

 UNISINOS UNIVERSIDADE DO VALE DO RIO DOS SINOS

CURSO DE :

# *LINGUAGEM C*

por

FERNADO SANTOS OSÓRIO

São Leopoldo - 1992

## ÍNDICE GERAL DE ASSUNTOS

### 1 - Introdução

- 1.1 - Histórico
- 1.2 - Características da linguagem
- 1.3 - Organização deste manual

### 2 - Tipos de dados

- 2.1 - Tipos de dados pré-definidos
- 2.2 - Conversão de tipos
- 2.3 - Definição de novo tipos de dados
- 2.4 - Tamanhos dos tipos de dados

### 3 - Operadores

- 3.1 - Operador de atribuição
- 3.2 - Operadores aritméticos
- 3.3 - Operadores lógicos
- 3.4 - Operadores relacionais
- 3.5 - Manipulação de bits
- 3.6 - Operadores de assinalamento
- 3.7 - Operadores de pré e pós incremento
- 3.8 - Operadores de endereço
- 3.9 - Tabela de operadores do "C"
- 3.10- Precedência dos operadores

### 4 - Expressões condicionais

### 5 - Definições

- 5.1 - Constantes
- 5.2 - Variáveis
  - Declaração
  - Tipos de variáveis
  - Inicialização
- 5.3 - Conversão de tipos

### 6 - Macros do Pré-processador

### 7 - Estruturas gerais

- 7.1 - Comandos
- 7.2 - Blocos
- 7.3 - Funções
- 7.4 - Programas

### 8 - Comandos de "C"

- 8.1 - If/Else
- 8.2 - While
- 8.3 - For
- 8.4 - Do/while
- 8.5 - Break

- 8.6 - Continue
- 8.7 - Switch/case
- 8.8 - Goto
- 8.9 - Sizeof
- 8.10 - Return

## 9 - Passagem de parâmetros

- 9.1 - Parâmetros do "main"
- 9.2 - Parâmetros de funções

## 10 - Apontadores

## 11 - Estruturas

## 12 - Uniões

## 13 - Funções pré-definidas

- 13.1 - Funções de E/S padrão
- 13.2 - Funções de manipulação de arquivos
- 13.3 - Funções de uso geral

## Apêndices :

- A - Compilador "C"
- B - Padrão K&R X ANSI
- C - Erros mais comuns de "C"
- D - Exemplos de programas
- E - Exemplo de Header

## Bibliografia

## 1 - INTRODUÇÃO :

### 1.1 - HISTÓRICO :

A linguagem "C" é uma linguagem que foi criada nos laboratórios da BELL por Brian W. Kernighan e Dennis Ritchie em 1972. Esta linguagem, teve suas idéias iniciais originadas da linguagem BCPL ( Basic Combined Programming Language ), desenvolvida por Martin Richards. Esta influência do BCPL se deu através de outra linguagem, chamada "B" e criada por Ken Thompson em 1970 para o primeiro sistema operacional UNIX no PDP-11.

A partir de sua criação a linguagem "C" sofreu uma longa evolução sendo que uma de suas primeiras utilizações foi a de reescrever o sistema operacional UNIX (1973) que estava escrito em linguagem assembler do PDP-11. Por este motivo é que se tem associado a linguagem ao S.O. UNIX, visto que o UNIX é composto atualmente, quase na sua totalidade, de programas escritos em "C" (Sist. Operacional, utilitários, compiladores, ...). Entretanto isto não implica que o "C" seja uma linguagem "amarrada" a um sistema operacional ou máquina.

Devido a evolução do "C", que seguia apenas o padrão descrito por Kernighan e Ritchie, tornou-se necessária uma padronização mais rígida para a linguagem, permitindo a portabilidade dos softwares escritos nesta linguagem. Isto foi feito pelo ANSI (American National Standards Institute), criando assim o padrão C ANSI.

### 1.2 - CARACTERÍSTICAS DA LINGUAGEM :

O "C" é uma linguagem de propósitos gerais e que tem como características principais :

- Controle de fluxo e estrutura de dados adotando conceitos modernos de linguagens de programação.
- Poderoso conjunto de operadores e tipos de dados.
- Permite a geração de um código bem otimizado e compacto (Quase tão otimizado quanto o assembler).
- Grande portabilidade de programas (A maioria das máquinas existentes no mercado suportam a linguagem "C").
- É uma linguagem de nível "relativamente baixo", mas com recursos de alto nível.
- Apresenta facilidade de manipulação direta do hardware da máquina.
- Uso de bibliotecas de funções, expandindo as potencialidades da linguagem.
- "*Liberdade com responsabilidade*"

### 1.3 - ORGANIZAÇÃO DESTES MANUAIS :

Através deste manual tenta-se dar uma visão global da linguagem "C" no ambiente do MS-DOS em equipamentos IBM-PC. Como temos certas características do "C" que podem diferir de uma implementação para outra dos compiladores, adotamos a seguinte notação que será usada para diferenciar o "modelo" de "C" utilizado :

K&R - Padrão descrito por Kernighan e Ritchie (Default)

ANSI - Extensão do "C" proposto por K&R, definido pelo ANSI

TC - Característica ligada ao compilador "C" da Borland (Turbo C - Suporta o "C" K&R, extensões ANSI e extensões próprias)

MCS - Característica ligada ao compilador da Microsoft

Este manual tenta expor de uma maneira ordenada os seguintes itens da linguagem :

- Tipos de Dados
- Operadores e Expressões
- Comandos da Linguagem
- Funções Básicas
- Biblioteca de Funções
- Manipulações de estruturas mais complexas : Pointers, Unions, Structs e arquivos

## 2 - TIPOS DE DADOS :

### 2.1 - TIPOS DE DADOS PRÉ-DEFINIDOS :

- CHAR** :           Caracter / Byte
- INT** :             Inteiro
- SHORT** :         Inteiro "curto". Expl.: short int a; short b;
- LONG** :           Inteiro "longo". Expl.: long num;  
Obs.: O nome "int" é opcional na declaração short/long
- FLOAT** :         Real de precisão simples
- DOUBLE** :        Real de precisão dupla
- Pointer (\*)** :    Apontador, contém um endereço
- Array ([ ])** :    Tabela de elementos de um determinado tipo. Declaração:  
                  <Tipo\_Var> <Nome\_Var> [<Tamanho\_Tabela>]  
                  Expl.: int tab[10] - Array de inteiros de 0 até (X-1) posições,  
                  chamado TAB. Elementos : tab[0] até tab[9]
- ENUM (ANSI)** : Enumeração de elementos. É considerado como sendo um inteiro, onde o primeiro elemento é sempre correspondente a zero.  
Expl.: enum days {seg, ter, qua, ... }, onde seg=0, ter=1, ...  
Podemos especificar os valores: enum unidades { um = 1, par=2, meia\_duzia=6, duzia=12 }
- UNSIGNED** : Indica que a variável será sem sinal . Se não for dado o tipo da variável será considerado como "int". Expl.: unsigned var; unsigned int var;
- SIGNED (ANSI)** : Indica que a variável será com sinal. Expl.: signed char a; signed int var;

OBS.:

- Os arrays sempre começam na posição (índice) zero ! Cuidado se você definiu um array a[10], ele terá as 10 posições de a[0] até a[9].

- Os arrays de caracteres (strings) devem ser dimensionados com uma posição extra para o caracter de fim de string '\0'.

2.2 - CONVERSÃO DE TIPOS :

Faz a conversão de tipo de uma variável para outro tipo.

**Sintaxe :** (nome-do-tipo)expressão  
**Exemplo :** vardoub = (double)varint;  
fatorial = fat((int)valor\_doub);

Obs.: Sendo X = Double , A e B int e a expressão x = a / b. O cálculo é feito primeiro com os inteiros e após a conversão

2.3 - DEFINIÇÃO DE NOVOS TIPOS DE DADOS :

Permite "criar" tipos de dados diferentes dos pré-definidos

**Sintaxe :** typedef <tipo> <novo\_tipo>  
**Exemplo :** typedef long int inteiro;  
inteiro var;  
typedef enum { sun, mon, tues, wed, thur, fri, sat } days;  
days today;

2.4 - TAMANHO DOS TIPOS DE DADOS :

TIPOS - TC	TAMANHO	FAIXA DE VALORES
UNSIGNED CHAR	8 Bits	0 á 255
CHAR	8 Bits	-128 á 127
ENUM	16 Bits	32768 á 32767
UNSIGNED SHORT	16 Bits	0 á 65535
SHORT	16 Bits	32768 á 32767
UNSIGNED INT	16 Bits	0 á 65535
INT	16 Bits	32768 á 32767
UNSIGNED LONG	32 Bits	0 á 4294967295
LONG	32 Bits	-2147483648 á 2147483647
FLOAT	32 Bits	3.4E-38 á 3.4E+38
DOUBLE	64 Bits	1.7E-308 á 1.7E+308
LONG DOUBLE	64 Bits	1.7E-308 á 1.7E+308
POINTER	16 Bits	Pointers intra-segmentos
POINTER	32 Bits	Pointers inter-segmentos

\* Turbo C - Borland compiler

TIPOS - MSC	TAMANHO	FAIXA DE VALORES
CHAR	8 Bits	0 á 255

UNSIGNED SHORT	8 Bits	0 á 255
SHORT	16 Bits	-128 á 127
UNSIGNED INT	16 Bits	0 á 65535
INT	16 Bits	-32768 á 32767
UNSIGNED LONG	32 Bits	0 á 4294967295
LONG	32 Bits	-2147483648 á 2147483647
FLOAT	32 Bits	3.4E-38 á 3.4E+38
DOUBLE	64 Bits	1.7E-308 á 1.7E+308
POINTER	16 Bits	Pointers intra-segmentos
POINTER	32 Bits	Pointers inter-segmentos

\* Microsoft C compiler

### 3 - OPERADORES :

#### 3.1 - OPERADOR DE ATRIBUIÇÃO :

"=" -> Atribui um valor ou resultado de uma expressão contida a sua direita para a variável especificada a sua esquerda.

Exemplo :     A = 10; B = C \* VALOR + GETVAL(X);  
                   A = B = C = 1; -> Aceita associação sucessiva de valores

#### 3.2 - OPERADORES ARITMÉTICOS :

Operam sobre números e expressões, resultando valores numéricos

"+"     -> soma  
 "-"     -> subtração  
 "\*"     -> multiplicação  
 "/"     -> divisão  
 "%"     -> módulo da divisão (resto da divisão inteira)  
 "-"     -> sinal negativo (operador unário)

#### 3.3 - OPERADORES LÓGICOS :

Operam sobre expressões, resultando valores lógicos de True ou False. Possuem a característica de "short circuit", ou seja, sua execução é curta e só é executada até o ponto necessário.

"&&"    -> operacao AND  
 "||"    -> operacao OR  
 "!"     -> operador de negação NOT (op. unário)

Exemplos de "short circuit" :

( A == B ) && ( B == C ) -> Se A != B para a avaliação da expressão

( A == B ) || ( B == C ) -> Se A == B para a avaliação da expressão

Caso crítico :

if (i<= LIMIT) && list[i] != 0) ... ;

#### 3.4 - OPERADORES RELACIONAIS :

Operam sobre expressões, resultando valores lógicos de True ou False.

">" ">="    -> maior e maior ou igual

"<" "<="	-> menor e menor ou igual
"=="	-> igual
"!="	-> nao igual (diferente)

Cuidado ! Não existem os operadores relacionais : "<=", ">=" e "<>". Não confunda a atribuição ("=") com a comparação ("==").

### 3.5 - MANIPULAÇÃO DE BITS (Bitwise Operators)

A manipulação é feita em todos os bits da variável, a qual não pode ser do tipo float ou double.

"&"	-> Bit and
" "	-> Bit or
"^"	-> Bit eor - exclusive or
-> Shift left	
-> Shift right	
"~"	-> Bit not (complemento)

Obs.:  $x \ll n$  - irá rotar n vezes a esquerda

### 3.6 - OPERADORES DE ASSINALAMENTO :

É expresso da seguinte forma : (operadores combinados)

$VAR = VAR OP EXPR \implies VAR OP= EXPR$

Onde temos OP como um dos seguintes operadores :

"+"	-> Soma
"-"	-> Subtração
"*"	-> Multiplicação
"/"	-> Divisão
"%"	-> Modulo da divisão
-> Shift right	
-> Shift left	
"&"	-> And
"^"	-> Eor - exclusive or
" "	-> Or

Exemplo de aplicação :  $I = I + 2$  será  $I += 2$

### 3.7 - OPERADORES DE PRÉ E PÓS INCREMENTO/DECREMENTO :

As operações abaixo podem ser representadas assim :

$I = I + 1$	-> $I = ++ I$
$I = I - 1$	-> $I = -- I$
$Z = A ; A = A + 1$	-> $Z = A ++$
$Z = A ; A = A - 1$	-> $Z = A --$
$A = A + 1 ; Z = A$	-> $Z = ++ A$
$A = A - 1 ; Z = A$	-> $Z = -- A$

### 3.8 - OPERADORES DE ENDEREÇO :

Usados com ponteiros (pointers), para acesso a endereços de memória

"&" -> Endereço de uma variável. Expl.: int var,\*x; x = &var;

"\*" -> Conteúdo do endereço especificado. Expl.: var = \*x;

### 3.9 - TABELA DE OPERADORES DO "C" :

Op.	Função	Exemplo "C"	Exemplo PASCAL
-	Menos unário	A = -B	A := -B
+	Mais unário	A = +B	A := +B
!	Negação Lógica	! FLAG	not FLAG
~	Bitwise NOT	A = ~B	A := not B
&	Endereço de	A = &B	A := ADDR(B)
*	Referência a ptr	A = *ptr	A := ptr^
sizeof	Tamanho de var	A = sizeof(b)	A := sizeof(b)
++	Incremento	++A ou A++	A := succ(A)
--	Decremento	--A ou A--	A := pred(A)
*	Multiplicação	A = B * C	A := B*C
/	Divisão inteira	A = B / C	A := B div C
/	Divisão real	A = B / C	A := B / C
%	Módulo da divisão	A = B % C	A := B mod C
+	Soma	A = B + C	A := B + C
-	Subtração	A = B - C	A := B - C
>>	Shift Right	A = B >> N	A := B shr N
<<	Shift Left	A = B << N	A := B shl N
>	Maior que	A > B	A > B
>=	Maior ou igual a	A >= B	A >= B
<	Menor que	A < B	A < B
<=	Menor ou igual a	A <= B	A <= B
==	Igual a	A == B	A = B
!=	Diferente de	A != B	A <> B
&	Bitwise AND	A = B & C	A := B and C
	Bitwise OR	A = B   C	A := B or C
^	Bitwise XOR	A = B ^ C	A := B xor C
&&	Logical AND	flg1 && flg2	flg1 and flg2
	Logical OR	flg1    flg2	flg1 or flg2
=	Assinalamento	A = B	A := B
OP=	Assinalamento	A OP= B	A := A OP B

### 3.10 - PRECEDÊNCIA DE OPERADORES :

Expressões em "C" tem uma regra de precedência de avaliação, de acordo com a qual, será executado antes ou depois algum operador de uma expressão. Em certas expressões onde há sobreposição de diversos operadores do mesmo nível de precedência, faz-se necessário o uso de parênteses para delimitar e corrigir a ordem de avaliação da expressão. Os níveis de precedência dos operadores em "C" podem ser encontrados na tabela listada abaixo :

Nível Prec.	Operadores
15 (+ Alta)	( ) [ ] -> .

	Parêntese Colchetes Ponteiro Ponto
14	! ~ * & ++ -- Negação NOT Ponteiro Endereço Incr. Decr. (TYPE) - sizeof Conversão Menos (Unário) Tamanho
13	* / % Multiplic. Divisão Módulo
12	+ (Soma) - (Subtração)
11	>> << Shift Left Shift Right
10	< <= > >= Menor Menor ou Igual Maior Maior ou Igual
9	== != Igual Diferente
8	& (Bitwise AND)
7	^ (Bitwise XOR)
6	(Bitwise OR)
5	&& AND Lógico
4	 OR Lógico
3	?: Condicional
2	= += -= *= /= %=  = ^= &= <<= >>= Atribuição Atribuição Combinada
1	, Vírgula

#### 4 - EXPRESSÕES CONDICIONAIS :

As expressões condicionais se apresentam da seguinte forma :

EXPR1 ? EXPR2 : EXPR3

Esta expressão é equivalente a :

SE EXPR1                    Onde : Expr1 -> Condição de teste  
ENTAO EXPR2                Expr2/Expr3 -> Valor retornado  
SENAO EXPR3

Exemplo :

```
#define IMIN(A,B) ((A<B)?A:B)
B = ((X == Y)?X:Y);
```

## 5 - DEFINIÇÕES :

### 5.1 - CONSTANTES :

Não podem ter seus valores alterados após a sua definição. Declaradas das seguintes formas :

```
const <tipo_var> <nome_const> = <valor>
# DEFINE <nome_const> <valor> <== Macro do pré-processor
```

Exemplos de constantes :

```
255    -> Decimal      * constantes Inteiras
0777   -> Octal       * começando com 0 -> Octal
0XFF   -> Hexadecimal * começando com 0x -> Hexa
nnnL   -> Long        * L,U -> forçam uso do tipo
nnnU   -> Unsigned    * podem ser usados juntos
nnnF   -> Float       * força armazenamento como float
'c'    -> Character (seu valor é o ascii de 'c' - tipo char)
'CC'   -> Character ( 2 caracteres - tipo int ) (TC)
      \XXX   -> Octal XX (expl.: '\014' é o form feed).
                São esperados 3 dígitos após o "\" !
\n     -> New Line    LF 0x0A * Sequências de escape
\b     -> Backspace   BS 0x08
\r     -> Carriage Return CR 0x0D
\f     -> Form Feed   FF 0x0C
\t     -> Tab Horiz   HT 0x09
\v     -> Tab Vertical VT 0x0B
\0     -> Null        0x00
\\     -> Backslash   \ 0x5C
\'     -> Single quote ' 0x2C
\"     -> Double quote " 0x22
\?     -> Question Mark ? 0x3F
\la    -> Bell        BEL 0x07 (ANSI)
\xNN   -> Repr. hexa  0xNN (ANSI)
```

OBS.: Note a diferença entre :

```
'x' -> caracter-contante
"x" -> string de um caracter
```

As strings são terminadas por um caracter '\0' devendo os arrays de caracteres ter uma posição a mais para comportar este caracter.

- Diferença entre strings constantes e arrays de caracteres :

```
1) char *msg;          2) char msg[20];
   msg = "COMO VAI ?";  strcpy (msg,"COMO VAI ?");
```

A diferença está no fato de que o exemplo 1 não aloca área para a string, e após associa apenas o pointer a uma área de dados onde estará a string definida pelo programa. No exemplo 2, já temos a área definida, e portanto podemos usá-la livremente.

### 5.2 - VARIÁVEIS :

### 5.2.1 - Declaração :

A declaração de variáveis pode ser feita em qualquer parte do programa sempre no início de um bloco (após uma "{"). Cabe aqui citarmos algumas informações a respeito declaração de variáveis :

- Nomes de variáveis começam com uma letra ('A'..'Z','a'..'z') ou pelo underscore ('\_').
- Após podem ser seguidos dígitos, letras e underscores. No caso do TC podemos ter até 32 caracteres definindo um identificador.
- Evite o uso do '\_' no primeiro caracter do identificador de uma variável, pois este tipo de identificadores é de uso do sistema.
- Normalmente ao declararmos uma variável esta será inicializada com zero, como veremos em seguida.
- São diferenciados os caracteres minúsculos dos maiúsculos no nome de qualquer variável.

Declaração de variáveis :

```
<modific> <tipo_dado> <nome_var>;      ou
<modific> <tipo_dado> <var1,var2,...>;
```

```
Expl.: extern  int   nroint;
        unsigned int  n1,n2,n3;
        char    letra;
```

### 5.2.2 - Tipos de variáveis :

Quanto ao acesso :

\* **Externas** : São variáveis GLOBAIS declaradas fora de qualquer função ou bloco, podendo ser referidas em outras partes do programa de forma explícita ou implícita. Também conhecidas como variáveis do tipo "common". Podem ser definidas da seguinte maneira :

Referência explícita : EXTERN <tipo\_var> <nom\_var>

Referência implícita : não citá-las.

Exemplo :       Arquivo UM       Arquivo DOIS

```
int x,y;          extern int x,y;
char ch;extern char ch;
main ( )          funcxy ( )
{                 {
...              x=y/10; ...
}                 }
```

As variáveis externas GLOBAIS podem ser acessadas por qualquer função do programa desde que já tenham sido declaradas.

\* **Internas** : São definidas dentro de funções, ou blocos e podem ser do tipo estáticas, automáticas ou registradores. Estas variáveis serão LOCAIS ao bloco ou função em que

forem definidas, ou seja, só são acessíveis dentro do bloco ou da função em que estão declaradas.

Quanto a alocação :

\* **Automáticas** : São as variáveis comuns internas de funções. Só existem durante a execução da referida função, sendo "apagadas" após o término desta e no caso de uma nova chamada da função não possuirão mais o seu antigo valor. São variáveis dinâmicas e são alocadas na pilha do sistema (STACK), as variáveis internas e os parâmetros passados para as funções são deste tipo.

\* **Estáticas** : Podem ser internas ou externas as funções, mas estas mesmo após o fim da execução, ao contrário das automáticas, continuam em memória. Permitem também a não interferência em outras funções ou seja podem haver outras variáveis declaradas com o mesmo nome destas pois elas serão invisíveis sendo vistas apenas na função onde foram declaradas. São declaradas da seguinte forma :

```
static <tipo_dado> <nome_var>;
```

As variáveis estáticas são armazenadas na área de dados do sistema, que é fixa. Variáveis estáticas são as globais e as definidas pelo STATIC.

\* **Registradores** : São variáveis muito usadas e que se possível serão alocadas em um dos registradores do processador. Usa-se com variáveis do tipo int, char e apontadores. São declaradas da seguinte forma :

```
register <tipo_dado> <nome_var>;
```

### 5.2.3 - Inicialização :

Regras de inicialização de variáveis :

- externas e estáticas -> inicializadas com "0"
- automáticas e registradores -> contêm lixo

Podemos também definir o valor inicial da variável no momento de sua declaração, o que é feito da seguinte forma :

```
<modific> <tipo_dado> <nome_var> [= valor];
```

Exemplos :  
char ch = 'A';  
int first = 1;  
float balance = 123.23;  
char msg[9] = "Como vai ?";

Arrays :  
char num[5] = { 1, 2, 3, 4, 5 };  
char str[5] = { 'A','B','C','D','E' };  
char z[4][2] = { { 0, 0 }, { 0, 1 }, { 1, 0 }, { 1, 1 }  
{ 2, 0 }, { 2, 1 }, { 3, 0 }, { 3, 1 } };

### 5.3 - CONVERSÃO DE TIPOS :

O "C" segue algumas regras para a conversão de tipos de dados, conversão esta que é feita para possibilitar ao compilador de realizar operações requeridas em expressões com tipos compatíveis entre si. As regras de conversão de tipos são as seguintes :

- Todos os valores não-inteiros ou não-double são convertidos como é mostrado na tabela abaixo. Após isto os dois valores a serem operados serão ou do tipo int (incluindo long e unsigned) ou do tipo double.
- Se um dos operandos é do tipo double, o outro operando também será convertido para double.
- Por outro lado, se um dos operandos for do tipo unsigned long, o outro será convertido para unsigned long.
- Por outro lado, se um dos operandos for do tipo long, então o outro operando será convertido para long.
- Por outro lado, se um dos operandos for do tipo unsigned, então o outro operando será convertido para unsigned.
- E por último, ambos os operandos serão do tipo int.

Tabela de conversão : (TC)

<u>TIPO</u>	<u>CONVERTIDO PARA</u>	<u>MÉTODO USADO</u>
char	int	extensão de sinal
unsigned char	int	zera o byte mais significativo
signed char	int	extensão de sinal
short	int	se unsigned, entao unsigned int
enum	int	mesmo valor
float	double	preenche mantissa com 0's

Exemplo :

```

char ch;   result = ( ch / i )   +   ( f * d )   -   ( f + i )
int  i;
float f;
double d;
double result;

```

$\begin{array}{ccccccc}
& | & / & & | & & | \\
& | & | & & | & & | \\
\text{int} & & \text{int} & & \text{double} & & \text{double} \\
& \backslash & / & & \backslash & / & \backslash & / \\
& & \text{int} & & \text{double} & & \text{double} \\
& & & & & & & | \\
& & & & +-----double-----+ & & & + \\
& & & & & & & +-----double-----+
\end{array}$

## 6 - MACROS DO PRÉ-PROCESSADOR :

As macros são instruções analisadas por um pré-processador (CPP - "C" pré-processor) que realiza as operações definidas. As principais macros existentes são :

```

#include
#define

```

```
#if
#ifdef
#endif
#line
#undef
#else
#ifndef
#elif
```

- Utilização, sintaxe e exemplos :

\* **DEFINE** : Força a substituição dos nomes dados pelo texto de reposição, dentro do código fonte do programa. Há também a macro-substituição com parâmetros, na qual os argumentos são trocados posicionalmente (de acordo com sua posição) e por correspondência. Esta operação é realizada antes do início da compilação, sendo transparente ao compilador. Sintaxe :

```
#define <nome> <texto_de_reposição>      ou
#define <nome> ( argumento, argumento, ...> <texto_de_reposição>
```

As definições podem ser bem extensas podendo continuar em outras linhas com o auxílio de uma "\n" no final da linha.

```
Exemplo :      #define MAXVAL 999
                #define then
                #define begin {
                #define end   }
                #define max(a,b) ((a>b) ? a:b)
```

Fonte antes do CPP

```
if (num == valor)
then begin
    i++;
    x = MAXVAL;
end;
else x = max (num,valor);
```

Fonte após o CPP

```
if (num == valor)
{
    i++;
    x = 999;
}
else x = ((num>valor) ? num:valor);
```

\* **UNDEF** : Desfaz a definição, faz com que o pré-processador "esqueça" a definição feita anteriormente. Sintaxe :

```
#undef <identificador>
```

Exemplo : #undef MAXVAL

\* **INCLUDE** : Causa a substituição da linha pelo conteúdo do arquivo informado. Inclui um trecho de um programa contido no arquivo especificado, a partir da linha onde foi dado o include. Sintaxe :

```
# INCLUDE "nome_do_arq"      ou
# INCLUDE <nome_do_arq>     -> São aceitas tanto as "" como <>
```

Obs.: No TC as "" indicam que o arquivo include se encontra no diretório corrente e o <> indica que o arquivo deve ser procurado no diretório definido como diretório de arquivos include.

Exemplos :

```
#include "arquivo"    * Inclui o texto do "arquivo" no texto fonte
#include <stdio.h>    * Inclui a biblioteca de rotinas STDIO.H
```

\* **Compilação Condicional** : Há um conjunto de instruções que são definidas para permitir uma compilação condicional do programa fonte. Certas funções podem ser ou não compiladas conforme estas diretivas :

```
#if <expr_constante> - Se != 0 então gera o código que segue
#ifdef <identificador> - Se identificador for definido gera código
#ifndef <identificador> - Se identificador não for definido gera cod.
#else
#elif
#endif                - Final do trecho de "código condicional"
- Funciona em conjunto com if.. (caso oposto)
- Associação de um else com if
```

Exemplo :

```
#define versão 2.3
#if versão >= 2.1    * Inclui o arquivo "ext_21" caso a versão
#include <ext_21.c>   * definida seja a 2.1
#endif
```

\* **LINE** : Faz com que o compilador acredite, para fins de diagnóstico de erros, que o número da próxima linha do fonte seja dado pelo valor constante indicado e que o nome do arquivo corrente de entrada seja dado pelo identificador também indicado. Sintaxe :

```
#line <cte_nro_lin> [ <ident_arq> ]
```

Exemplo : #line 1000 ARQATUAL

## 7 - ESTRUTURAS GERAIS :

### 7.1 - COMANDOS :

Todos os comandos de "C" devem ser escritos em minúsculas e devem ser seguidos de um ";". Sintaxe : comando;

### 7.2 - BLOCOS :

Os blocos são definidos de uma "{" até a outra "}", equivalem aos blocos de PASCAL definidos pelo "begin" e "end". Dentro de cada bloco podemos então definir um conjunto de comandos, como se fossem um só. Outra característica interessante dos blocos é que sempre no início destes (após o "{") podemos declarar novas variáveis (variáveis locais ao bloco), independentemente da posição deste bloco no programa. Não pode-se por ";" após um fim de um bloco.

```
Formato : { -> início de bloco
           } -> fim de bloco
```

### 7.3 - FUNÇÕES :

São subrotinas (conjunto de comandos) que podem ser chamados de diferentes partes do programa quando necessário. Normalmente retornam um valor ao final de sua execução e

podem receber parâmetros de entrada para a sua execução. Para executar uma função basta referenciar o seu nome. São definidas da seguinte forma :

```
[<tipo_ret>] <nome_da_função> ( [<lista_argumentos>] )
[<declaração_de_argumentos>;]
{
    <declarações_locais>;
    <comandos>;
    [ return [ (<resultado>) ];
}

```

Chamada : <nome\_da\_função> ( [<lista\_argumentos>] );

A definição de funções NÃO pode ser "aninhada" ! Note também que quando definimos uma função não colocamos o ";" nesta linha. Além de uma definição, as funções podem ter também uma declaração, que servirá para identificar ao compilador o tipo retornado pela função caso esta não retorne o tipo default (int). É feito da seguinte forma :

```
<tipo_ret> <nome_função> ( ); ou
<tipo_ret> <nome_função> ( <declaração de parâmetros> ); (ANSI)

```

Exemplo :      char \*readtxt ( );  
                 double sqrt(int x);  
                 void plot (unsigned int x1,unsigned int x2,char cor);

A forma alternativa de declaração de funções pelo padrão ANSI, onde definimos um esqueleto da função, damos o nome de "prototype".

Obs.: void = não retorna nada / retorno indefinido.

Temos uma função especial nos programas esta função é chamada de MAIN e é chamada sempre que um programa começa a ser executado. Quando um programa for ser executado pelo computador, ele iniciará a execução pela função main, que obrigatoriamente tem que estar presente em qualquer programa "C" (Exceto módulos linkados).

#### 7.4 - PROGRAMAS :

Sua estrutura segue aproximadamente esta forma :

```
#include <bibliotecas>
#define <funções_de_macros>
#define <constantes>

declaração_de_funções;
definição_de_novos_tipos_de_dados;
definição_de_variáveis_globais;

main ( )
{
    definição_variáveis_locais;
    comandos;
}

/* Comentários : texto livre */

tipo_ret função (argumentos)
```

```

definição_de_argumentos;
{
    definição_variáveis_locais;
    comandos;
    return (variável);
}

```

Características dos programas em "C" :

- Os módulos do programa (funções, defines, includes, def. globais) podem estar localizados em qualquer parte do programa. O MAIN pode ser localizado também em qualquer parte, mas por motivos de organização, devemos colocá-lo no início ou fim do programa.

- Não são aceitas funções aninhadas, ou seja, funções declaradas dentro de funções e os comentários ("/\*", "\*/") podem ou não ser aninhados conforme o compilador (K&R - Comentários não aninhados).

- São opcionais : A existência de um valor de retorno nas funções, o tipo de retorno da função (se retornar o tipo default = int) e a declaração de funções no caso de retornarem valores inteiros ou estarem declaradas antes de seu uso.

## 8 - COMANDOS DE "C" :

### 8.1 - IF/ELSE :

Execução condicional de comandos. A instrução IF causa a execução de uma ou um conjunto de instruções, dependendo do valor resultante da expressão avaliada. Sintaxe :

```

if (expressão)
    comando1;
[ else
  ]
[ comando2;]

```

- As instruções de execução condicional dependem da avaliação de uma expressão condicional. Esta expressão é composta por várias operações, que são efetuadas através dos operadores lógicos e relacionais, devolvendo um valor booleano TRUE ou FALSE. No "C" a expressão condicional pode devolver um valor inteiro, que terá o seguinte significado :

0 - FALSE  
 Não Zero - TRUE.

- Expressão : é uma sequência de operadores lógicos, relacionais, aritméticos e valores. Pode possuir atribuições e chamadas de funções, que serão interpretadas como valores comuns e devem vir entre parênteses. Na atribuição o valor avaliado será o valor atribuído e na chamada de função é o valor retornado. Também podemos ter listas de comandos dentro de uma expressão, separados por vírgulas (','), e que serão avaliados da esquerda para a direita, sendo o último comando o que indicará o valor a ser interpretado na expressão. Exemplos :

```

((sum = 5 + 3) <= 10)    ==> Expr. avaliada : 8 <= 10
((ch=getch( )) == 'Q')  ==> Expr. avaliada : ch == 'Q'
                          Antes é executada a ch=getch( )
(oldch=ch,ch=getch( )) ==> Expr. avaliada : ch != 0

```

## 8.2 - WHILE :

Enquanto a condição descrita pela expressão for satisfeita (verdadeira) o comando será repetido. A avaliação da expressão é realizada da mesma forma que no comando IF. O comando WHILE é o único (e o principal) comando de repetição realmente necessário, sendo que pode substituir qualquer outra construção do tipo : FOR, REPEAT/UNTIL, DO/WHILE, GOTO ...

```
Sintaxe : while (expressão)
           comando;
```

## 8.3 - FOR :

O comando FOR serve para a execução de um número fixo de vezes, enquanto uma variável percorre uma determinada faixa de valores. Esta variável é chamada de variável de índice. Sintaxe :

```
for (início; condição; modific)
    comando;

ou

for (expr1,expr2,...;expr3,...;expr4,...)
    comando;
```

Onde temos "início" como sendo uma expressão que irá gerar o valor inicial da variável de índice utilizada pelo comando. A "condição" irá nos indicar uma condição para o prosseguimento do loop (enquanto tal condição for verdadeira irá repetir o loop). E finalmente "modific" será o comando dado para cada execução do loop, sendo este realizado ao final de um loop do FOR, modificando o valor da variável de índice, antes de um novo teste da condição. Como pode ser visto no segundo formato do comando FOR, cada um destes elementos (início,condição e comando) pode estar dividido em sub-comandos, como nas expressões do IF.

Para criarmos um loop infinito (sem fim) podemos fazer um comando FOR da seguinte forma : FOR ( ; ; ) - comando sem expressões de controle. O comando for é equivalente a seguinte estrutura :

```
início;
while (condição)
{
    comando;
    modific;
}
```

## 8.4 - DO/WHILE :

O comando DO/WHILE é o inverso do comando WHILE, ou seja o teste é executado ao final deste ao invés de ser no início. Este comando também é equivalente ao comando REPEAT/UNTIL (Pascal) só que a expressão avaliada em um estaria negada em relação ao outro. Sintaxe :

```
do comando
```

while (expressão);

O comando será executado e depois será testada a condição dada pela expressão, e caso esta seja verdadeira (TRUE) será novamente executado o comando.

#### 8.5 - *BREAK* :

Causa a saída no meio de um loop (para comandos : for, while, do, switch). Provoca a antecipação do fim do loop. É muito usado com o comando CASE como será visto mais adiante.

#### 8.6 - *CONTINUE* :

Comando para a re-execução do loop, a partir do teste. Irá causar a reinicialização do laço (não funciona para o switch). Um exemplo prático de seu uso pode ser visto abaixo :

```
while (x <= valmax)
{
    printf ("Entre com valor : ");
    scanf ("%d",val);
    if (val < 0)
        continue;
    ...
}
```

#### 8.7 - *SWITCH/CASE* :

Faz uma associação de valores com comandos a executar. Conforme o valor dado, executa um certo número de instruções. Serve como uma estrutura mais sofisticada que os IF's encadeados :

```
if (valor == x)
    comando1;
else if (valor == y)
    comando2;
else if (valor == z)
    comando3;
...
else comando;
```

```
Sintaxe :      switch (variável)
                {
                case <valor1> : <comando1>;
                case <valor2> : <comando2>;
                case <valor3> :
                case <valor4> : <comando3>;
                                <comando4>;
                ...
                case <valor5> : comando5;
                [ default      : comando6; ]
                }
```

No comando switch não é possível definir intervalos para os quais um comando será executado, temos que definir os valores textualmente, um a um. Os valores utilizado no CASE

devem ser do tipo inteiro (compatível). Não poderemos ter valores como : float, double, strings e pointers.

O uso do comando BREAK é muito importante quando associado ao SWITCH, pois caso este não seja usado a cada CASE, será feita execução de todos os comandos até encontrar o fim do SWITCH, sendo executado inclusive o DEFAULT se houver. Por isso a estrutura comumente usada em um comando CASE será como :

```
case 'A' : x++;  
        break;  
case 'B' :  
case 'b' : y++;  
        break;
```

### 8.8 - GOTO :

Desvio incondicional do fluxo de execução de um programa para um determinado ponto apontado por label.

```
Sintaxe :      goto <label>;  
              <label>: <comandos>
```

### 8.9 - SIZEOF :

É uma função da linguagem que irá nos fornecer um inteiro que é igual ao tamanho do objeto especificado, tamanho este dado em bytes.

```
Sintaxe :      sizeof (objeto)
```

### 8.10 - RETURN :

Usado para retornar de uma função, normalmente retornando um parâmetro para a função que a chamou.

```
Sintaxe :      return;                ou  
              return (parâmetro);
```

## 9 - PASSAGEM DE PARÂMETROS :

### 9.1 - PARÂMETROS DO "MAIN" :

O módulo principal do programa (main) pode receber dois parâmetros externos opcionais, são eles : ARGV e ARGC. Sintaxe utilizada :

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

Onde temos :

ARGC - Número de apontadores

ARGV - Array de apontadores para cadeias de caracteres que contêm os argumentos.

Teremos em argc um valor maior que zero caso hajam parâmetros pois argv[0] nos dá o nome pelo qual o programa foi chamado. O exemplo abaixo mostra a associação dos parâmetros :

```
A>prog_c param1 b:param2 xyz
```

```
ARGC = 3 e   ARGV [0] = Ponteiro para "prog_c"  
             ARGV [1] = Ponteiro para "param1"  
             ARGV [2] = Ponteiro para "b:param2"  
             ARGV [3] = Ponteiro para "xyz"
```

## 9.2 - PARÂMETROS DE FUNÇÕES :

\* A passagem de parâmetros em "C" é feita normalmente BY VALUE ou seja pela pilha. O compilador faz uma cópia dos valores passados como parâmetro, colocando-os na pilha (stack) do sistema, os valores dos dados não são alterados ao se retornar de uma rotina, mesmo que seu conteúdo tenha sido modificado no interior da função, a alteração só afeta a cópia do dado. Somente a passagem de um array como parâmetro é feita BY REFERENCE.

\* Como já foi visto, devemos na definição de uma função, definir também os seus parâmetros. Um caso especial na definição de argumentos de uma função serão os arrays, onde não é feita a definição do tamanho do array apenas de seu nome, que deve ser declarado da seguinte forma :

```
função (argumento)  
char argumento[ ];
```

\* Passagem de parâmetros by reference :

- Com o uso de pointers
- No caso de arrays

\* Retorno de parâmetros em funções :

- É feito através do comando RETURN (var)
- Funções sempre retornam um INT

\* Retorno de outros tipos de dados além de inteiros e caracteres :

- Exemplo de retorno de um DOUBLE

```
double função( );  
  
main ( )  
{  
...  
}  
double função (arg1,arg2,...)  
{  
...  
}
```

- É portanto necessário declarar as funções que não retornam valores do tipo int.

- Na declaração da função somente declaramos seu nome seguido de "( )" (lista vazia de argumentos) - K&R. No padrão ANSI, declaramos o "prototype" da função.

- Temos ainda o tipo de dado (ANSI) que indica um tipo de dado indefinido ou inexistente :  
VOID . Exemplos de uso do void :

```
void myfunct(arg1,arg2)
double nro_rand(void);
void init(void)
```

- Quando fazemos um "prototype" de uma função com um número variável de parâmetros, podemos usar "..." para indicar esta indefinição quanto ao número exato de parâmetros (ANSI).

\* Exemplos de funções e suas chamadas :

soma (a,b) int a,b;	->	x = soma (10,21)	BY VALUE
readval(a,b) long *a,*b;	->	long val1,val2; readval(&val1,&val2)	BY REFERENCE
init ( )	->	init ( );	SEM PARÂMETROS
double raiz(val) int val;	->	int x; x = (int)raiz(val);	CONVERSÃO

## 10 - APONTADORES : (Pointers)

Os apontadores ou pointers são tipos de dados que tem por função "apontar" (referenciar) variáveis através de seus endereços físicos (posição da variável na memória), permitindo uma maior flexibilidade em nossos programas como : acesso indireto por ponteiros, passagem de parâmetros By Reference, alocação dinâmica de memória, criação de listas encadeadas e outras estruturas de dados mais complexas.

São declarados da seguinte forma :

```
int x,y;          /* Declaração de dois inteiros */
int *px;         /* Declaração de um pointer para inteiros */
double vard, *pd;
```

Os pointers são definidos em função do tipo de variável ao qual ele será ligado, são pointers do tipo "aponta para um dado tipo de dado". Como se usa um pointer :

```
pt = &x          /* pt recebe o endereço de x, aponta para x */
y = *pt         /* y recebe o valor apontado por PT */
*pt = 12        /* O endereço dado por pt recebe o valor 12 */
/* Se pt = &x então *pt = 12 é igual a x = 12 */
```

É impossível apontar para registradores e constantes definidas através do comando #define do processador de macros. Um uso muito prático dos pointers é com os arrays, pois este tipo de dado tem características que se comportam de maneira muito parecida a eles. Na realidade todo o array é definido como sendo um apontador para uma posição onde se encontram alocados os dados. Por isso temos :

```
char car,a[10],*ptr;
ptr = a;        /* ptr aponta para o endereço de a[0] */
ptr = &(a[0])  /* igual ao exemplo anterior a = &a[0] */
car = *(++ptr); /* car recebe o conteúdo de a[1] */
```

```
int var[5], *pint;
ptr = var      /* ptr aponta para var[0] */
ptr = *(ptr+2); /* ptr aponta para var[2] */
```

Como já pode ser visto a indexação dos arrays é feita da mesma maneira em que se trata com pointers. Então, incrementar um pointer significa somar ao endereço atual do pointer tantas unidades quanto for o tamanho do tipo de dado apontado, assim como para o endereço de um elemento de um array pode ser obtido apenas somando-se tantas unidades quanto for o tamanho do elemento do array. Note a equivalência :

```
<tipo> var[100], *ptvar;

(ptvar + i) == &(var[i])
*(ptvar + i) == var[i]
(ptvar + i) ==> Deslocamento de i*sizeof(var) sobre end. atual
```

Portanto podemos fazer somas e incrementos com pointers operando como se fossem meros endereços sendo que no caso de incrementos, o acréscimo será feito de acordo com tipo de dado ao qual o pointer atua (*soma tantos bytes quanto for o tamanho do tipo*).

Como foi visto, pointers acessam diretamente a memória do micro, por isso constituem uma ferramenta poderosa mas ao mesmo tempo perigosa, pois um descuido qualquer pode causar sérios danos. Sempre que fomos usar um pointer ele já deverá ter sido inicializado, ou seja, já deve ter sido atribuído algum endereço a ele. Outro cuidado a ser tomado é quanto a precedência de operadores, pois é comum a confusão em casos como estes :

```
int (*array)[13]; - aponta p/ array de 13 posições
int *array [13]; - array de apontadores de inteiro
```

Arrays, algumas informações extras ...

- A única diferença entre um array e um pointer é que quando definimos um array, uma área de memória é reservada para ele e quando definimos um pointer, não há alocação de memória. Exemplo :

```
char *string -> Reserva área somente para o pointer (2/4 bytes)
char string[10] -> Reserva área p/ 10 caracteres (10 bytes)
```

- Arrays são passados como parâmetros para funções como sendo um pointer para o início do array. E no caso de arrays de mais de uma dimensão, a alocação em memória se dará do seguinte jeito :

```
int tab [10] [10];
    I  J  -> tab [0] [0], tab [0] [1],.....tab [0] [9], tab [1] [0],.....
```

- Como consequência do modo de alocação de arrays de mais de uma dimensão, quando for passado como parâmetro o array, temos que indicar as outras dimensões, exceto a principal. Isto se dá pelo fato de que é passado apenas o endereço inicial do array, que é tratado como um vetor linear. Sem a indicação das outras dimensões não conseguimos distinguir os elementos de uma ou de outra dimensão. Exemplo :

```
função (a)
int a [ ] [10]
    | |
opcional obrigatório
```

## 11 - ESTRUTURAS :

As estruturas são conjuntos (agrupamentos) de dados, que definem um novo tipo de dado mais complexo, formado por tipos de dados mais simples. As estruturas podem ser definidas das seguintes formas :

```
struct tipo {
    tipo_var nome_var;
    tipo_var nome_var;
    ...
} nome_da_estrutura;
```

OU

```
struct {
    tipo_var nome_var;
    tipo_var nome_var;
    ...
} nome1,nome2,...;
```

OU

```
struct tipo {
    tipo_var nome_var;
    tipo_var nome_var;
    ...
};
struct tipo nome1,nome2,...;
```

OU

```
typedef struct {
    tipo_var nome_var;
    tipo_var nome_var;
    ...
} record;
record nome1,nome2,...;
```

Exemplo :

```
struct data {
    int dia,mes,ano,dia_ano;
    char nome_mes[10];
};
struct data today = { 3, 9, 1987, 250, "novembro" };
```

\* Referência a um elemento da estrutura :

```
nome_estrut.elemento ==> data.dia = 23;
(*ptrname).elemento ==> (*ptrdt).dia = 10;
ptrname->elemento ==> ptrdt->dia = 5;
```

\* Características das estruturas :

- Não podem ser inicializadas na declaração ou copiadas através de uma atribuição.

- Podem ser acessadas com o auxílio de pointers. Permitem a criação de listas encadeadas (Vide programa exemplo - Apêndice A).

\* Array de estruturas :

```
struct tipo1 { tipo2 nome_eleme; ... } arranjo [x];
      |      |              |
      tipo_estrut  tipo_eleme      nome_estrut
```

\* Estrutura de bits : (Bitfields)

```
struct {
    unsigned nome1 : numero_bits;
    unsigned nome2 : numero_bits;
    ...
} nome_estrut;
```

Exemplo :

```
struct {
    unsigned flag1 : 1;
    unsigned doisbit : 2;
} flags;
```

## 12 - UNIÕES :

É alocado para este tipo de dados o espaço para a variável de maior tamanho, sendo todas variáveis alocadas na mesma área de memória e só podendo existir fisicamente uma das variáveis por vez. Sintaxe :

```
union tipo_uni_o {
    tipo_var nome_var;
    tipo_var nome_var;
    ...
} nome_uni_o;
```

O acesso é feito como são acessadas as variáveis de uma structure, como é mostrado no exemplo abaixo :

```
typedef union {
    int word;
    struct {
        char lob;
        char hib;
    } byte;
} worb;

worb myvar;

myvar.word = val_int;
myvar.byte.lob = val_byte;
myvar.byte.hib = val_byte;
```

- Esta estrutura equivale a construção de "variant record" ou "variant fields" existente na linguagem Pascal.

## 13 - FUNÇÕES PRÉ-DEFINIDAS :

As funções pré-definidas do "C", são definidas nos arquivos header (com extensão ".H"), sendo que existem diversos header's padrão no "C", entre eles podemos citar alguns :

CTYPE.H	- Definição de tipos e teste de tipos de dados e variáveis
STDIO.H	- Funções de I/O nos dispositivos stdin, stdout e files
DOS.H	- Funções de acesso as INT's do BIOS e DOS do MS-DOS
STRING.H	- Funções de manipulação de strings (TC)
MATH.H	- Funções matemáticas em geral

### 13.1 - FUNÇÕES DE E/S PADRÃO :

\* Para termos acesso a estas funções devemos no programa fazer a seguinte declaração :

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

\* PRINTF : Faz parte da família de funções de saída formatada, enviando os dados para a STDOUT (saída padrão).

Sintaxe :

```
printf ("format_string",arg1,arg2,...);
```

Onde :

Format\_string : formado por caracteres ordinários mais especificações de formato. Os caracteres ordinários são copiados diretamente na saída padrão.

Especif\_formato : Composto pelos seguintes elementos dados abaixo

```
% [flags] [width] [precision] [size] type
```

Flags : Podem ser os caracteres,

```
"-" ==> Implica no ajuste a esquerda  
"+" ==> Implica na impressão com sinal ("+" ou "-")  
" " ==> Implica na impressão com sinal negativo apenas
```

Width : Pode conter os seguintes valores,

```
<n> ==> Ao menos <n> caracteres são impressos, caso o  
comprimento seja menor que <n>, será completado com  
brancos.
```

```
O<n> ==> Caso o comprimento seja menor que <n> completa com zeros.
```

Precision : Pode conter os seguintes valores,

```
.0 ==> Impressão de float sem ponto decimal  
<n> ==> Limita o número de casas decimais a <n>
```

Size : Pode conter os seguintes caracteres,

```
L ==> Implica na definição de um modificador LONG  
H ==> Implica na definição de um modificador SHORT
```

Type : Pode conter os seguintes caracteres,

- D - O argumento é dado na forma inteiro decimal
- O - O argumento é apresentado em octal (inteiro)
- X - O argumento é dado na forma inteiro hexadecimal
- U - O argumento é considerado decimal inteiro sem sinal
- C - O argumento é um caracter único
- S - O argumento é uma cadeia de caracteres
  - E - O argumento é dado como float ou double em notação científica (com expoente)
  - F - O argumento é dado como float ou double em notação decimal comum (sem expoente)
- G - Usar "E" ou "F", conforme a necessidade
- P - O argumento é um pointer (TC)
- % - Reproduz o próprio símbolo "%"

Exemplo :

```
printf ("\nResultado :%3.2f",num);
printf ("Frase : %s\nContador = %x",string);
printf ("a : %-5Db : %-5Dc : %-5D",a,b,c);
```

\* PUTCHAR : Saída padrão de um caracter apenas. Irá escrever apenas um caracter, equivale ao comando : printf ("%c",carac). Sintaxe : putchar (c);

\* PUTS : Saída de uma string na saída padrão. A string será adicionada com um caracter '\n'. Equivale a : printf ("%s\n", string). Sintaxe : puts (string);

\* SCANF : Entrada formatada, através da entrada padrão. Pertence ao grupo de funções de entrada formatada, e atua de forma análoga a saída formatada (printf), mas com sentido inverso. Sintaxe :

```
[nro_itens_lidos =] scanf ("format_string",arg1,arg2,arg3...);
```

Onde :

Format\_string :

- Espaços, caracteres de tabulação e new line serão ignorados
- Caracteres ordinários, são aqueles caracteres que são esperados como próximos caracteres de entrada
- Caracteres especiais de conversão, para a leitura de um tipo determinado de dado. Dados de forma similar aos caracteres definidos na função printf

Argumento : São apontadores para a área de armazenamento, ou seja pointers para variáveis (fornecer o endereço da variável, exceto para arrays !).

\* GETCHAR : Lê um caracter da entrada padrão. Equivale a função putchar, mas em sentido inverso.

Sintaxe :    carac = getchar ( );

\* GETS : Lê uma string da entrada padrão. Análoga a função puts, mas em sentido inverso.

Sintaxe :     string = gets( );

### 13.2 - FUNÇÕES DE MANIPULAÇÃO DE ARQUIVOS :

\* Declaração : Definição de pointers para arquivos

Sintaxe :

```
FILE *fopen( ), *fp;
|       |       |
|       |       |   apontador para arquivo
|       |       |   função que retorna apontador para arquivo
|       |       |   tipo de dado definido no stdio.h
```

A declaração de apontadores e funções de manipulação de arquivos devem ser feitas juntamente com a declaração de variáveis.

\* FOPEN : Abre um arquivo. Os dispositivos console, impressora e saída serial podem ser abertos como arquivos.

Sintaxe :

```
file_pointer = fopen (nome_arquivo, modo);
```

Nome\_arquivo e modo : São pointers para caracteres (strings). Onde nome\_arquivo é o nome atribuído externamente ao "elemento físico" aberto e o modo pode conter um dos seguintes valores :

"r"	- Leitura (open)
"w"	- Escrita (rewrite)
"a"	- Adição (append)
"r+"	- Para atualizar um arquivo (read e write)
"w+"	- Para criar um arquivo para atualização
"a+"	- Adição, em arquivo para leitura e escrita

A função fopen deverá retornar um valor que poderá ser o valor de um apontador para o arquivo ou NULL em caso de erro na abertura do arquivo. Todas as outras operações sobre os arquivos serão referenciadas através deste "apontador de arquivo".

Existem certos tipos de apontadores de arquivos padrão os quais são constantes :

STDIN	- Entrada padrão (normalmente teclado)
STDOUT	- Saída padrão (normalmente vídeo)
STDERR	- Saída padrão de mensagens de erro (Vídeo)

\* GETC : Lê um caracter de um arquivo indicado pelo pointer de arquivo. Retorna o próximo caracter lido do arquivo apontado por file\_pointer. No caso de encontrar o fim do arquivo retorna EOF.

Sintaxe :     carac = getc (file\_pointer);

\* PUTC : Escreve um caracter em um arquivo. Escreve o caracter na próxima posição do arquivo apontado por file\_pointer. É a operação inversa de getc.

Sintaxe :     putc (caracter, file\_pointer);

\* UNGETC : "Devolve" o último caracter lido do arquivo. Devolverá o caracter dado, para o arquivo apontado pelo file\_pointer. Apenas um caracter devolvido será aceito (Coloca em um buffer temporário de um caracter) por cada arquivo.

Sintaxe : ungetc (carac, file\_pointer);

\* FPRINTF : Saída formatada em arquivos. É equivalente a função printf mas com a diferença de que esta atua sobre arquivos, os quais são descritos através do primeiro parâmetro que é um apontador para o arquivo sobre o qual se deseja escrever.

Sintaxe : fprintf (file\_pointer,"controle",arg1,arg2,...);

\* FSCANF : Entrada formatada a partir de arquivos. Atua como a função inversa de fprintf e análoga a função scanf, só que atuando sobre arquivos.

Sintaxe : fscanf (file\_pointer,"controle",arg1,arg2,...);

\* FCLOSE : Fecha um arquivo em uso. Ocasiona o fechamento do arquivo apontado pelo file\_pointer.

Sintaxe : fclose (file\_pointer);

\* Outras funções :

fgets	- Lê uma linha do arquivo (terminada por um '\n')
fputs	- Escreve uma linha em um arquivo
fgetchar	- Lê um caracter de um arquivo
fputchar	- Escreve uma linha em um arquivo
fseek	- Reposiciona o pointer do arquivo dentro deste
fflush	- Esvazia o buffer de e/s do arquivo

\* O arquivo stdio.h além de definir as funções de manipulação de arquivos também contém definições de valores como o EOF, que é uma constante pré-definida neste arquivo.

### 13.3 - FUNÇÕES DE USO GERAL :

Manipulação de tipos de dados :

\* ISUPPER : Verifica se é maiúscula. Retorna diferente de zero se o caracter dado for maiúscula e zero no caso contrário (minúscula). Possui função análoga mas de funcionamento inverso a esta : islower (c).

Sintaxe : valor\_log = isupper (carac)

\* ISALPHA : Verifica se é caracter alfabético. Segue o mesmo princípio de isupper.

Sintaxe : valor\_log = isalpha (carac)

\* Funções análogas :

isdigit (c)	- Verifica se é um dígito
isspace (c)	- Verifica se é caracter de espaço
isascii (c)	- Verifica se é caracter ASCII
isprint (c)	- Verifica se é caracter de impressão

\* TOLOWER : Converte de maiúscula para minúscula. Possui função análoga mas de funcionamento inverso a esta : toupper (c).

Sintaxe : carac = tolower (carac)

Manipulação de strings :

\* STRLEN : Retorna o tamanho de uma string.

Sintaxe : strlen (string); onde "string" é um tipo char \*s

\* STRCPY : Copia uma string em outra.  
Sintaxe :       strcpy (s,t);    onde "s" e "t" são do tipo char \*x

\* STREMP : Compara duas strings.  
Sintaxe :       strem (s,t);    onde "s" e "t" são do tipo char \*x

\* STRCAT : Concatena duas strings.  
Sintaxe :       strcat (s,t);    onde "s" e "t" são do tipo CHAR \*x

\* SPRINTF : Formata argumentos de saída para string. Tem um funcionamento análogo ao da função printf mas com a diferença de a saída não se dar na saída padrão mas sim em uma outra cadeia de caracteres, indicada por nome.

Sintaxe :       sprintf (nome,"controle",arg1,arg2,...);

\* SSCANF : Saída de uma string formatada para variável.  
Sintaxe :       sscanf (nome,"controle",arg1,arg2,...);

Miscelânea de funções :

\* SYSTEM : Executa comando do MS-DOS.  
Sintaxe :       system (cmd);    onde "s" é uma string que contém o comando a ser executado pelo DOS

\* SQRT : Calcula a raiz de um número.  
Sintaxe :       sqrt (x);        onde "x" é um double

\* EXIT : Retorno da execução (Volta para o DOS).  
Sintaxe :       exit (x);        onde "x" é um inteiro (valor de retorno)

\* CALLOC : Alocação dinâmica de espaço de memória. Faz a alocação dinâmica de memória, retornando um apontador para "n" objetos do tamanho especificado, ou retornará NULL se não for possível alocar mais memória.

Sintaxe :       pointer = calloc (n,sizeof(objeto));

\* MALLOC : Alocação dinâmica de espaço de memória. Faz a alocação de um elemento do tamanho especificado. A diferença entre a calloc é que esta função aloca espaço apenas para um elemento enquanto que a outra aloca espaço para "n" elementos.

Sintaxe :       pointer = malloc (sizeof(objeto));

\* FREE : Libera espaços de memória.  
Sintaxe :       cfree (p);    onde "p" é um pointer para um objeto que foi alocado por uma chamada ao calloc

\* Outros grupos de funções :

- Manipulação de diretório : Cd, Md, Rd, ...
- Manipulação de memória : move, set, swap, ...
- Matemáticas : sin, cos, log, ...
- Acesso ao hardware : peek, poke , inport, outport
- Tempo : data, hora

\* Hoje em dia é muito comum encontramos software houses que vendem diferentes tipos de bibliotecas "C" para as mais diversas aplicações :

- Funções de manipulação de Banco de Dados

- Rotinas de comunicação de dados
- Pacotes gráficos

## **APÊNDICE A - COMPILADOR "C" :**

Arquivos padrão utilizados pelo "C" :

<nome>.C - arquivo fonte  
 <nome>.O - arquivo objeto  
 <nome>.EXE - arquivo executável

Compilação :

- No Turbo C temos um ambiente integrado de desenvolvimento de programas onde se encontram reunidos o compilador "C" o linker e um editor de textos.

- No Microsoft C / Lattice, há a necessidade de se executar dois passos de compilação (MC1 e MC2), e a ligação (Link). Sugerimos o uso de discos virtuais para o trabalho com este tipo de compilador e a compilação pode se dar com o uso de um programa ".BAT" como o listado abaixo :

```

echo off
mc1 %1
if errorlevel 1 goto msg1
mc2 %1
if errorlevel 1 goto msg2
link cs+%1,%1,nul,mcs
if errorlevel 1 goto msg3
goto fim
:msg1
echo **** Erro no primeiro passo da compilacao ****
goto fim
:msg2
echo **** Erro no segundo passo da compilacao ****
goto fim
:msg3
echo **** Erro na linkagem do programa ****
:fim
  
```

- Em ambos os compiladores temos os seguintes arquivos :

Compilador (TC.COM ou MC1 e MC2)  
 Linker (Link)  
 Biblioteca de funções (Arquivos com extensão ".H")  
 Biblioteca de objetos (Arquivos com extensão ".OBJ")

- O Microsoft C é encontrado em diferentes versões :

Microsoft 3.0 / Lattice  
 Microsoft 4.0 + Codeview (depurador)  
 Microsoft 5.0

## **APÊNDICE B - PADRÃO K&R X ANSI :**

Ao ser criada a linguagem "C" por Kernighan e Ritchie, foi definido um padrão, padrão este que se encontra descrito no livro "base" para a programação em "C", escrito por K&R. Mas após a disseminação do uso do "C" e tornou necessária uma melhor padronização e criação de outras estruturas mais flexíveis, isto se deu através da padronização ANSI. Este padrão é mais recente e alguns compiladores ainda não o seguem, mas a tendência é que todos os compiladores "C" passem a seguir o padrão ANSI.

Vamos aqui descrever as principais diferenças entre o padrão K&R em relação ao padrão ANSI, citando vantagens e desvantagens apresentada por ambos :

\* Function Prototypes : São uma forma mais eficiente de declaração de funções, pois permitem ao compilador fazer uma verificação mais consistente a respeito das funções e seus parâmetros de entrada. Exemplos de declarações de função :

```
char *myfunct ( );                ==> K&R
char *myfunct (int arg1,char arg2,char *arg3);  ==> ANSI
```

Com o uso do "prototype" uma função poderá ser chamada no programa antes de ser definida, mas mesmo assim o compilador terá condições de informar ao programador quanto a erros referentes a passagem de parâmetros de tipo incompatível e em número insuficiente ou superior ao número real de parâmetros necessários a função. Assim como a declaração é feita com o uso desta notação podemos ter a definição desta forma :

```
char *myfunct ( )                ==> K&R
int arg1;
char arg2,*arg3;
{ ... }

char *myfunct (int arg1,char arg2,char *arg3)  ==> ANSI
{ ... }
```

\* Enumeração : O padrão ANSI permite a declaração de um dado do tipo "enum" (enumeração), que apesar de também poder ser obtido por vias indiretas no padrão K&R, permite uma maior facilidade de uso e melhor documentação de programas. Exemplos de uso de enumerações :

```
#define sun 0                    ==> K&R
#define mon 1
#define tues 2
#define wed 3
#define thur 4
#define fri 5
#define sat 6

typedef enum (sun, mon, tues, wed, thur, fri, sat ) days;
days todays,date;              ==> ANSI
```

\* Constantes : Com o modelo K&R só podemos declarar constantes com o uso da macro #define, que fará a substituição dos itens declarados nele pelos seus respectivos valores. No padrão ANSI temos a possibilidade de declarar um novo tipo de dado - o "const". As variáveis declaradas como sendo "const" não poderão sofrer modificações como : incremento, decremento, atribuição, ... A diferença principal entre constantes e defines é que as constantes estão alocadas em memória, como as variáveis. Exemplos :

```
#define PI 3.1415926             ==> K&R
const float pi = 3.1415926;     ==> ANSI
```

As constantes serão protegidas contra os acessos indevidos, mas é claro que através de pointers podemos modificar seus valores e não teremos como evitar. Por exemplo :

```
pi = 3.0;          /* Inválido : atribuição a constante */
pi = pi++;        /* Inválido : incremento de contante */
str = "Hello";    /* Inválido : atribuição de novo end. */
strcpy(str,"Hello") /* Válido : alteração por pointers ! */
```

\* Itens constantes : O padrão ANSI aceita as seguintes estruturas abaixo que não são aceitas pelo padrão K&R.

```
msg = "Primeira frase." /* Definir strings que ocupem */
"Segunda frase."       /* várias linhas ... */
'\x0a'                 /* Sequência de escape em Hexa */
```

\* Declaração Void : O tipo de dado "void", usado somente nas declarações e definições de funções serve para identificar um tipo nulo. Há dois significados para o "void" : não há retorno de parâmetros pela função ou a função retorna um valor genérico (pointer para qualquer tipo de dado). Exemplos de uso do "void" :

```
void myfunct( );    /* Myfunct não retorna valores */
int Myfunct(void); /* Myfunct não possui parâmetros */
void *Myfunct( ); /* Retorna pointer genérico */
```

\* Modificador Signed : O padrão ANSI permite além do modificador "unsigned" um outro modificador equivalente a este, mas de sentido contrário, é o "signed", que indica que o dado é do tipo com sinal.

\* Modificador Volatile : Definido pelo padrão ANSI serve para indicar o oposto do "const", pois esta variável poderá ser modificada a qualquer momento, não apenas por você, mas por algo "externo", como uma rotina de interrupção ou porta de I/O. Esta declaração previne o compilador para evitar o uso desta variável em um registrador. Exemplo :

```
volatile int ticks;
timer ( ) /* Rotina de atendimento da */
{ /* interrupção do timer */
    ticks++;
}
```

## **APÊNDICE C - ERROS COMUNS EM "C" :**

- Falta de parâmetro em uma declaração de função ou passagem incorreta de parâmetros para funções ou procedures. A passagem de parâmetros a mais ou a menos pode ter graves consequências.

- Igualdades mal escritas nas comparações (IF). Exemplo :IF (X=Y) significa que X receberá o valor de Y e após será testado o valor deste, este tipo de erro não é percebido pois não é identificado como um erro de sintaxe, o correto seria : IF (X == Y) ...

- Erro na definição de arrays e uso de subscritos fora da faixa permitida. Exemplo :

```
int matriz[10] -> Matriz de inteiros com índices de 0 a 9
                São válidos matriz[0] a matriz[9]
```

O elemento matriz[10] não existe !!!

- Esquecimento do BREAK após cada comando dentro de um comando SWITCH/CASE (principalmente se tiver o "default").

- Manipulação de strings somente por funções, ou seja não é permitida a atribuição direta de um string a uma variável do tipo array de caracteres. Exemplo :

```
char var[80];  
var = "texto"; - Está incorreto  
strcpy (var,"texto"); - É o comando certo
```

- Não confundir os diferentes tipos de dados em "C" :

"A" - String composta pelo caracter A. Representada internamente como um pointer para um endereço que contém a letra A.

'A' - Constante equivalente ao ascii da letra A. Representa apenas o caracter.

9 - Número nove nunca confundir com o caracter '9' que tem o valor ascii correspondente a este caracter.

- Sempre deixar uma posição reservada para o '\0' nas strings (arrays de caracteres). Se formos definir uma variável para conter uma string de dez letras devemos dimensioná-la como :

```
char var[11] - Lembre-se que valem as posições de 0 a 10
```

- Nunca esqueça de na passagem de parâmetros para a função da biblioteca SCANF fornecer o endereço da variável como parâmetro. E cuidado, pois um array já é um endereço !

```
scanf ("%c",letra) - Incorreto  
scanf ("%c",&letra) - Correto  
scanf ("%s",string) - Correto  
scanf ("%s",&string) - Incorreto
```

- O erro acima citado é muito comum e pode ser generalizado como sendo a passagem incorreta de parâmetros, passando dados como endereços e endereços como dados.

- Um erro muito comum é o de após a definição do nome de uma função ou procedure incluirmos um ';', o que irá gerar um erro de sintaxe. Exemplo :

```
main (); - Está incorreto  
{ ... }
```

- Nomes de subdiretórios escritos incorretamente devido a confusão causada pelo caracter de escape '\\.

Exemplo :

```
"C:\\MEUDIR\\MEUARQ"
```

- Uso de pointers : Pointers não inicializados, que podem causar danos a área de dados, em atribuições incorretas.

Exemplo :

```
int *ptr;  
*ptr = valor;
```

- Um pointer para caracter não aloca área, portanto não pode ser usado em comandos como scanf ou strcpy, sem a devida alocação da memória necessária.

Erros de programadores Pascal :

- Omissão do parênteses em funções sem parâmetros.
- Indexação de arrays multidimensionais (a[i][j] e não a[i,j]).
- Esquecimento que "C" considera para nomes de variáveis tanto as maiúsculas quanto as minúsculas.
- Colocar ";" em excesso ao final de blocos e definição de funções.

#### **APÊNDICE D - EXEMPLO DE PROGRAMA :**

```
#include <stdio.h>

struct nodo {
    char texto [80];
    struct nodo *prox;
};

struct nodo *novonodo;
struct nodo *raiz;
struct nodo *nodoptr;

char textaux [70];
int continua;

main( )
{
    continua = 1;
    raiz = (struct nodo *)malloc(sizeof(struct nodo));
    printf ("\n\n*** CRIACAO DE UMA LISTA ENCADEADA ***\n\n");
    strcpy (raiz->texto,"<<<< RAIZ DA ESTRUTURA >>>>");
    raiz->prox = '\0';
    nodoptr = raiz;
    printf ("Entre com o Texto : ");
    gets (textaux);
    while (continua == 1)
    {
        novonodo = (struct nodo *)malloc(sizeof(struct nodo));
        strcpy (novonodo->texto,textaux);
        novonodo->prox = '\0';
        nodoptr->prox = novonodo;
        nodoptr = novonodo;
        printf ("Entre com o texto : ");
        gets (textaux);
        if (strcmp(textaux,"FIM") == 0)
            continua = 0;
    }
    printf ("\n\n>>> FIM DA ENTRADA DE DADOS");
    printf ("\n\n\n*** LISTAGEM DA ESTRUTURA DE DADOS ***\n\n");
    nodoptr = raiz;
    printf ("%s\n",nodoptr->texto);
    while (nodoptr->prox != '\0')
```

```

    {
        nodoptr = nodoptr->prox;
        printf ("%s\n",nodoptr->texto);
    }
    printf ("\n\n>>> FIM DA LISTAGEM DE DADOS\n\n");
}

```

## APÊNDICE E - EXEMPLO DE HEADER :

```

/**
 *
 * This header file defines the information used by the standard I/O
 * package.
 *
 **/
#define _BUFSIZ 512          /* standard buffer size */
#define BUFSIZ 512          /* standard buffer size */
#define _NFILE 20          /* maximum number of files */

struct _iobuf
{
    char *_ptr;              /* current buffer pointer */
    int _rcnt;              /* current byte count for reading */
    int _wcnt;              /* current byte count for writing */
    char *_base;            /* base address of I/O buffer */
    char _flag;             /* control flags */
    char _file;             /* file number */
    int _size;              /* size of buffer */
    char _cbuf;             /* single char buffer */
    char _pad;              /* (pad to even number of bytes) */
};

extern struct _iobuf _iob[_NFILE];

#define _IOREAD 1          /* read flag */
#define _IOWRT 2          /* write flag */
#define _IONBF 4          /* non-buffered flag */
#define _IOMYBUF 8        /* private buffer flag */
#define _IOEOF 16         /* end-of-file flag */
#define _IOERR 32         /* error flag */
#define _IOSTRG 64        /* read-write (update) flag */
#define _IORW 128         /* read-write (update) flag */

#define NULL 0             /* null pointer value */
#define FILE struct _iobuf /* shorthand */
#define EOF (-1)          /* end-of-file code */

#define stdin (&_iob[0]) /* standard input file pointer */
#define stdout (&_iob[1]) /* standard output file pointer */
#define stderr (&_iob[2]) /* standard error file pointer */

#define getc(p) (--(p)->_rcnt>=0? *(p)->_ptr++:_filbf(p))
#define getchar() getc(stdin)
#define putc(c,p) (--(p)->_wcnt>=0? ((int)*(p)->_ptr++=(c)):_flsbf((c),p))
#define putchar(c) putc(c,stdout)
#define feof(p) (((p)->_flag&_IOEOF)!=0)

```

```
#define ferror(p) (((p)->_flag&_IOERR)!=0)
#define fileno(p) (p)->_file
#define rewind(fp) fseek(fp,0L,0)
#define fflush(fp) _flsbf(-1,fp)
```

```
FILE *fopen( );
FILE *freopen( );
long ftell( );
char *fgets( );
```

```
#define abs(x) ((x)<0?-x):(x)
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<=(b)?(a):(b))
```

## **BIBLIOGRAFIA**

- 1 - The "C" Programming Language  
Brian W. Kernighan & Dennis M. Ritchie  
Englewood Cliffs. Prentice - Hall, 1978.  
(Edição em português - Editora Campus)
- 2 - Turbo C User's Guide  
Borland International Inc. California - USA, 1987.
- 3 - Turbo C Reference Guide  
Borland International Inc. California - USA, 1987.
- 4 - Linguagem "C" - Guia do Usuário  
Hebert Schildt  
Ed. McGraw-Hill, 1985
- 5 - Microsoft C compiler - Reference Manual  
Microsoft Corporation, 1983.