* e *b* não tem seus valores modificados após terem sido utilizadas como argumentos de uma função.

Se deseja-se que a própria variável seja passada para a função que vai modificar seu valor utilizamos a passagem "**por referência**". Neste caso, o argumento recebe o endereço de memória da variável e a função chamada modifica o conteúdo deste endereço diretamente. Para passar valores por referência, utiliza-se os operadores "&" e "*":

& - " o endereço de " - endereço da variável

* - " no endereço de" - o que está contido no endereço da variável.

Ex:

```
int saida = 5;
   ...
   incrementa(&saida);
   ...
   incrementa(int *numentra)       /* numentra contém o endereço e não o valor de saida  */
/* conteúdo do endereço numentra é do tipo int */
   {  (* numentra)++;              /* incrementa o conteúdo de numentra  */
      return;
   }
```

→ No final da função *incrementa()*, saida tem o valor 6

9.3 - VALORES DE RETORNO

Quando o tipo de valor de retorno da função não é especificado, por *default* a função vai retornar um valor inteiro.

Quando a função deve retornar um tipo que não o *int*, é necessário declarar-se o mesmo.

Quando a função não retorna nada, no caso de compiladores C ANSI, o tipo deve ser *void*.

9.4 - ESCOPO DE VARIÁVEIS

Um programa em C é um conjunto de uma ou mais funções, sendo que uma destas funções é a principal (main()), que será a primeira a ser executada. Como saber a que função pertence determinada variável, como seu valor muda de função para função e em qual(is) função(ões) ela existe?

Variáveis Locais ou Automáticas

São todas as variáveis declaradas dentro de uma função. Como só existem enquanto a função estiver sendo executada, são criadas quando tal função é chamada e destruídas quando termina a execução desta. Parâmetros formais são variáveis locais.

Somente podem ser referenciadas pela função onde foram declaradas e seus valores se perdem entre chamadas da função.

Ex:

```
void func1(void)
 {  int x;
    x = 10; }
 void func2(void)
 { int x;
   x = -199;      }
```

→ O x da func1() e o x da func2() são duas variáveis diferentes, armazenadas em posições de memória diferentes, com conteúdos diferentes, apesar do mesmo nome.

Uma variável local deve ser declarada no início da função (antes de qualquer comando), por motivos de clareza e organização do código e porque alguns compiladores assim o exigem. Existem

compiladores, no entanto, que permitem que a declaração seja feita em qualquer ponto do corpo da função, desde que antes da utilização da variável.

Variáveis Globais

São variáveis declaradas e/ou definidas fora de qualquer função do programa. Podem ser acessadas por qualquer função do arquivo e seus valores existem durante toda a execução do programa. Também por motivos de clareza convenciona-se declará-las no início do programa, após os comandos do pré-processador e das declarações de protótipos de funções.

Ex:

```
...
 int conta;              /* conta é global  */
 void main(void)
 {   conta = mul(10,123);
 ...  }
func1()
{ int temp;
  temp = conta;
...   }
func2()
{   int conta;
    conta = 10;      /* esta conta é local   */
...   }
```

Variáveis externas

Um programa em C pode ser composto por um ou mais arquivos-fonte, compilados separadamente e posteriormente *linkados*, gerando um arquivo executável. Como as várias funções do programa estarão distribuídas pelos arquivos-fonte, variáveis globais de um arquivo não serão reconhecidas por outro, a menos que estas variáveis sejam declaradas como externas. A variável externa deve ser definida em somente um dos arquivos-fonte e em quaisquer outros arquivos deve ser referenciada mediante a declaração com a seguinte sintaxe:

extern *tipo_var nome_var*;

onde *tipo_var* é o tipo da variável e *nome_var*, o nome desta.

Ex:

Arquivo 1

```
int x, y;
 char ch;
 void main(void)
 { ... }
 func1()
 {    x = 123;
 ...   }
```

Arquivo 2

```
extern int x, y;
    extern char ch;
    func23()
     {   x = y/10;
     }
     func24()
{  y = 10; }
```

Variáveis Estáticas

São variáveis reconhecidas e permanentes apenas dentro dos arquivos-fonte ou funções onde foram declaradas. Uma variável estática mantém seus valores entre chamadas da função o que é muito útil quando se quer escrever funções generalizadas (sem o uso de variáveis globais) e biblioteca de funções. A sintaxe é:

static *tipo_var nome_var*;

onde *tipo_var* é o tipo da variável e *nome_var*, o nome desta.

Ex:

```
static int rand(void)
 {    static int semente = 1;
      semente = (semente * 25173+ 13849)%65536;     /* formula magica  */
      return (semente);
 }
 ...
 void main(void)
 {   int c;
     for(c=1; c<=5; c++)
         printf("Número randômico: %d \n", rand());
 }
```

A saída deste programa será:

```
Número randômico:  -26514
Número randômico:  -4449
Número randômico:  20196
Número randômico:  -20531
Número randômico:  3882
```

Variáveis Registradores

Uma variável declarada com o modificador **register** indica ao compilador para utilizar um registrador da CPU, ao invés de alocar memória para a variável. Variáveis armazenadas em registradores são acessadas muito mais rápido que as armazenadas em memória, o que aumenta muito a velocidade de processamento. Se o número de variáveis designadas como **register** exceder o número disponível de registradores da máquina, então o excesso será tratado como variáveis automáticas.

Variáveis registradores não podem ser globais e geralmente aplicam-se aos tipos *int* e *char*.

Obs: Existem programadores que costumam colocar variáveis contadoras em registradores, para tornar o processamento o mais rápido possível.

Ex:

```
/***********************************************/
// Este programa mostra a diferença que uma variável register
// pode fazer na velocidade de execucao de um programa
/***********************************************/
#include <stdio.h>
#include <time.h>
unsigned int i;   // variável não-register
unsigned int delay;

void main(void)
{   register unsigned int j;
   long t;
   t = time('\0');
   for(delay = 0;delay < 50000; delay++)
      for(i = 0; i< 64000; i++) ;
   printf("tempo de loop não register: %d \n", time('\0')-t);
   getch( );
   t = time('\0');
   for(delay = 0; delay < 50000; delay++)
      for(j=0; j< 64000; j++) ;
   printf("tempo do loop register: %d \n", time('\0')-t);
}
```

10. MATRIZES

São grupos (de uma ou mais dimensões) de variáveis indexadas, do mesmo tipo. Matrizes unidimensionais são mais conhecidas por vetores ou "*arrays*".

Sintaxe:

tipo nome[*tamanho*] = { elem0, elem1, ... , elemn} ;

onde,

tipo: tipo dos elementos da matriz;

nome: nome da matriz;

tamanho: número de elementos da matriz;

= { } : termo opcional. Representa a definição de uma matriz, já na declaração. Caso a matriz seja definida na declaração é desnecessário especificar-se o *tamanho* desta.

elemx: elemento da matriz de índice x.

Todas as matrizes e vetores em C iniciam pelo **elemento de índice 0 (zero)**. Elementos não-inicializados recebem o valor 0 (zero) por *default*.

Os elementos da matriz são referenciados individualmente pela especificação do nome da matriz, seguido do índice do elemento entre colchetes:

nome[*índice*]

Ex:

Matriz que armazena as notas das 4 provas de um aluno:

```
float notas[4] = {9.5, 5.0, 10, 6.8};
  ...
  notas[0] = 9.5;
  notas[2] = 10;
  ...
```

ou

```
float notas[4];
  ...
  notas[0] = 9.5;
  ...
  notas[3] = 6.8;
```

ou

```
float notas[] = { 9.5, 5.0, 10, 6.8};
  ...
  notas [0] = 9.5;
  ...
```

10.1 - STRINGS

Uma string em C equivale a um vetor de caracteres sempre terminado pelo caractere NULL ('\0').

Para inicializar-se um vetor de caracteres pode-se fazê-lo individualmente, elemento a elemento, ou não. Mesmo que se esqueça de incluir o caractere NULL, este será acrescentado à string. Além disto, deve-se sempre lembrar de dimensionar o tamanho da string como o número de caracteres da expressão + 1 (para o caractere NULL).

Exs:

```
char matcar[] = "ABCD";
```

onde:

```
matcar[0] = 'A';
matcar[1] = 'B';
matcar[2] = 'C';
matcar[3] = 'D';
```

```
matcar[4] = '\0';
```

ou

```
char matcar[5] = { 'A', 'B', 'C', 'D', '\0'};
```

se:

```
char matcar[10] = "ABCDE";
```

temos:

```
matcar[0] = 'A';
matcar[1] = 'B';
...
matcar[4] = 'E';
matcar[5] = '\0';
matcar[6] = '0';
matcar[7] = '0';
matcar[8] = '0';
matcar[9] = '0'
```

A string nula ("") é: { '\0'}.

10.2 - MATRIZES MULTIDIMENSIONAIS

Para representar, por exemplo, a matriz bidimensional "Mat", abaixo:

$$Mat = \begin{vmatrix} 2 & 4 & 6 & 8 \\ 1 & 2 & 3 & 4 \\ 7 & 8 & 9 & 0 \end{vmatrix}$$

utilizamos a sintaxe:

tipo nome [*número_linhas*] [*número_colunas*];

ou seja, no primeiro par de colchetes indicamos o número de linhas da matriz e no segundo par, o número de colunas. O tamanho da matriz será:

número_linhas X *número_colunas*

No caso de matrizes multidimensionais, a sintaxe é:

tipo nome [tamanho1] [tamanho2] ... [tamanhon];

onde *tamanhoX* é o número de elementos da X-ésima dimensão.
Ex:

```
int mat [3] [4] =  {  { 2, 4, 6, 8 },
                      { 1, 2, 3, 4 },
                      { 7, 8, 9, 0 }  } ;
```

onde:

```
elemento mat [0] [0] = 2;
elemento mat [2] [3] = 0;
elemento mat [1] [2] = 3;
etc.
```

Exercícios

Faça um programa que cria uma matriz de 3 dimensões (2 x 4 x 2) e atribui valores a mesma.

10.3 - MATRIZES PASSADAS PARA FUNÇÕES

Na linguagem C, o nome de uma matriz é equivalente ao **endereço do primeiro elemento da matriz**! Isto significa que quando queremos passar uma matriz como argumento ("por referência") a uma função, basta utilizar o nome da matriz, sem os colchetes e índices. Se apenas um elemento da matriz deve ser modificado, então utiliza-se seus índices em colchetes.

Ex:

```
/*****************************************/
/*** Programa que converte uma string para maiúscula ***/
/*****************************************/
#include <ctype.h>
#include <stdio.h>
void imprime_maius(char[81]);
void main(void)
{  char s[81];
   printf("Digite uma frase");
   gets(s);
   imprime_maius(s);
}
void imprime_maius(char string[])
 {  register int t;
    for(t=0;string[t]; t++)
    {   string[t] = toupper(string[t]);
        printf("%c", string[t]);
    }
 }
```

```
/***********************************/
/*** MATRIZ DO JOGO DA VELHA      ***/
/***********************************/
#include <stdlib.h>
#include <stdio.h>

void pega_mov_jogador(void);
void pega_mov_computador(void);
void exibir_matriz(void);
int check(void);

char matriz[3][3] = { ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '};

void main(void)
{  char feito;

   printf(" ** Este e' o jogo da velha!! ***\n");
   printf(" Voce vai jogar contra o computador \n");
   feito = ' ' ;
   do
   {    exibir_matriz();
   pega_mov_jogador();
   feito = check();   // verifica quem ganhou
   if(feito != ' ' )
      break;
   pega_mov_computador();
   feito = check();    // verifica quem ganhou
   }while (feito == ' ' );
   if (feito== 'X' ) printf(" Voce venceu!*** \n");
```

```
  else printf(" Eu ganhei!!\n");
  exibir_matriz();     // mostra posicoes finais
}

void pega_mov_jogador(void)
{   int x, y;
    int ok = 0;
    printf(" Digite as coordenadas para o seu X: " );
    do
    {  scanf(" %d %d", &x, &y);
       x--; y--;
       if (matriz[x][y] != ' ')
      printf(" Movimento invalido, \nTente de novo.\n");
       else
       {     matriz[x][y] = 'X';
             ok = 1;}
    }while(!ok);  }
void pega_mov_computador(void)
{  register int t, i;
   // procura lugar nao usado ainda
   for(t=0;t<3;t++)
      for(i=0;i<3;i++)
    if(matriz[t][i] == ' ')
       if(t*i==9)
       {  printf(" Empate!! \n");
          exit(0);  // terminar o programa
       }else
       {matriz[t][i] = 'O';
        t=3; i=3;
            }
}

void exibir_matriz(void)
{  int t;
   for(t=0;t<3;t++)
   {   printf(" %c | %c | %c " , matriz[t][0], matriz[t][1], matriz[t][2]);
       if(t!=2) printf("\n---|---|---\n");
   }
   printf("\n");
}

// verifica se existe um ganhador; caso contrario, retorna ' '
int check(void)
{  int t;
   for(t=0;t<3; t++)
     if(matriz[t][0]==matriz[t][1] && matriz[t][1]==matriz[t][2])
       return matriz[t][0];
   for(t=0;t<3;t++)
     if(matriz[0][t]==matriz[1][t] && matriz[1][t]==matriz[2][t])
       return matriz[0][t];
   if(matriz[0][0]==matriz[1][1] && matriz[1][1]==matriz[2][2])
     return matriz[0][0];
   if(matriz[0][2]==matriz[1][1] && matriz[1][1]==matriz[2][0])
     return matriz[0][2];
   return ' ';
}
```

10.4 - ORGANIZAÇÃO DE MATRIZES NA MEMÓRIA

Como já foi dito, o nome de uma matriz contém o endereço do primeiro elemento da matriz (ou seja, é um ***ponteiro*** para o primeiro elemento) . Além disto, os elementos são armazenados um em seguida do outro, em endereços consecutivos. Ou seja, numa matriz de caracteres (variáveis de 1 palavra), se o endereço do primeiro elemento (índice 0) for 1500, por exemplo, o endereço do segundo (índice 1) será 1501 e assim por diante.

Quando utilizamos o nome de uma matriz como parâmetro para uma função, não estaremos passando a matriz, mas na verdade apenas o endereço do primeiro elemento desta, e por consequência, os outros.

Ex:

```
char alfa[27];
  ...
  alfa[0] = 'A';
  alfa[1] = 'B';
   ...
  alfa[25] = 'Z';
   ...
   escreve(alfa);
  ...
```

Note que, apesar de termos reservado 27 bytes para a matriz 'alfa', utilizamos apenas 26. Como trata-se de uma string, o último lugar é reservado para o caractere NULL ('\0').

Na memória, a representação desta matriz seria (suponha o endereço de 'alfa' como sendo 1492 e do primeiro elemento, alfa[0] como sendo 1500):

1490	**1491**	**1492**	**1493**	**1494**	**1495**	**1496**	**1497**	**1498**	**1499**
X	X	alfa	X	X	X	X	X	X	X
X	X	1500	X	X	X	X	X	X	X

1500	**1501**	**1502**	**1503**	**1504**	**1505**	**1506**	**1507**	**1508**	**1509**
alfa[0]	alfa[1]	alfa[2]	alfa[3]	alfa[4]	alfa[5]	...	...	...	...
65	66	67	68	69	70	...	...	...	...

...

1520	**1521**	**1522**	**1523**	**1524**	**1525**	**1526**	**1527**	**1528**	**1529**
...	...	...	...	alfa[24]	alfa[25]	alfa[26]	X	X	X
...	...	...	...	89	90	00	X	X	X

onde o valor 65 é o código ASCII do caractere 'A' , 90 é o código do caractere 'Z' e 00 é o código do caractere NULL.

Observação Importante sobre Matrizes:

A linguagem C não verifica se alguma matriz tomar espaço de memória que não devia. Por exemplo:

```
int notas[30];
...
for(i=0; i<40; i++)
   scanf("%d", notas + i);
```

No exemplo acima, declaramos a matriz notas como tendo 30 elementos e depois, por descuido provavelmente, atribuímos 40 valores a esta (notas[0] a notas[39]). Neste caso, 10 espaços de memória seguintes ao espaço ocupado pela matriz, serão apagados e receberão valores de notas. Isto pode

danificar dados importantes do programa, ao apagar os valores de variaveis ou constantes, ou mesmo partes do código. Não se pode prever o que aconteceria.

A responsabilidade do programador redobra, portanto, ao lidar com matrizes e vetores e é preciso muita atenção.

Outro erro que poderia ocorrer no exemplo acima, seria pedir a entrada de valores tipo *float* (usando "%f", por exemplo) em *scanf()* para elementos de uma matriz de inteiros. Como o tipo *float* é maior que o tipo *int*, cada elemento não caberia no espaço reservado para si e tomaria memória do elemento seguinte.

* Note que utilizamos aqui a expressão " scanf("%d", notas + i); ", ao invés de "scanf("%d", &notas[i]);" ou mesmo "scanf("%d", notas[i]);" . Porque? Todas estas expressões são aceitas? Qual a diferença entre elas???

Exercícios

Faça um programa que lê 4 notas para cada um dos 30 alunos de uma turma, armazena-as em uma matriz, calcula e devolve a média de cada aluno.

11 - PONTEIROS

Uma das características mas poderosas oferecidas pela linguagem C é o uso de ponteiros. Mas também é visto como um do tópicos mais complexos da linguagem. Isto ocorre principalmente porque os conceitos embutidos em ponteiros podem ser novos para muitos programadores (tendo-se em vista que nem todas as linguagens utilizam ponteiros), mas também porque os símbolos usados para notação de ponteiros em C não são tão claros; por exemplo, o mesmo símbolo ("*") é usado para duas diferentes finalidades.

Mas, o que são ponteiros?

Ponteiros proporcionam um modo de acesso a variáveis sem referenciá-las diretamente, através do seu endereço. Basicamente, um ponteiro é uma representação simbólica de um endereço.

Para declarar uma variável como ponteiro, a sintaxe é:

tipo * *nome_da_variável;*

onde *tipo* é o tipo de variável para qual o ponteiro vai apontar e *nome_da_variável* é o nome do ponteiro. Ou seja, a variável ponteiro *nome_da_variável* vai conter o endereço de uma outra variável do tipo *tipo*. Além disto, quando quisermos definir o ponteiro, utilizaremos o operador "**&**" para atribuir "**o endereço de**" uma variável ao ponteiro.

Ex:

```
int  * pont_i ;              // pont_i é uma variável ponteiro para variáveis int
    float * pont_f;              //  pont_f é uma variável ponteiro para variáveis float
    int i=5;
    ...
    pont_i = &i;                 // coloca o endereço de i em pont_i
```

A variável *pont_i* agora "aponta para i". Caso precisemos acessar o conteúdo do endereço apontado por pont_i, ou seja, o conteúdo de i, utilizaremos o operador "*", que neste caso significa "**valor no endereço de**":

```
*pont_i = 5;        // coloca o valor 5 no endereço apontado por pont_i  (faz i = 5)
```

Os ponteiros podem ser inicializados já na declaração, como qualquer variável:

```
int *pi = &i;
```

Além disto, os ponteiros podem ser utilizados na maioria das expressões válidas em C.

Ponteiros são usados em situações em que a passagem de valores é difícil ou indesejável. Algumas razões para o uso de ponteiros são:

1 - fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem;

2 - para passar matrizes e *strings* mais convenientemente de uma função para outra;

3 - para manipular matrizes mais facilmente através da movimentação de ponteiros para elas;

4 - para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra;

5 - para comunicar informações sobre memória, como na função *malloc()* que retorna a localização de memória livre através do uso de ponteiro;

6 - a notação de ponteiros compila muito mais rapidamente, tornando o código mais eficiente.

Outra coisa muito importante, é lembrar que podemos passar vários valores como argumentos para uma função, porém apenas sabemos como retornar um único valor (através do comando "return"). Mas como fazer para que uma função altere mais de um valor para a função chamadora? Visto que não há mecanismos próprios para isto, devemos contar com o uso de ponteiros.

Ex:

```
#include <stdio.h>
void main(void)
{     int x=4, y=7;
      int *px, *py;

   printf("x é:   %d,  y é:  %d. \n", x, y);
      px = &x;
   py = &y;
   printf(" O endereço de x é: %u, e o de y é:  %u \n", px, py);
   *px = *px + 10;                    // pode-se usar  *px += 10;
   *py = *py + 10;                   // e também    (*py ) +=10
    printf(" Agora x é:  %d  e y é:  %d \n", x, y);
  printf(" ... e agora px aponta para: %u, e py para: %u\n", ++px, ++py);
  printf(" ... que contém os valores: %d, %d \n", *px, *py);
}
```

Um operador "&" pode ser aplicado somente a variáveis e elementos de matrizes. Construções tais com "&(x+1)" e "&(3)" são ilegais.

Ponteiros são sempre inicializados com o valor 0 (NULL) que não é um endereço válido, obrigando portanto, que inicializemos sempre nossos ponteiros antes de utilizá-los.

11.1 - PONTEIROS E MATRIZES

Existe uma correspondência muito grande entre ponteiros e matrizes. O nome de uma matriz é equivalente ao endereço do primeiro elemento na matriz. Como isto é imutável, podemos chamar o nome de uma matriz de "**ponteiro constante**". Já um ponteiro comum tem conteúdo variável, pode conter qualquer endereço, sendo porisso chamado de "**ponteiro variável**". Pode-se fazer, portanto:

```
int imat[5] = {5,7,31,18,22};
  int *ip;
  ...
  ip = imat;        //   OU  ip=&imat[0];
```

e teríamos, por exemplo, "***(ip+n)**" como sendo **o valor do *n-ésimo* elemento da matriz** imat. Isto porque o significado da adição ou subtração de um inteiro com um ponteiro é adicionar ou subtrair o tamanho de memória do tipo da variável (**sizeof**) para o qual o ponteiro foi declarado. Em outras palavras, ip não vai ser incrementado de n *bytes* de memória, mas de 2 *bytes* * n, pois o tipo *int* ocupa 2 bytes. A disposição na memória, por exemplo, seria:

Variável	Endereço	Valor
imat	567	630
imat[0]	630	5
imat[1]	632	7
imat[2]	634	31
imat[3]	636	18
imat[4]	638	22
ip	802	630

então:

Expressão	Valor	Explicação
ip	630	endereço de imat[0]
*ip	5	valor em 630
ip + 1	632	endereço de imat[1]
*(ip+1)	7	valor em 632
*ip + 1	6	valor em 630 mais 1
imat[0]	5	
*(imat)	5	
imat[1]	7	
*(imat+1)	7	

Com *strings* é a mesma coisa:

```
char  *pc="ABC";
```

define um ponteiro para a série de caracteres "ABC" (terminada com "\0"). O ponteiro pc tem portanto, como valor, o endereço do primeiro elemento da série, o caractere 'A'.

Pode-se ter, também, matrizes de ponteiros:

```
static char *cp[3] = {"XYZ", "QRS", "KLM"};
```

por exemplo, declara cp como sendo uma matriz de 3 ponteiros para caracteres e já inicializa-os, fazendo-os apontar para as três sequências de caracteres.

Na memória isto ficaria:

Variável	Endereço	Valor
constante	731	'X'
constante	732	'Y'
constante	733	'Z'
constante	734	'\0'
constante	735	'Q'
constante	736	'R'
constante	737	'S'
constante	738	'\0'
constante	739	'K'
constante	740	'L'
constante	741	'M'
constante	742	'\0'
cp[0]	900	731
cp[1]	902	735
cp[2]	904	739

11.2 - ARITMÉTICA DE PONTEIROS

Existem poucas operações que devem ou podem ser efetuadas com o valor dos ponteiros. Essas operações são a atribuição de valores a outros ponteiros, soma e subtração de inteiros, e comparação de igualdade com outro valor de ponteiro. Ponteiros podem ser adicionados e subtraídos de outros ponteiros, quando estes ponteiros apontam para elementos diferentes de uma mesma matriz. Por exemplo:

```
static int imat[10];
  int *ip= &imat[0];
  int *iq= &imat[4];
  ...
  distancia = iq - ip;     // distancia= 4 e significa o número de elementos int entre iq e ip
```

Os operadores unários utilizados em ponteiros são "++" e "--". Supondo um ponteiro para ponto flutuante p1, com valor atual de 2000. Após a operação:

```
p1++;
```

o conteúdo de p1 será 2008 e não 2001. Para cada incremento de p1, este apontará para o **double seguinte**, que na maioria dos computadores tem 8 palavras de comprimento.

O mesmo vale para decréscimos:

```
p1--;
```

fará com que p1 tenha o valor 1992.

Pode-se comparar ponteiros, através dos testes relacionais ">=, <=, < e >". Deve-se, no entanto, tomar cuidado para não comparar ponteiros que apontam para tipos diferentes de variáveis, pois os resultados serão sem sentido.

Não se pode multiplicar, dividir, deslocar os bits, somar ou subtrair **floats** e **double** aos ponteiros.

11.3 - PONTEIROS PARA MATRIZES USANDO FUNÇÕES

Vamos analisar como uma função pode usar ponteiros para acessar elementos de uma matriz cujo endereço é passado para a função como argumento.

Como exemplo, vamos ver a função adcon1(), que adiciona uma constante a todos os elementos de uma matriz.

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

void adcon1(int *, int, int);

  void main(void)
  {   static int matriz[TAM] = { 3,6,7,9,11};
      int c=10;
      int j;

      adcon1(matriz, TAM, c);
      for(j=0; j<TAM; j++)
           printf("%d", *(matriz+j));
}

 // adcon1() - adiciona constante a cada elemento da matriz
void adcon1(int *ptr,int num,int con)
{   int k;
    for(k=0; k<num; k++)
    {   *ptr = *ptr + con;   ptr++;  }
}
```

A saída será:

13 16 17 19 21

Na definição da função adcon1(), a declaração " int *ptr; " é equivalente a " int ptr[]; ". Em outras palavras, a primeira declaração cria um ponteiro variável, enquanto a segunda, um ponteiro constante.

Exercícios

1 - "Os ponteiros permitem a passagem de valores por referência para uma função". Demonstre esta propriedade através de um programa.

2 - Faça um programa que calcule a média aritmética de um número arbitrário de notas de provas, usando matrizes.

3 - Repita o programa acima, agora utilizando ponteiros.

Exemplos

```
/*****************************************************************************/
/***** Programa que procura um caractere em uma "string"                 **********/
/*****************************************************************************/
#include <stdio.h>
#include <conio.h>

char *procstr(char *, char);

void main(void)
{
   char *ptr;
   char ch, lin[81];

   puts("Digite uma frase:  ");
   gets(lin);
   printf("Digite o caractere a ser procurado:  ");
   ch = getche();
   ptr = procstr(lin, ch);
   printf("\n A string comeca no endereço %u.\n",lin);
   if(ptr)
   {   printf("Primeira ocorrencia do caractere: %u.\n", ptr);
      printf("E a posição: %d", ptr-lin);
   }
   else     printf("\n caractere nao existe. \n");
   getche();
}

char *procstr(char *linha,char c)
{  while(*linha != c && *linha != '\0')
         linha++;
   if(*linha != '\0')
         return(linha+ 1);
   else
         return(0);
}
```

12 - TIPOS DE DADOS COMPLEXOS E ESTRUTURADOS

A linguagem C permite que o usuário "crie" seus próprios tipos complexos de variáveis. Enumerações, uniões, estruturas e definição de tipos serão os tópicos estudados neste capítulo.

12.1 - ENUMERAÇÕES

Enumerações são classes, conjuntos de valores relacionados, criados para melhorar a legibilidade do código-fonte. Uma variável de um tipo enumeração somente pode receber valores que foram declarados para aquele tipo. A sintaxe da declaração de um tipo **enum** é:

enum *rótulo* {*enum1, enum2, ..., enumn*};

onde *rótulo* é uma identificação para esta enumeração e *enum1, enum2, ..., enumn* são os valores possíveis para *rótulo*.

A sintaxe para a definição de uma variável como sendo de um tipo *rótulo* é:

enum *rótulo nome_variável*;

onde *nome_variável* é o nome da variável que vai assumir algum valor dentre os valores possíveis da enumeração *rótulo*.

Como qualquer tipo em C, a definição de uma variável como sendo de um certo tipo enumeração *rótulo*, pode ser feita na própria declaração do tipo.

enum *rótulo* {*enum1, enum2, ..., enumn*} *nome_variável*;

Ex:

```
enum dias { seg, ter, qua, qui, sex, sab, dom};                    (1)
...
enum dias hoje, dia_semana;                                        (2)
```

ou

```
enum dias { seg, ter, qua, qui, sex, sab, dom} hoje, dia_semana;   (3)
```

Neste exemplo, definimos uma enumeração *dias* (1), que pode ter os valores *seg, ter, qua, qui, sex, sab, dom*. Logo abaixo, (2), declaramos as variáveis *hoje* e *dia_semana*, como sendo do tipo *dias*. Pode-se fazer os dois passos ao mesmo tempo, como na linha (3) do exemplo.

A definição da variável será feita com a seguinte sintaxe:

nome_variável = *enumx*;

onde *enumx* é qualquer um dos n valores contidos na enumeração.

Ex:

dia_semana = ter;

Cada valor possível na enumeração recebe um valor inteiro com o qual pode ser representado. Caso não seja explicitado, o primeiro valor vai receber o inteiro 0, o segundo, o inteiro 1 e assim por diante. Isto é feito para que se possa comparar cada valor da enumeração. Por exemplo, na enumeração *dias*:

Identificador	**Valor**
seg	0
ter	1
quar	2
qui	3
sex	4
sab	5

dom 6

se fizermos:

Expressão	Valor
dia_semana == seg	1 se dia_semana for seg, 0 caso contrário
hoje > sex	1 se hoje for sab ou dom, 0 caso contrário
ter > quar	0 (falso)

Alguns compiladores permitem que se altere o inteiro atríbuido a cada valor da enumeração. Exs:

```
enum estações { primavera = 1, verão = 2, outono = 3, inverno = 4} estação;

enum fim_de_semana { sab = 6, dom );
```

No exemplo acima, dom terá automaticamente, o valor 7.

12.2 - ESTRUTURAS

Estrutura é um grupo de variáveis, cujo formato é definido pelo programador e ao contrário das matrizes, pode ser composto por tipos diferentes. Em outras linguagens, por exemplo Pascal, estruturas são conhecidas como registros.

O exemplo tradicional de uma estrutura é o registro de uma folha de pagamento: um funcionário é descrito por um conjunto de atributos tais como *nome* (uma "string"), o *número do seu departamento* (um inteiro), *salário* (um float) e assim por diante. Como provavelmente existirão vários funcionários, pode-se criar uma matriz desta estrutura como sendo o banco de dados completo de pagamentos.

Definição

A definição de uma estrutura é feita da seguinte forma:

```
struct rótulo
    {  declaração da variável1;
       declaração da variável2;
       ...
       declaração da variáveln;
    };
```

onde *rótulo* é uma identificação para esta estrutura e como veremos mais tarde, é opcional. As linhas de comando *declaração da variável* são declarações de variáveis de tipos convencionais (int, float, char) que vão compor os **campos** da estrutura.

Declaração de uma variável do tipo estrutura:

A declaração de uma variável como sendo do tipo estrutura *rótulo* é feita com a seguinte sintaxe:

```
struct rótulo nome_variável;
```

O exemplo acima declara a variável *nome_variável* como sendo uma estrutura do tipo *rótulo*. A declaração da variável pode ser feita já na definição do tipo estrutura:

```
struct rótulo
    {  declaração da variável1;
       declaração da variável2;
       ...
       declaração da variáveln;
    }  nome_variável;
```

como usualmente é feito.

Ex:

```
struct registro
{       char nome[20];
        int departamento;
        float salário;
};
struct registro meu_registro;
...
struct registro folha_pagamento[50];
```

Neste exemplo, definimos a estrutura *registro* que contém dados sobre um determinado empregado, como *nome*, número de *departamento* e *salário*. Declaramos também, a variável *meu_registro* como sendo do tipo *registro*, ou seja, uma estrutura de dados.

Matriz de estruturas:

A seguir, no exemplo acima, declaramos a **matriz de estruturas** *folha_pagamento*, como tendo 50 elementos. Isto significa que a folha de pagamento da empresa conterá 50 registros, cada qual com o nome, número de departamento e salário do empregado.

A sintaxe para a declaração de uma matriz de estruturas é:

struct *rótulo nome_mat_estrut*[*dimensão*];

onde *nome_mat_estrut* é o nome da matriz de estruturas e *dimensão* é o número de elementos da matriz.

Rótulo da estrutura:

O *rótulo* de um tipo estrutura é opcional quando declaramos uma estrutura na própria definição do seu tipo. Caso façamos a declaração da variável em linha de comando subsequente ou queiramos declarar outras estruturas como sendo daquele tipo deve-se utilizar o *rótulo*.

Inicialização da estrutura:

A inicialização de estruturas assemelha-se à inicialização de matrizes:

```
static struct registro meu_registro = { "Fernanda Marques", 21, 10000};
```

ou

```
static struct registro meu_registro = { { 'F', 'e', 'r', 'n', 'a', 'n', 'd', 'a' , ' ', 'M', 'a', 'r', 'q', 'u', 'e', 's', '\0'}, 21, 10000};
```

ou

```
static struct registro meu_registro = {  'F', 'e', 'r', 'n', 'a', 'n', 'd', 'a' , ' ', 'M', 'a', 'r', 'q', 'u', 'e', 's', '\0', 0, 0, 0, 21, 10000};
```

Na memória, os campos de uma estrutura são armazenados um ao lado do outro. Portanto, o endereço da estrutura é o endereço do **primeiro byte do primeiro campo** desta. Os 3 zeros (entre o caractere nulo, '\0', e o valor do segundo campo, 21) na inicialização da estrutura *meu_registro* acima, foram justamente acrescentados para completar o espaço de memória alocado na declaração do campo *nome* (string de 20 bytes).

Acessando membros da estrutura:

Para acessar individualmente cada campo de uma estrutura, utilizamos o operador de seleção "**.**". A sintaxe é:

```
estrutura.campo = valor;
```

onde *campo* é cada variável declarada na estrutura *estrutura*.
Exs:

```
meu_registro.nome =  "Fernanda Marques";
meu_registro.nome[0] = 'F';
```

Atribuições entre estruturas:

Na maioria dos compiladores mais modernos é possível igualar-se duas estruturas **do mesmo tipo** da seguinte forma:

```
estrutura1 = estrutura2;
```

Note que não foi preciso igualar cada campo individualmente!

Endereço da estrutura:

A sintaxe do endereço de uma estrutura, como um todo, é:

&nome_estrutura

Passando e devolvendo estruturas para funções:

Para passar (por valor) uma estrutura como argumento para uma função simplesmente passamos o nome da estrutura:

```
...
struct xyz {                  /*   tipo estrutura xyz, com dois campos: a (inteiro) e b (caractere)  */
   int a;
   char b;
} estr1, estr3(struct xyz);         /* declaração da  variável global estr1 e função estr3() como tipo xyz */
...
void main(void)
{   struct  xyz estr2;       /*  declaração da variável local estr2 como tipo xyz  */
   ...
   estr1.a = 234;             /*   atribuição ao campo a da variável estr1
   estr1.b = 'J';             /*   atribuição ao campo b da variável estr1
   estr2 = estr3(estr1);      /* atribuição do valor retornado pela função estr3 à variável estr2...
   ...                           /* ... (ambos os campos). O argumento é uma cópia de estr1
}
struct xyz estr3(struct xyz estr) /* função estr3 - recebe (estr) e retorna (estrlocal) vars xyz */
{   struct xyz estrlocal;              /*  declaração da variável local estrlocal  */
    estrlocal.a = estr.a + 1;          /*  atribuição ao campo a da variável estrlocal  */
    estrlocal.b = estr.b + 2;          /*   atribuição ao campo b da variável estrlocal  */
     ...
    return(estrlocal);                 /*   retorna valor atualizado de estrlocal   */
}
```

O algoritmo acima não é trivial e devemos fazer algumas considerações sobre o mesmo:

1) Note que o tipo estrutura *xyz* foi definido como global;
2) A variável global *estr1* e a função *estr3()* foram definidas como sendo do tipo *xyz*;
3) Dentro da função *main()* foi declarada uma variável local, *str2*, do tipo *xyz*;
4) Os campos *a* e *b* da variável estrutura *estr1* recebem valores em *main()*;
5) A variável *estr2* vai receber o valor retornado pela função *estr3*(), que por sua vez recebeu o valor de *estr1* como argumento;
6) Na declaração da função, deve-se colocar "*struct xyz estr3(argumentos)"*, para que o programa saiba que o tipo retornado pela função também é uma estrutura;

7) Também na lista de argumentos, "*(struct xyz estr)*", deve-se especificar que o tipo recebido pela função é uma estrutura *xyz*;

8) Dentro da função *estr3()* foi declarada uma variável local, *estrlocal*, do tipo *xyz*;

9) Os campos *a* e *b* da variável estrutura *estrlocal* recebem valores (no caso, o inteiro 235 e o caractere 'L') em *estr3()*;

10) No final, a função *estr3()* retorna a estrutura *estrlocal* para a expressão chamadora.

Estruturas aninhadas:

Assim como podemos ter matrizes de matrizes (várias dimensões), podemos também ter estruturas que contém outras estruturas.

Ex:

```
/*****************************************************************************/
/* Programa para biblioteca que separa livros em dois grupos: Dicionario e Literatura */
/*              mostra o uso de estruturas aninhadas                         */
/*****************************************************************************/
 ...
 struct livro {
      char titulo[30];
     int regnum;
 };
 struct grupo {
     struct livro dicionario;
     struct livro literatura;
 };
 struct grupo grupo1 =  {  {"Aurélio", 134},
                                  {"Iracema", 321}     };

 void main(void)
 {
          printf("\nDicionario: \n");
         printf("  Titulo: %s \n", grupo1.dicionario.titulo);
         printf("   No. do registro: %03d\n", grupo1.dicionario.regnum);
         printf("   \nLiteratura: \n");
         printf("   Titulo: %s\n", grupo1.literatura.titulo);
         printf("    No. do registro: %03d\n", grupo1.literatura.regnum);
 }
```

Considerações:

1) Quando uma matriz de várias dimensões é inicializada, usamos chaves dentro de chaves; do mesmo modo inicializamos estruturas dentro de estruturas;

2) Para acessar um elemento da estrutura que é parte de outra estrutura utilizamos:

grupo.dicionario.titulo

No exemplo acima temos o elemento *titulo* da estrutura *dicionario,* que por sua vez é um elemento da estrutura *grupo*;

Para compor um banco de dados completo, vamos utilizar tudo o que aprendemos até agora: matrizes de estruturas, passagem de estruturas para funções, estruturas aninhadas, etc. Seguindo o exemplo da livraria, temos:

```
/*****************************************************************************/
/***      PROGRAMA LIVRARIA - Demonstra como criar bancos de dados em C, utilizando */
/***     estruturas        ********************************************************/
#include <stdlib.h>                    /* para a função atof(), que transforma uma string em float */
#include <stdio.h>
```

```
#include <conio.h>
void novonome(void);
void listatot(void);

struct lista
{      char titulo[30];
       char autor[30];
       int regnum;
       double preco;
};

struct lista livro[50];                    /* declara uma matriz com 50 livros (estruturas do tipo lista)  */

int n=0;
void main (void)
{      char ch;

               for(;ch != 's';)
               {    printf("\nDigite: 'e' para adicionar um livro\n");
                        printf("\t'l ' para listar os livros\n");
                     printf("\t's' para sair:\n");
                        ch = getche();
                     printf("\n");
                   switch(ch)
                    {      case 'e' :  novonome();   break;
                          case 'l' :  listatot();  break;
                             case 's' :  printf("Fim do programa");  break;
                          default :  puts("\nDigite somente opções válidas: ");
                    }
               }
}

/***** FUNÇÃO NOVONOME   - adiciona um novo livro ao arquivo    ***/
void novonome(void)
{       char numstr[81];
        printf("\n Registro   %d. \nDigite titulo:   ", n+1);
                gets (livro[n].titulo);
                printf("Digite autor:  ");
                gets(livro[n].autor);
                printf("Digite o número do livro (3 digitos):  ");
                gets(numstr);
                livro[n].regnum = atoi(numstr);             /* função atoi() converte uma string para inteiro */
                printf("Digite preço:");
                gets(numstr);
                livro[n++].preco = atof(numstr);
                printf("\n_____\n");
}
/***** FUNÇÃO LISTATOT - lista os dados de todos os livros      *****/
void listatot(void)
{
        int i;
        if( !n)
            printf("\n\nLista vazia!!\n");
        else
            for( i=0; i<n; i++)
            {   printf("\n Registro %d.\n", i+1);
                        printf("Titulo: %s. \n", livro[i].titulo);
                printf("Autor: %s.\n", livro[i].autor);
```

```
        printf("Numero do registro: %3d.", livro[i].regnum);
        printf("Preço: %4.2f. \n\n", livro[i].preco);
         }
printf("\n_____\n");
}
```

Note que para acessarmos cada elemento da matriz de estruturas utilizamos "*livro[2].titulo*", por exemplo. Isto mostra que o índice da matriz é atribuído à *livro* e não ao campo, *título*. Se tivéssemos a construção "*livro[2].titulo[3]*", por exemplo, estaríamos referindo-nos ao quarto elemento (caractere) da *string titulo*, da terceira estrutura de livros.

Campos de bits

Uma estrutura pode conter **campos de bits**, em vez de bytes. Estes campos são normalmente utilizados para acessar valores dependentes da máquina (como registradores, por exemplo). Campos de bits são uma alternativa à utilização dos operadores bit a bit no acesso individual de bits dentro de um inteiro.

A sintaxe é:

```
struct{
        unsigned int var1: num_bits1;
        unsigned int var2: num_bits2;
        ...
        unsigned int varn: num_bitsn;
    } var_estrut;
```

onde *unsigned int* é o tipo de todos os campos da estrutura; *var1, var2,..., varn* são os campos da estrutura; *num_bits* é o número de bits que cada campo tem e *var_estrut* é a variável declarada como estrutura de campos de bits. Um campo não pode ser ultrapassar o tamanho de um **int**. Se isto ocorrer, o campo será alinhado no próximo inteiro.

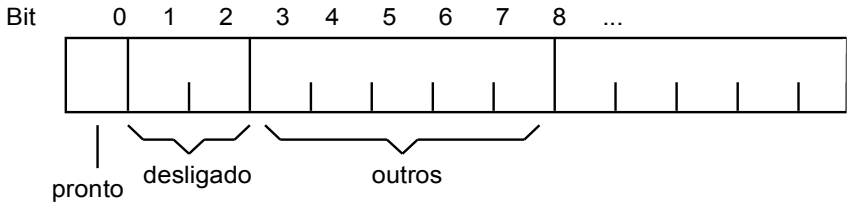
Um exemplo de uma declaração para campos de bits é:

```
struct {
    unsigned int pronto: 1;
     unsigned int desligado: 2;
     unsigned int outros: 5;
      } modem;
```

Isto declara modem como sendo uma estrutura de 8 bits com os membros:

Membro	**Referência**
modem.pronto	primeiro bit
modem.desligado	próximos dois bits
modem.outros	próximos cinco bits

Estas expressões podem ser utilizadas em qualquer lugar onde um inteiro unsigned possa ser utilizado. **modem** terá a distribuição de memória mostrada no diagrama abaixo (assumindo que ints são obrigatoriamente alinhados em bytes pares e que bits, em nossa máquina-exemplo, são atribuídos da esquerda para a direita, já que isto depende do computador):



O operador de endereço (&) não pode ser utilizado com campos.

Ponteiros para Estruturas:

Existem diversos motivos para se usar ponteiros para estruturas. Assim como ponteiros são mais rápidos e fáceis de manipular que matrizes, também os ponteiros para estruturas são melhores que matrizes de estruturas. Várias representações de dados que parecem fantásticas são constituídas de estruturas contendo ponteiros para outras estruturas.

A sintaxe da declaração de um ponteiro para estrutura é:

struct *rótulo* **nome_ponteiro*;

onde rótulo é o identificador para o tipo de estrutura e nome_ponteiro é o nome do ponteiro que aponta para tipos estrutura. Na verdade, a sintaxe é a mesma de qualquer declaração de ponteiro.

Ex:

struct lista *ptrl;

declara o ponteiro *ptrl* que pode apontar para qualquer estrutura do tipo *lista*.

A sintaxe da definição de um ponteiro para estrutura é:

nome_ponteiro = &*estrutura*;

onde estrutura é uma *estrutura* qualquer do tipo que o ponteiro *nome_ponteiro* pode apontar.

Ex:

ptrl = &livro[0];

define o conteúdo de *ptrl* como sendo o endereço do primeiro elemento da matriz de estruturas *livro*.

Sabemos que se conhecemos o nome de uma dada variável estrutura, podemos acessar seus campos usando seu nome acompanhado do operador ponto.

Para acessar os campos de uma estrutura através do ponteiro não teria sentido utilizar-se a construção "*ptrl.preco*", por exemplo, porque *ptrl* é um ponteiro e não um nome de estrutura.

Temos duas formas de acessar os campos:

1) Utilizando a construção:

(*ptrl).preco

que indica "o campo *preco* da estrutura apontada por *ptrl* ". Isto é equivalente a "*livro[0].preco*", se *ptrl == & livro[0]* !

ou

2) Utilizando o operador "->" (sinal de "menos" seguido pelo sinal "maior que"):

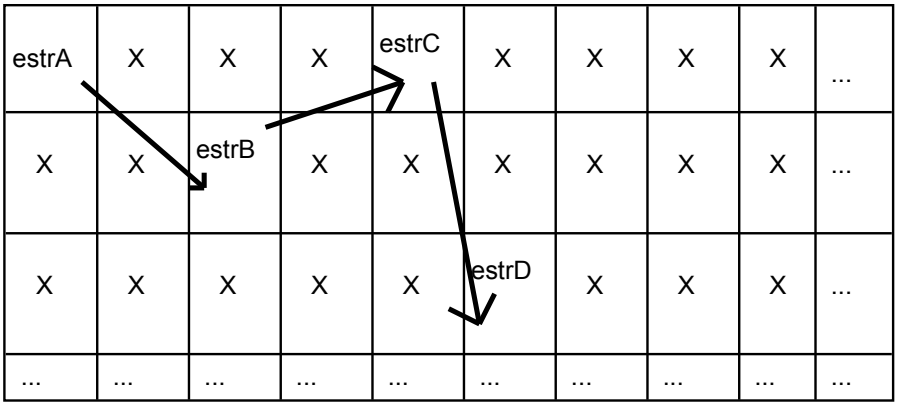
ptrl -> preco

Este é o método mais comum e tem o mesmo significado da primeira opção: *ptrl* é um ponteiro para estrutura, mas "*ptrl -> preco*" é uma variável double.

12.3 - LISTAS ENCADEADAS

A lista encadeada ou "lista ligada" é uma estrutura de dados abstrata que pode ser criada em C, utilizando-se o mecanismo de ponteiros para estruturas. Uma lista encadeada assemelha-se a uma corrente em que as estruturas estão penduradas sequencialmente. Isto é, a corrente é acessada através de um ponteiro para a primeira estrutura, chamado "cabeça", e cada estrutura contém um ponteiro para a sua sucessora. O ponteiro da última estrutura tem valor NULL ("\0"), indicando o fim da lista.

Normalmente uma lista encadeada é criada dinamicamente na memória. O diagrama abaixo ilustra a grosso modo como ficaria uma lista encadeada na memória:



onde cada flecha é um ponteiro apontando para a próxima estrutura da lista. Esta "flecha" é um campo da estrutura, criado como tipo ponteiro, que conterá o endereço da estrutura seguinte.

Abaixo temos o exemplo "Livraria", utilizando-se listas ligadas, ao invés de matrizes de estruturas:

```
/*****************************************************************************/
/***      PROGRAMA LIVRARIA 2 - Demonstra como criar bancos de dados em C, utilizando   */
/***     estruturas  e listas ligadas                                                    */
/*****************************************************************************/
#include <stdlib.h>                     /* para a função atof(), que transforma uma string em float */
#include <stdio.h>
#include <conio.h>
#define TRUE 1

void novonome(void);
void listatot(void);

struct prs                              /*  estrutura básica para listas ligadas    */
{     char titulo[30];
      char autor[30];
      int regnum;
      double preco;
      struct prs *ptrprox;
};

struct prs *ptrprim, *ptratual, *ptrnovo;       /* declara estruturas tipo prs para montar a lista encadeada   */

void main (void)
{     char ch;

            ptrprim = (struct prs *) NULL;        /*  sem dados ainda   */
             for(;ch != 's';)
             {    printf("\nDigite: 'e'  para adicionar um livro\n");
                  printf("\t'l '  para listar os livros\n");
                   printf("\t's'  para sair:\n");
                  ch = getche();
                  printf("\n");
                 switch(ch)
                 {      case 'e' :  novonome();   break;
                     case 'l' :  listatot();  break;
                        case 's' :  printf("Fim do programa");  break;
                     default :  puts("\nDigite somente opções válidas: ");
                 }
            }
```

```
}

/***** FUNÇÃO NOVONOME   - adiciona um novo livro ao arquivo    ***/
void novonome(void)
{       char numstr[81];

                ptrnovo = (struct prs *) malloc (sizeof(struct prs));      /* novo ponteiro p/  estrut. da lista  */
 /* reserva espaço na memória (malloc)  para armazenar estrutura do tamanho (sizeof)  de prs   */
                if (ptrprim == (struct prs *) NULL)         /* se o 1o. ponteiro da lista tem conteudo NULL */
                        ptrprim = ptratual = ptrnovo;       /* lista ainda não tem dados */
                else
                {       ptratual = ptrprim;
                        while(ptratual -> ptrprox != (struct prs *) NULL)         /* procura novo item  */
                              ptratual = ptratual -> ptrprox;
                      ptratual -> ptrprox = ptrnovo;
                      ptratual = ptrnovo;
                }

               printf(" Digite titulo:   ");
               gets (ptratual -> titulo);
               printf(" Digite autor:  ");
               gets(ptratual -> autor);
               printf(" Digite o número do livro (3 digitos):  ");
               gets(numstr);
               ptratual -> regnum = atoi(numstr);            /* função atoi() converte uma string para inteiro */
               printf(" Digite preço:");
               gets(numstr);
               ptratual -> preco = atof(numstr);
               printf("\n_____\n");
                ptratual -> ptrprox = (struct prs *) NULL;                        /* último       */
}

/***** FUNÇÃO LISTATOT - lista os dados de todos os livros      *****/
void listatot(void)
{
        if(ptrprim == (struct prs *) NULL)
        {     printf("\n\nLista vazia!!\n");
              return;
        }
        ptratual = ptrprim;
        do
        {      printf("Titulo: %s. \n", ptratual -> titulo);
               printf("Autor: %s.\n", ptratual -> autor);
               printf("Numero do registro: %3d.", ptratual -> regnum);
               printf("Preço: %4.2f. \n\n", ptratual -> preco);
              ptratual = ptratual -> ptrprox;
        } while(ptratual != (struct prs *) NULL);

        printf("\n_____\n");
   }
```

A grande vantagem neste último programa "Livraria", que utiliza listas ligadas, é que a memória necessária para armazenar as estruturas é alocada à medida que a lista aumenta. No primeiro programa "Livraria", foi necessário alocar 50 espaços de memória, cada um do tamanho de uma estrutura, o que representa um enorme disperdício se entramos com apenas 3 registros, por exemplo, ou que pode ser insuficiente (e tomar espaços indevidos na memória), se um desavisado resolve entrar com 100 registros!

A função malloc()

Note que no programa acima utilizamos a função **malloc()**, para alocar memória e armazenar as estruturas criadas dinamicamente. Isto significa que só alocamos memória, quando for realmente necessário! A função malloc() toma como argumento um inteiro sem sinal que representa a quantidade de bytes de memória requerida. A função retorna um ponteiro para o primeiro byte do bloco de memória disponível que foi alocado. Se não houver memória suficiente para alocar, malloc() devolverá um ponteiro NULL.

A função sizeof()

O operador unário sizeof() devolve o tamanho, em bytes, do argumento que recebe.
Utilizamos no programa acima o operador sizeof() para determinar o tamanho da estrutura *prs* e poder alocar memória para armazená-la.

12.4 - UNIÕES

Uniões são localizações de memória usadas para agrupar um número de variáveis de tipos diferentes juntas, tal como as estruturas. Porém, enquanto os membros (campos) de uma estrutura são armazenados em espaços diferentes de memória, numa união os membros compartilham da mesma localização de memória. Ou seja, a união é uma forma de tratamento de uma área de memória contendo um tipo de variável numa ocasião e um outro tipo de variável noutra ocasião.
A sintaxe de definição e a de uso de uma união é a mesma que a de uma estrutura:

```
union rótulo
{    declaração1;
     declaração2;
     ...
     declaraçãon;
} var_união;
```

onde *rótulo* é uma identificação para a união e *declaraçãox* é uma declaração de variável, um membro da união. A variável *var_união* é declarada, na sintaxe acima, como sendo uma união do tipo *rótulo*. O tamanho de uma união será o tamanho do maior de seus membros.
Ex:

```
union demo
{   char inicial;
    int idade;
    float salario;
}
...
union demo pessoal;
```

declara uma variável de nome *pessoal* do tipo *union demo* e para esta variável foram reservados 4 bytes de memória tendo em vista que o maior de seus membros é do tipo float.
Para acessar um membro da união, utilizamos o operador ponto (.), como em estruturas:

```
var_união.membro
```

onde membro é qualquer membro declarado na união var_união.
Ex:

```
"       pessoal.idade
```

Uma das razões para se utilizar uniões é a possibilidade de se usar um único nome para dados de tipos diferentes. Por exemplo, se queremos utilizar um mesmo dado em funções diferentes, que aceitam argumentos de tipos diferentes, declaramos este dado como união dos dois tipos passados como argumentos para as funções.

APÊNDICE A - ROTINAS DE ENTRADA E SAÍDA (I/O)

Geralmente os programas desenvolvidos têm alguma forma de entrada e saída de dados. Por exemplo, num programa que calcula o fatorial de um número fornecido pelo usuário, o dado de entrada seria este número e a saída, o fatorial deste.

As funções de entrada e saída mais utilizadas são, respectivamente, **scanf()** e **printf()**. Neste apêndice veremos com detalhes a utilização destas e outras funções de I/O importantes da linguagem C.

Obs: Como cada compilador possui conjuntos de comandos de I/0 diferentes, convém procurar-se no Help de cada um quais as funções mais convenientes para cada caso. No Turbo C, deve-se teclar no menu "*Help"* (ou digital Alt-h), depois no submenu "*Index"* (ou Alt-i) e por último no botão "*Search"* (ou Alt-s). No quadro de texto deve-se digitar *scanf*, *printf*, *gets*, *getchar*, etc, que a busca será feita automaticamente. Quando o tópico procurado for encontrado, clica-se em "*Show Topics* " e finalmente "*Goto*".

A função printf()

A função printf(), assim como scanf(), é uma função **formatada** de I/O, no caso, entrada de dados. A sintaxe é:

printf("*expressão_de_controle*", *lista_de_argumentos*);

A expressão de controle consiste de dois tipos de itens. O primeiro tipo será feito de caracteres que serão impressos na tela. O segundo tipo contém comandos de formatação que definem a maneira como os argumentos subsequentes são apresentados. Deve existir o mesmo número de comandos de formatação que o número de argumentos e os comandos de formatação e os argumentos são combinados em ordem. Por exemplo:

```
printf("Oi, %c %d %s \n", 'c', 10, "alunos!");
```

apresenta na tela:

```
Oi, c 10 alunos!
```

Os códigos de formatação da função printf() (que valem também para scanf()) são:

Código	Função
%c	um único caractere
%d	um inteiro decimal
%e	um número em notação científica
%f	um número em ponto flutuante
%o	um inteiro octal
%s	uma série de caracteres (string)
%x	um número hexadecimal
%u	um decimal sem sinal
%l	um inteiro longo

Obs1: Os códigos de controle poderão ter modificadores que especifiquem a largura do campo, o número de casas decimais e um indicador de alinhamento à esquerda. Um inteiro colocado entre o sinal "%" e o comando de formatação atua como um *especificador de largura-de-campo mínimo*, o que preenche a saída com brancos ou zeros para assegurar que tenha ao menos um comprimento mínimo. Caso uma série ou um número sejam maiores que o mínimo, serão completamente impressos. O preenchimento normal é feito com espaços. Caso queira-se preencher com zeros, basta colocar um zero antes do especificador de largura-de-campo. Por exemplo, "%05d" vai preencher um número com menos de 5 dígitos com 0's.

Obs2: Para especificar o número de casas decimais impressas para um número em ponto flutuante, coloque um ponto decimal seguido do número de casas decimais que se quer apresentar. Por exemplo, "%10.4f" apresenta um número com pelo menos 10 caracteres de comprimento, com 4 casas decimais. No caso de strings, "%5.7s" apresenta uma série com pelo menos cinco caracteres, mas não excedendo sete (o resto é truncado).

Obs3: Como condição normal, toda saída é alinhada pela *direita* do campo. Para forçar-se o alinhamento pela esquerda, coloca-se um sinal de menos ("-") logo após o "%". Por exemplo, "%-10.2f" forçará um alinhamento à esquerda de um número em ponto flutuante, com duas casas decimais e um campo de 10 caracteres de largura.

Exemplos de formatação da saída printf()

Declaração print()	**saída**
("%-5.2f", 123.234)	|123.23 |
("%5.2f", 123.234)	| 123.23|
("%10s", "hello")	| hello|
("%-10s", "hello")	|hello |
("%5.7s", "123456789")	| 1234567|
("%010d", 1234)	|0000001234|

A função scanf()

A função scanf() realiza a operação de **formatação** nas entradas a partir da entrada padrão (teclado). Existem inúmeras funções da família scanf() que permitem a leitura de dados formatados de outras entradas. Por exemplo, fscanf() lê os dados em um arquivo, e não do teclado. A sintaxe é:

scanf("*expressão_de_controle*", *lista_de_argumentos*)

A expressão_de_controle contém códigos de formatação, precedidos por um sinal "%", indicando qual o tipo de dado será lido. Se após o sinal "%" colocarmos um sinal "*" indica que o dado será lido mas não será atribuído a variável nenhuma, neste caso a lista de argumentos não existe. Isto é útil quando se tem um conjunto de entradas mas somente uma parte da entrada será lida pelo programa.

A lista de argumentos contém um endereço de variável para cada dado lido e formatado pela expressão_de_controle. Se a variável é simples, e não um nome de matriz, ou um ponteiro (que já tem endereços associados ao seu próprio conteúdo), é necessário acrescentar-se o operador "&" ("endereço de"), para especificarmos o endereço de cada variável.

Por exemplo:

```
scanf("%f", &anos);
```

indica que será lido um dado do tipo *float* e atribuído à variável *anos*.

Ex2:

```
scanf("%3d/%2d",&int_arg1, &int_arg2);
```

Neste caso, se for digitado "345/67" e *Enter*, *int_arg1* receberá o valor 345 e *int_arg2* o valor 67.

Obs1: O *modificador de máxima largura de campo* pode ser aplicado aos códigos de formatação. Por exemplo, se deseja-se ler um máximo de 20 caracteres para a string *address*, escreve-se:

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

Obs2: Espaços, tabulações e caracteres de linha-nova apenas são utilizados como separadores de campo, quando não se estiver lendo caracteres simples. Neste caso, serão lidos como caracteres mesmo, e não como separadores. Por exemplo, com uma lista de entrada "x y":

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

voltará com o caractere "x" em *a*, um espaço em *b* e o caractere "y" em *c*.

Obs3: Qualquer outro caractere na expressão de controle (inclusive espaços, tabulações e caracteres de linha-nova) será usado para comparar caracteres da lista de entrada. Caso sejam iguais, serão descartados. Por exemplo, dada a lista de entrada "abcdttttttefg":

```
scanf("%stttttt%s", &name1, &name2);
```

colocará os caracteres "abcd" em *name1* e os caracteres "efg" em *name2*.

Entradas e Saídas Não-Formatadas:

As funções getchar() e putchar()

São as funções mais simples de I/O em C. A função getchar() lê um caractere da entrada padrão (normalmente o teclado). Quando o programa encontra esta instrução, a execução pára e aguarda até o pressionamento de uma tecla, para em seguida devolver seu valor. Normalmente getchar() ecoa o caractere digitado para o vídeo. A sintaxe é:

```
getchar()
```

Já a função putchar() escreve um argumento de caractere no vídeo do computador, se o argumento for parte do conjunto de caracteres que o computador pode apresentar. A sintaxe é:

```
putchar()
```

O exemplo abaixo ilustra a utilização das duas funções, lendo caracteres pelo teclado e imprimindo-os em modo reverso: maiúsculas em minúsculas e vice-versa. O programa pára ao se digitar um ponto:

```
#include <stdio.h>
                                                        #include<ctype.h>
 void main(void)
  {   char ch;
      do
      {     ch = getchar();
            if( islower(ch))  putchar( toupper(ch));
            else putchar(tolower(ch));
      } while(ch != '.');       /* use ponto para parar */
  }
```

As funções gets() e puts()

Estas funções permitem que se leia e escreva séries de caracteres (strings).

A função gets() retorna uma série terminada por nulo em seu argumento de vetor de caracteres ("*nome_string*"). Isto significa que quando utiliza-se gets() pode-se digitar caracteres no teclado até a operação de retorno de carro (Enter). O retorno de carro coloca um terminado nulo no fim da série e gets() retorna. A sintaxe é:

```
gets(nome_string)
```

A função puts() escreve uma série no vídeo. Esta função reconhece os mesmos códigos de *sequências de escape* ("\n", "\t", etc) que printf(). Apesar de ser pouco utilizado por não permitir formatação da saída, conversões de formato, nem colocação de números, a função puts() é muito mais rápida e simples que printf(), quando trata-se de apenas imprimir strings simples. A sintaxe é:

```
puts(string)
```

onde *string* pode ser uma sequencia de caracteres propriamente dita (delimitada por ""), ou o nome de uma.

Ex:

```
  #include <stdio.h>
   #include <stdlib.h>
   #define is_digit(x)  ((x >= '0'  && x <= '9') ?  1:0 )

   number(char *s)

   {    int t;
       for(t = 0; s[t]; ++t)
```

```
            if (!is_digit(s[t]))
                return 0;
        return 1;
 }

 getnum()
 {    char num[80], n;
      do
      {
          gets(num);
          if(!number(num))
          {         puts("Deve ser numero.\n");
                    n = 0;
          } else n=1;
      } while(!n);
   return(atoi(num));
 }

 void main(void)
{     puts("Entre com um numero\n");
      printf("%d",getnum());
}
```

APÊNDICE B - PROGRAMA-EXEMPLO PARA ROTINAS GRÁFICAS

```
/*********************************************************/
//  TWINDOW.C - Teste para as funcoes de janela no modo texto
/*********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>

#define H_LINE '\xC4'    /* ASCII estendido da IBM */
#define V_LINE '\xB3'
#define DH_LINE '\xCD'
#define DV_LINE '\xBA'

#define TLC  '\xDA'   /* caractere para canto superior esquerdo, etc */
#define TRC  '\xBF'
#define BLC  '\xC0'
#define BRC  '\xD9'
#define DTLC '\xC9'
#define DTRC '\xBB'
#define DBLC '\xC8'
#define DBRC '\xBC'

#define SINGLE_BORDER 1
#define TWIN_BORDER   2
#define UNDERLINE  textcolor(YELLOW); textbackground(BLACK);
/* alternativa para underline() */

   struct text_info mytext; /* global: fornece o status da janela,atributos, etc */

// as funcoes a seguir mostram como os atributos para texto interpretam aqueles usados para a selecao //de cor
void magenta(void)
{   textcolor(MAGENTA);
    textbackground(BLACK); }
void green(void)
{   textcolor(GREEN);
    textbackground(BLACK); }
void underline(void)
{   textcolor(BLUE);
    textbackground(BLACK); }
void reversevideo(void)
{   textcolor(BLACK);
    textbackground(WHITE); }
void hide(void)
{   textcolor(BLACK);
    textbackground(BLACK); }

/** Somente para modo texto   **********/
/* MY_HLINE() desenha uma linha horizontal com o caractere se
lecionado de (startx, starty) a (endx, starty). Retorna com
 o numero de caracteres desenhados. startx pode ser maior
 que endx. O cursor ficará na posicao imediatamente seguinte
```

```
ao ultimo caractere exibido. As coordenadas sao relativas
a janela ativa.    */

int my_hline(int startx, int starty, int endx, char line_char)
{
    int i;
    gotoxy(startx, starty);
    if(startx == endx) return (0);
    if(startx < endx)
    {   for ( i = startx; i <= endx; i++)
     putch(line_char);
    return(i - startx);
    }
    gotoxy(endx, starty);
    for( i = endx; i <= startx; i++)
     putch(line_char);
    return(i- endx);
}

/**** Somente para modo texto   *********/
/* MY_VLINE() desenha uma linha vertical de (startx, starty) a
(startx, endy). starty pode ser maior que endy. line_cahr é o
simbolo selecionado para a linha. As coordenadas são relativas
a janela ativa  */

int my_vline(int startx, int starty, int endy, char line_char)
{
    int i;
// nao e' necessario o gotoxy aqui
    if(starty == endy)  return (0);
    if(starty < endy)
    {     for(i=starty; i<=endy; i++)
     {        gotoxy(startx, i);
              putch(line_char);
     }
     return (i-starty);
    }
    for( i = endy; i <= starty; i++)
    {   gotoxy(startx, i);
    putch(line_char);
    }
    return (endy-i);
}



/**** Somente para o modo texto   ******/
/** MY_RECT() desenha um retangulo com coordenadas (tlx, tly)
para o canto superior esquerdo e (brx, bry) para o canto
inferior direito. style=1 fornece uma borda simples, style =2
fornece uma borda dupla   */

int my_rect(int tlx, int tly, int brx, int bry, int style)
{
    int w, h;
    char hline_ch, vline_ch, tlc, trc, brc, blc;

    switch(style)
    {   case 1:
```

```
    case 0:
            hline_ch = H_LINE;
            vline_ch = V_LINE;
            tlc = TLC; trc = TRC;
            brc = BRC; blc = BLC;
            break;
    case 2:
            hline_ch = DH_LINE;
            vline_ch = DV_LINE;
            tlc = DTLC; trc = DTRC;
            brc = DBRC; blc = DBLC;
            break;
    default:
            return(0);
    }
    gotoxy(tlx, tly);
    putch(tlc);
    w = my_hline(tlx+1, tly, brx-1, hline_ch);
    putch(trc);
    h = my_vline(brx, tly+1, bry-1,vline_ch);
    gotoxy(brx, bry);
    putch(brc);
    my_hline(brx-1,bry, tlx +1, hline_ch);
    gotoxy(tlx, bry);
    putch(blc);
    my_vline(tlx, bry-1, tly+1, vline_ch);
    return(w*h);        /* area delimitada  */
}


 void main(void)
{    int graphmode;
    int graphdriver;
    int himode, lomode;
    char savewin1[300], savewin2[300];

    directvideo = 1;
// = 0 significa que usa as chamadas a ROM BIOS; = 1 usa o
// acesso direto ao video

/******    detectgraph(&graphdriver, &graphmode);*****/
// verifica o hardware do video -- encontra o modo de maior
// resolucao


  //   getmoderange(graphdriver, &lomode, &himode);
    textmode(C4350);
// ajusta para colorido VGA em 80x25 - somente texto - sem
// grafico

    gettextinfo(&mytext);
// pega o modo texto atual e a posicao da janela
    clrscr();
    magenta();
    my_rect(9,7,49,25,SINGLE_BORDER);

    green();
    my_rect(10,8,48,13,TWIN_BORDER);
    window(11,9,47,12);
```

```
    gotoxy(2,1);    /* coordenadas relativas  */
    normvideo();
    // reversevideo();
    cprintf("driver = %d,maior modo = %d", graphdriver, graphmode);
    gettext(12,9,47,11, savewin1);
// salva a janela de texto no array savewin1
    normvideo();

    window(1,1,80,25);
    my_rect(10,14,48,19,TWIN_BORDER);
    window(11,15,47,18);

    gotoxy(2,1);   /* =12,15 em coordenadas absolutas  */
// underline() ou MACRO
    UNDERLINE
    cprintf("TxtModo =%d, Winleft =%d, Wintop =%d",mytext.currmode,
mytext.winleft, mytext.wintop);
    gotoxy(2,2);   /* = 12,16 em coordenadas absolutas */
    normvideo();
    cprintf("Modo baixo= %d, Modo alto = %d", lomode, himode);
    gettext(12,15,47,17, savewin2);
// salva uma determinada area de texto no array savewin2
    underline();
    highvideo();
    cputs("<cr>."); getch();

// pausa de tempo - ate' pressionar uma tecla
    gotoxy(1,1);
    clrscr();     // limpa a janela de texto 2
    puttext(12,15,47,17,savewin1);
// troca a janela 1 para a janela 2
    window(1,1,80,25);
    window(11,9,47,12);
    gotoxy(1,1);
    clrscr();
    puttext(12,9,47,11,savewin2);
// troca a janela 2 para a janela 1
  closegraph();
}
```

Apêndice C - DIRETIVAS DO PRÉ-PROCESSADOR

É possível incluir várias instruções para o compilador no código-fonte de um programa em C. Essas instruções são chamadas diretivas do pré-processador e embora não sejam realmente parte da linguagem C, ampliam o escopo do ambiente de programação C.

Este apêndice contém uma lista de diretivas e detalhes sobre as mais utilizadas.

De acordo com o padrão ANSI, o pré-processador contém as seguintes diretivas:

#define
#else
#elif
#endif
#error
#if
#ifdef
#ifndef
#include
#line
#pragma
#undef

O comando #define

Utiliza-se #define para definir um identificador e um valor (ou string). O compilador substituirá o identificador pelo valor cada vez que aquele for encontrado no arquivo-fonte. O padrão ANSI proposto refere-se ao identificador como *nome de macro* e ao *processo de substituição como substituição de macro.* A sintaxe é:

```
#define  identificador  valor
```

Note que este comando não contém ponto e vírgula, encerrando-se portanto, com o final da linha.

Ex:

```
#define VERDADEIRO 1
...
printf("%d ", VERDADEIRO);
```

O que causará a impressão da constante 1, no comando *printf()*.

Por convenção, escreve-se as macros em letras maiúsculas, o que ajuda a distingui-las dentro do código-fonte, normalmente escrito em minúsculas.

Ex2:

```
#define xyz "isto é um teste"
...
printf(xyz);
```

Neste exemplo, será impressa a string "*isto é um teste*", no comando *printf().*

Pode-se utilizar a diretiva #define para pequenas funções:

Ex3:

```
#define min(a,b)  ((a < b)? a:b )
```

Ou seja, toda vez que o programa encontrar a macro min(x, y), vai devolver o valor mínimo entre x e y.

O comando #include

Esta diretiva instrui o compilador para incluir um arquivo-fonte ao arquivo que contém o #include.
A sintaxe é:

```
#include  "arquivo"        ou        #include <arquivo>
```

onde *arquivo* é o arquivo-fonte a ser incluído. Se forem utilizadas as aspas, o compilador procurará o arquivo primeiro no diretório de trabalho atual, depois nos diretórios especificados na linha de comando e por último nos diretórios padrões. Se forem utilizados os sinais de "maior e menor que" o compilador procurará o arquivo primeiro nos diretórios especificados na linha de comando do compilador, depois nos diretórios padrões e por último, no diretório de trabalho atual.

Esta diretiva é muito utilizada para acessar-se as funções já incorporadas aos compiladores tradicionais (como Turbo C, Borland C) e facilitar a vida do programador que não precisará recriar funções, como por exemplo, de entrada e saída (printf(), scanf(), puts(), gets(), etc). Estas funções encontram-se em arquivos que convencionou-se chamar de "*Header*" (porisso a extensão ".h") e o conteúdo de cada arquivo varia de compilador para compilador.

Comandos de compilação condicional (#if, #else, #elif, #endif)

Estas diretivas permitem que se compile de forma seletiva partes do código-fonte do programa.
Na diretiva #if, se a expressão constante que vem depois de #if for verdadeira, então o processador compilará o código entre esse comando e a diretiva #endif. A sintaxe é:

```
#if expressão_constante
    sequência de comandos
#endif
```

A diretiva #else funciona como um "senão" do C à diretiva #if e a diretiva #elif, como um "senão se".

Os comandos #ifdef e #ifndef (ou #if defined #if !defined)

Estas diretivas, que podem tomar os formatos acima, dependendo do compilador, selecionam o que será compilado "se algo foi definido" ou "se algo não foi definido".
Ex:

```
#if defined( _Windows )
    #error BGI graphics not supported under Windows
#endif
```

Ou seja, se (_Windows) foi definido, a diretiva **#error** (que obriga o compilador a parar a compilação e apresentar a mensagem de erro " BGI graphics not supported under Windows") é executada. Caso contrário, o compilador não entra no bloco #ifdef.

BIBLIOGRAFIA - LINGUAGEM C

MIZRAHI, VICTORINE V. - *Treinamento em Linguagem C - Módulos 1 e 2* - , McGraw-Hill, São Paulo, 1990.

KELLY-BOOTLE, STAN - *Dominando o Turbo C* - 2a. ed. - Ed. Ciência Moderna, Rio de Janeiro, 1989.

PUGH, KENNETH - *Programando em Linguagem C* - McGraw-Hill, São Paulo, 1990.

SCHILDT, HERBERT - *Turbo C: Guia do Usuário* - 2a. ed. revisada - McGraw-Hill, São Paulo, 1988.

PAPPAS, CHRIS H. & MURRAY, WILLIAM H. - *Turbo C++: Completo e Total* - McGraw-Hill, São Paulo, 1991.

SCHILDT, HERBERT - *Linguagem C: Guia do Usuário* - McGraw-Hill, São Paulo, 1986.

RITCHIE, DENNIS M. & KERNIGHAN, BRIAN W. - *C, a linguagem de programação* - Ed. Campus, Rio de Janeiro; Edisa, Porto Alegre; 1986.

WIENER, RICHARD S. - *Turbo C, passo a passo* - Ed. Campus, Rio de Janeiro, 1991.