FUNDAÇÃO EDUCACIONAL DE ITUIUTABA
ASSOCIADA À UNIVERSIDADE DO ESTADO DE MINAS GERAIS
CURSO DE **SISTEMA DE INFORMAÇÃO**

CURSO BÁSICO DE PROGRAMAÇÃO EM TURBO C

Prof. Walteno Martins Parreira Júnior

www.waltenomartins.com.br

waltenomartins@yahoo.com

2011

SUMÁRIO

1 – DEFINIÇÕES BÁSICAS

Programa – Conjunto de instruções distribuídas de maneira lógica, com a finalidade de executar satisfatoriamente determinada tarefa .

Linguagem de Programação – Conjunto de instruções possíveis utilizadas pelo homem para se comunicar com a máquina.

Endereço de memória – Número de uma posição de memória.

Compilador – Programa que traduz programas em linguagem de alto nível para linguagem de máquina.

Erro de compilação – Erro no programa em linguagem de alto nível que é detectado pelo compilador.

Erro de execução – Erro cometido em um programa que não é detectado até que o programa seja executado.

Variável – Símbolo que representa uma posição de memória.

Ponteiros – Tipos de variáveis que nos permite manipular endereços de outras variáveis.

1.1 – INTRODUÇÃO E COMANDOS BÁSICOS

A linguagem C assim como Pascal é uma linguagem estruturada.

Uma linguagem é chamada estruturada quando é formada por blocos chamados de funções. Um programa em C é formado por uma coleção de funções. Em um programa bem escrito cada função executa apenas uma tarefa. Cada função tem um nome e uma lista de argumentos que a mesma receberá.

A execução do programa escrito em C sempre começa pela função principal **main( ).**

1.1.1 – Exemplo

```
#INCLUDE <STDIO.H>
#INCLUDE <CONIO.H>
void  main( )
{
        int   idade ;
        clrscr( );
        printf (" Digite  sua  idade ");
        scanf (" %d",&idade );
        printf ( "Sua  idade  é  %d" ,idade );
        getch( );
}
```

Analisando cada linha do programa:

```
#INCLUDE <STDIO.H> → especifica a biblioteca que deverá ser
#INCLUDE <CONIO.H>    ligada quando da compilação do programa
void main( )  → especifica o nome e o tipo da função
                  (nesse caso void)
{              → inicio da função main
int   idade    → declara uma variável de nome idade e tipo
inteiro
clrscr( )      → função predefinida para limpar a tela
printf(" Digite sua idade ")  → imprime a mensagem entre
aspas na tela
scanf(" %d",&idade )   → lê via teclado um valor que é
                         colocado na variável idade
getch( )               → função predefinida, espera uma tecla
                         ser pressionada
}                      → fim da função main
```

1.2 – VARIÁVEIS

Uma variável é um espaço de memória reservado para armazenar um certo tipo de dado e tendo um nome para referenciar o seu conteúdo.

Duas variáveis globais não podem ter o mesmo nome, uma variável local pode ter o mesmo nome de uma variável local de outra função.

1.2.1 – DECLARANDO VARIAVEIS

tipo lista_variaveis;

TABELA CONTENDO OS TIPOS E TAMANHOS DE VARIAVEIS VÁLIDAS EM C

TIPO	EXTENSAO DO BIT	ESCALA
char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
long int	32	-2147483648 a 147483648
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308

** ver item 6.4 para os códigos de formatação

1.2.2 – INICIALIZANDO VARIAVEIS

Inicializar uma variável significa atribuir um primeiro valor a essa variável. Variáveis globais são sempre inicializadas com zero. Exemplo:

```
int k = 5 ;
char op = 'f';
float num = 21.5;
char nome[20] = "Fernanda";
```

FUNÇAO *printf( )*

- Função predefinida no arquivo STDIO.H, e serve para imprimirmos um determinado dado na tela.
- Sintaxe

```
printf("string de controle",variavel);
```

- Exemplo :

```
int k=25;
printf("%i",k);
```

Códigos que podem ser utilizados para formatação da impressão

Código	Significado
\n	Nova linha (LF)
\a	Alerta (beep)
\f	Nova tela ou nova página (FF)
\t	Tab
\b	Retrocesso (BS)
\0	Nulo
\"	Aspas duplas
\'	Aspas simples
\\	Barra invertida

FUNCAO *scanf( )*

- Função predefinida no arquivo STDIO.H, e serve para ler um determinado dado (valor) via teclado.
- Sintaxe

```
scanf("string de controle",&variavel);
```

- Exemplo :

```
char op;
scanf("%c",&op);
```

1.2.3 – Exemplo

```
#INCLUDE <STDIO.H>
#INCLUDE <CONIO.H>
void  main( )
{
      int   k = 5;
      int p, q;
      clrscr( );
      printf ( "Sua o valor %i" ,q );
      scanf (" %i",&q );
      p = q * k;
      printf ( "O valor calculado  é  %i" ,p );
      getch( );
}
```

1.2.4 – TABELA CONTENDO OS TIPOS E TAMANHOS DE VARIAVEIS VÁLIDAS EM C

TIPO	EXTENSAO DO BIT	ESCALA
char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
long int	32	-2147483648 a 2147483648
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308

1.3 – ALGUNS TERMOS COMUNS

Tempo de compilação: Os eventos ocorrem enquanto seu programa está sendo compilado .

Tempo de execução: Os eventos ocorrem quando seu programa está sendo executado.

Biblioteca: É um arquivo contendo as funções padrão que seu programa poderá usar.

Código fonte: É o texto de um programa que o usuário pode ler, comumente chamado de programa.

1.4 – Regras Gerais de um programa em C

- Toda função deve ser iniciada por uma chave de abertura ({) e encerrada por uma chave de fechamento (}).
- Toda função apresenta um par de parêntesis após o seu nome.
- Todo programa deverá conter a função main.
- As linhas de código sempre terminam com um ponto e virgula.
- A formatação do programa é livre, mas é conveniente uma identação mínima para manter a legibilidade.
- Os comandos são executados na sequencia em que foram escritos.
- Os comentários são delimitados por (/*) no início e (*/) no final.
- Para comentários que não ultrapassam uma linha podem ser utilizados (//).

1.5 – FUNÇOES DE LEITURA VIA TECLADO

FUNCAO	OPERACAO
getchar ( )	lê um caractere ; espera por <enter>
getche ( )	lê um caractere com eco; não espera por <enter>
getch ( )	lê um caractere sem eco; não espera por <enter>
putchar ( )	imprime um caractere na tela
gets ( )	lê uma string via teclado
puts ( )	imprime uma string na tela

1.6 – CODIGOS DE FORMATAÇÃO PARA PRINTF E SCANF

CODIGO	PRINTF	SCANF
%d	imprime um inteiro decimal	lê um inteiro decimal
%f	Imprime ponto decimal flutuante	lê um número com ponto flutuante
%s	Imprime string de caracteres	lê uma string de caracteres
%c	Imprime um único caractere	lê um único caractere
%i	Imprime decimal	lê um inteiro decimal
%p	imprime um ponteiro	lê um ponteiro
%e	Imprime em notação cientifica	lê um número com ponto flutuante

1.6.1 – Exercício

Faça um programa que leia dois números e em seguida mostre, o produto, a soma e a subtração entre eles.

2 – OPERADORES

C é uma linguagem rica em operadores. Alguns são mais usados do que outros como é o caso do operador de atribuição e dos operadores aritméticos, a seguir mostramos tabelas contendo os operadores aritméticos, relacionais e lógicos.

2.1 – OPERADORES MATEMÁTICOS

OPERADOR	AÇÃO
-	subtração
+	adição
*	multiplicação
/	divisão
%	resto da divisão
--	decremento
++	incremento

2.2 – OPERADORES LOGICOS

OPERADOR	AÇÃO
&&	and
| |	or
!	not

2.3 – OPERADORES RELACIONAIS

OPERADOR	ACAO
>	maior que
<	menor que
>=	maior ou igual
<=	menor ou igual
==	igual a
!=	diferente de

2.4 – FUNCOES

Um programa em C é formado por um conjunto de funções.
- Declarando uma função:

```
tipo identificador(lista de parâmetros)
{
     declaração de variáveis locais;
     comando ou bloco de comandos;
}
```

2.4.1 – Exemplo

```
void quadrado(int p)
{
        int k;
        k = p*p;
        printf("%i",k);
}
void main( )
{
        int k=25;
        quadrado(k);
        getch( );
}
```

2.4.2 – COMANDO *return*

Serve para retornarmos um valor calculado dentro de uma função quando chamada de alguma parte do programa.

2.4.3 – Exemplo:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
float calc_sin(float arg)
        {
                float val;
                printf ("Valor recebido: %f",arg);
                val = sin(arg);
                return(val);
}
void main( )
{
        clrscr ( );
        float valor;
        valor = calc_sin(50);
        printf("  Valor calculado: %f",valor);
        getch ( );
        }
```

3 – COMANDOS DE DECISAO

Os comandos de decisão permitem determinar qual é a ação a ser tomada com base no resultado de uma expressão condicional.

3.1 – COMANDO IF–ELSE

O comando if instrui o computador a tomar uma decisão simples.
Forma geral:

```
if  ( condição )  comando ;
    else comando ;
```

3.1.1 – Exemplo

```
/*programa do numero magico */
#include <stdio.h>
#include <conio.h>
void main( )
{
 int magico , entrada;
 magico = random(20); //gera um nro entre 0 e 20
 clrscr( );
  printf( "Adivinhe o numero : ");
  scanf("%d",&entrada);
  if (entrada == magico) printf("== Você acertou ==");
     else printf("Você não acertou pressione qualquer tecla);
  getch( );
   }
```

3.2 – COMANDO SWITCH CASE

O comando switch pode ser usado no caso de alternativas múltiplas.
Forma geral:

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

O comando switch ao avaliar a expressão entre parênteses, desviamos para o rótulo case cujo valor coincida com o valor da expressão. O comando break serve para sairmos do bloco mais interno ao qual o break aparece. O comando break garante a execução de apenas uma chamada dentro do switch.

3.2.1 – Exemplo

```
#include <stdio.h>
void main ( )
{
        char opção;
        clrscr( );
        printf("A - imprime a letra f");
        printf("B - imprime a letra g");
        printf("C - imprime a letra h");
        opção = getch( ) ;
        switch(opção)
        {
                case 'a' : printf("f");break;
                case 'b': printf("g");break;
                case 'c' : printf("h");break;
        }
}
```

3.3 – LISTA DE EXERCÍCIOS

3.3.1 – Faça um programa contendo um menu com as seguintes opções:

S – soma
P – produto
U – subtração
D – divisão
Q – sair

O programa deve conter uma função para executar cada tarefa solicitada: soma, subtração, etc. Quando o usuário teclar ESC o programa deve terminar.

3.3.2 – Desenvolver um programa que:

a) leia dois número inteiros e diferentes;
b) imprima o maior entre os dois números;
c) Subtrair o menor do maior número e imprimir o resultado;
d) Dizer se o primeiro número é impar ou par;

3.3.3 – Escreva um programa onde o usuário entra com um número qualquer e o programa responde se o número é par ou impar.

Se for par emite a mensagem " O número é par " ou caso contrário "O número é impar ".

3.3.4 – Escreva um programa dois números diferentes e o programa responde qual é o maior dos dois. Depois, o programa testa o primeiro número lido e emite a mensagem " O número é par " ou caso contrário "O número é impar ". E para terminar, o programa imprime todos os números compreendidos entre o menor e o maior número lido

4 – LAÇOS

Laços são comandos da linguagem C úteis sempre que uma ou mais instruções devam ser repetidas enquanto uma certa condição estiver sendo satisfeita.

4.1 – LAÇO FOR

O laço for é geralmente usado quando queremos repetir algo um número fixo de vezes. Isto significa que utilizamos um laço for quando sabemos de antemão o número de vezes a ser repetido.

Forma geral:

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

4.1.1 – Exemplo

```
/* programa que imprime os números de 1 a 100 */
#include <stdio.h>
#include <conio.h>
void  main ( )
{
      int x;
      for ( x=1;x<=100;x++)
      printf ( "%d",x);
      getch( ) ;
}
```

4.1.2 – Exemplo

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void raiz( float n)
  {
  printf("\nn = %f  raiz quadrada = %f",n,sqrt(n));
  }
void main( )
  {
      float num;
      for (num=1;num<20;num++)  raiz(num);
}
```

4.2 – LAÇO WHILE

Um laço while é apropriado para situações em que o laço pode ser terminado inesperadamente, por condições desenvolvidas dentro do laço.

Forma geral:

```
while ( expressão de teste)  comando ;
```

4.2.1 – Exemplo:

```
void  imprime(char op)
```

```
{
        int k=0;
        while ( k != 50 )
        {
                if (op==’p’)
                   if (k%2==0)  printf(“%i”,k);
                if (op==’i’)
                    if (k%2!=0)  printf(“%i”,k);
                k++;
        }
}
```

4.2.2 – LAÇO DO–WHILE

Este laço é bastante similar ao laço while e é utilizado em situações em que é necessário executar o corpo do laço pelo menos uma vez e depois avaliar a expressão de teste.

Forma geral:

```
do
{
      comando ou bloco de comandos;

}
while(expressão de teste);
```

4.2.3 – Exemplo :

```
void main( )
{
        char op;
        int sair = 0;
        do
        {
                op = getche( );
                switch(op)
                {
                        case ‘s’ : somatorio( );break;
                        case ‘f’  : fibonacci( );break;
                        case ‘q’ : sair = 1;
                }
        }while(sair!=1);
}
```

4.3 – LISTA DE EXERCÍCIOS

4.3.1 – Suponha um número N qualquer

se N é par então N agora é N / 2

se N é ímpar então N agora é 3*N + 1

Assim para N = 3 calculamos a seguinte tabela:

3 → 10	4 → 2
10 → 5	2 → 1
5 → 16	1 → 4
16 → 8	4 → 2

8 → 4 2 → 1

Observe que a partir de sete iterações a seqüência 4 2 1 começa a se repetir. Faça um programa que calcule para um dado N o número de iterações até se chegar ao primeiro 1.

4.3.2 – Faça um programa que imprima os elementos de uma PA e o somatório dos mesmos dados: primeiro termo, número de termos e razão

4.3.3 – Faça um programa que imprima um elemento da seqüência de Fibonacci, dado o número do elemento.

4.3.4 – Faça um programa onde o usuário entra com um número decimal e ele calcula e imprime o número no sistema binário.

4.3.5 – Escreva um programa onde o usuário digita um caracter do teclado e ele imprime o caracter e seu código ASCII, até que o usuário tecle ESC.

OBS: Código Ascii da tecla ESC = 27;

4.3.6 – Faça um programa onde o usuário entra com dois números A e B o programa devolve como resultado A elevado a B.

4.3.7 – Escreva um programa que solicite ao usuário três números inteiros a,b,c onde a é maior que 1. Seu programa deve somar todos os inteiros entre b e c divisíveis por a.

4.3.8 – escreva um programa que lê dois números e imprime qual o maior deles. Após a impressão, o programa pergunta se deseja realizar um novo cálculo e:

a) se a resposta for sim, procede a leitura dos valores e imprime qual dos dois é o maior;

b) se a resposta for não, termina a execução do programa.

5 – FUNÇÕES

5.1 – DEFININDO UMA FUNÇÃO

Uma função é um conjunto de instruções organizadas para cumprirem uma tarefa e que é agrupada em uma unidade com um nome para ser referenciada.

O uso de funções reduz o tamanho do programa, pois qualquer seqüência que aparece mais de uma vez no programa é candidata a ser uma função.

Um programa pode ter uma ou mais funções, sendo que uma delas tem que ser a função principal denominada **main()** e a execução sempre começa por ela. Quando o controle do programa encontra uma instrução que inclui o nome de uma função, a função é chamada e o controle é transferido para a função e passa a executar as instruções dela. Ao terminar a execução da função, o controle retorna a posição seguinte a chamada da função.

5.1.1 – Padrão de uma função

Uma função no C tem a seguinte forma geral:

```
Tipo-de-retorno nome-da-função (declaração-de-parâmetros)
{
   corpo-da-função
}
```

Explicando as partes. a) O **tipo-de-retorno** é o tipo de variável que a função vai retornar. O padrão é o tipo **int**, ou seja, uma função para qual não é declarado o tipo de retorno é considerado como retornando um inteiro. b) A **declaração-de-parâmetros** é uma lista da seguinte forma geral:

```
tipo nome1, tipo nome2, ... , tipo nomeN
```

Observe que o tipo das variáveis deve ser especificado para cada uma das N variáveis de entrada. A declaração de parâmetros que informa para o compilador quais serão as entradas da função (assim como é informado a saída no tipo de retorno). c) O **corpo-da-função** é nele que as entradas são processadas, as saídas são geradas ou outras ações são feitas.

5.1.2 – O Comando return

O comando **return** tem a seguinte forma geral:

```
return valor-de-retorno; ou return;
```

Supondo que uma função está sendo executada. Quando se chega a uma declaração return a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor. Deve-se lembrar que o valor de retorno fornecido tem que ser, pelo menos, compatível com o tipo de retorno declarado para a função. Uma função pode ter mais de uma declaração return em seu corpo.

a) Primeiro exemplo:

```
#include <stdio.h>
int Square (int a)
{
      return (a*a);
}
main ()
{
      int num;
      printf ("Entre com um numero: ");
      scanf ("%d",&num);
      num=Square(num);
      printf ("\n\nO quadrado vale: %d\n",num);
}
```

b) Segundo exemplo:

```
#include <stdio.h>
int EPar (int a)
{
      if (a%2) /* Verifica se a e divisível por dois */
            return 0; /* Retorna 0 se não for divisível */
      else
            return 1; /* Retorna 1 se for divisível */
}
main ()
{
      int num;
      printf ("Entre com numero: ");
      scanf ("%d",&num);
      if (EPar(num))
            printf ("\n\nO numero e par.\n");
      else
            printf ("\n\nO numero e impar.\n");
}
```

5.1.3 – O Tipo void

Em inglês, void quer dizer vazio e é isto mesmo que o void é. Ele permite fazer funções que não retornam nada e funções que não têm parâmetros. Pode-se escrever o protótipo de uma função que não retorna nada:

void nome-da-função (declaração-de-parâmetros);

Numa função, como a acima, não se tem valor de retorno na declaração return. neste caso, o comando return não é necessita aparecer no corpo da função.

Pode-se, fazer funções que não têm parâmetros:

Tipo-de-retorno nome-da-função (void);

ou, que não tem parâmetros e não retornam nada:

void nome_da_função (void);

a) Um exemplo de função que usa o tipo void:

```
#include <stdio.h>
void Mensagem (void);
main ()
{
      Mensagem();
      printf ("\tEscreva novo:\n");
      Mensagem();
}
void Mensagem (void)
{
      printf ("Ola! Estou escrevendo.\n");
}
```

Observe que a função está colocada depois de **main(),** mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente. Usando protótipos pode-se construir funções que retornam quaisquer tipos de variáveis. Eles não só ajudam o compilador. Eles ajudam a entender melhor o programa. Usando protótipos o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado.

5.2 – FUNÇÕES RECURSIVAS

Uma função é dita recursiva quando se é definida dentro dela mesma. Isto é, uma função é recursiva quando dentro dela está presente uma instrução de chamada a ela própria.

5.1 – Exemplo

```
// imprime uma frase invertida. Usa recursão
#include <stdio.h>
#include <conio.h>
void inverte( )
void main( )
{
      clrscr( );
      scanf( “%c”,’\n’);
      inverte( );
      getch();
}
void inverte ( )
{
      char ch ;
      if ((ch=getche( )) != ‘\r’ ) inverte( );
      scanf(“%c”,ch)
}
```

6 – STRINGS, MATRIZES E VETORES

Uma matriz é uma coleção de variáveis do mesmo tipo que são referenciadas pelo mesmo nome.

A forma geral de se declarar uma matriz unidimensional é:
tipo nome_var[*tamanho*];

6.1 – STRING

Strings são matrizes unidimensionais de caracteres sempre terminada em zero '\0'. Ex: char str[11];

São funções pré-definidas para se trabalhar com strings.

Nome	Função
gets(str)	lê a string str via teclado
puts(str)	imprime string str
strcpy(s1 , s2)	copia o conteúdo de s2 para s1
strcat(s1, s2)	anexa s2 ao final de s1
strcmp(s1,s2)	retorna 0 se as duas strings forem iguais
strlen(str)	calcula e retorna o comprimento de str

6.2 – VETOR

O vetor permite a construção de um tipo cujos valores são agregados homogêneos de um tamanho definido, isto é, seus componentes são todos de um mesmo tipo. O formato de definição para vetor é o seguinte:

tipo nome_do_vetor = vetor [limite_inferior.. limite_superior] de tipo;

onde limite_inferior e limite_superior são constantes do tipo inteiro e tipo é o tipo das componentes. Exemplo:

tipo VET_ALUNOS = vetor [1..10] de caractere;

Pode-se criar um tipo, onde os elementos são compostos por 10 nomes. Um elemento deste tipo pode conter os nomes dos 10 alunos de uma classe. Supondo que uma variável TURMA do tipo VET_ALUNOS possua os seguintes valores:

Ana	Pedro	Paulo	Carla	José	João	Maria	Cláudia	Mário	Iara
1	2	3	4	5	6	7	8	9	10

Cada elemento em um vetor possui um índice que indica sua posição dentro deste vetor. Caso queira referenciar o item **Carla** usa-se a seguinte notação:

TURMA[4]

O conteúdo deste item é Carla.

6.3 – MATRIZ

A matriz, assim como o vetor, permite a construção de um tipo onde os valores são agregados homogêneos de um tamanho definido, isto é, seus componentes são todos de um mesmo tipo. O formato de definição para vetor é:

tipo nome_da_matriz = matriz [lim_inf_1..lim_sup_1; lim_inf_2..lim_sup_2] de tipo;

onde lim_inf_1, lim_inf_2, lim_sup_1 e lim_sup_2 são constantes do tipo inteiro e tipo é o tipo das componentes. Exemplo:

tipo MAT_CLASSE = matriz [1..3; 1..10] de caractere;

Pode-se criar um tipo onde os elementos são compostos por 30 nomes. Um elemento deste tipo pode conter o nome dos 10 alunos de cada uma das 3 classes de uma escola.

Supondo que uma variável TURMA_QUINTA do tipo MAT_CLASSE possua os seguintes valores:

	1	2	3	4	5	6	7	8	9	10
1	Ana	Pedro	Paulo	Carla	José	João	Maria	Cláudia	Mário	Iara
2	Juca	Roger	Ivo	Cristine	Joel	Márcio	Márcia	Sara	Denise	Carlos
3	Lucy	Darci	Lara	Osmar	Daniel	Diogo	Analice	Cris	Josy	Julia

Cada elemento numa matriz é referencial por dois índices, que indicam sua posição dentro da matriz. Caso queira referenciar o item **Márcia**, pode-se usar a seguinte notação:

TURMA_QUINTA [2, 7]

O conteúdo deste elemento é Márcia.

Exemplo de manipulação de vetor:

```
int V[10];
int i, j, aux;
{
    int i, j, aux;
    for (i = 1; i < n; i++)
       for (j = n - 1; j >= i; j--)
            if (v[j-1] > v[j])
            {
               aux = v[j-1];
               v[j-1] = v[j];
               v[j] = aux;
            }
}
```

6.4 – INICIALIZAÇÃO DE VETORES E MATRIZES

Em C pode-se inicializar Vetores e Matrizes globais. Não pode é inicializar matrizes locais.
Exemplo de inicialização de matrizes:

```
int  mat[8]={5,2,1,5,4,5,4,7};
int  sqr[2][3]={
        2,3,6,
        4,5,5,
         };
char str[80] = “Linguagem  C”;
```

6.5 – Lista de Exercícios

6.5.1 – Faça um programa que leia uma senha digitada. Se a senha for correta imprime uma mensagem e se for falsa imprime outra mensagem.

6.5.2 – Escreva um programa que leia uma string e imprime a mesma de trás para frente.

6.5.3 – Construa um programa em que se entra com uma determinada quantidade de nomes de alunos. Depois da entrada dos nomes você digita o número do aluno e o programa mostra o nome do mesmo.

6.5.4 – Faça um programa onde o usuário entra com a ordem da matriz, o programa deve ler a matriz do tipo inteiro e imprimir a mesma.

6.5.5 – Faça uma programa que calcule a matriz transposta e oposta de uma matriz digitada pelo usuário.

6.5.6 – Escreva um programa que verifica a identidade de duas matrizes de mesma ordem.

7 –TIPOS DE DADOS DEFINIDOS PELO USUARIO

7.1 – ESTRUTURAS

Estrutura é uma coleção de variáveis referenciadas por um nome. As variáveis que formam a estrutura são chamados de elementos da estrutura.

7.1.1 –Exemplo

```
#include <stdio.h>
#include <conio.h>
struct endereco {
                    char nome[30];
                    char rua[40];
                    char cidade[20];
                    unsigned long int  cep;
                  };
struct endereco ficha ;
void main( )
{
 gets(ficha.nome);
 ficha.cep = 12345;
 printf("%u",ficha.cep);
 register int i;
 for ( i= 0; ficha.nome[i];i++)   putchar(ficha.nome[i]);
}
```

7.2 – MATRIZES E ESTRUTURAS

Por exemplo, para se criar 100 conjuntos de variáveis do tipo struct ficha, você deve escrever:

```
struct endereco ficha[100];
```

Para acessar um elemento da estrutura 3, você deve escrever:

```
printf( "%u",ficha[2].cep);
```

7.2.1 – Faça uma agenda com o nome de 5 clientes. Cada cliente é cadastrado, usando-se a opção cadastrar, a agenda deve conter também as opções consultar e sair. Cada cadastro deve ter nome, endereço, cidade e telefone.

7.2.2 – Exemplo com o uso de estruturas e módulos

```
#include <stdio.h>
#include <conio.h>
struct endereco {
                    int nro;
                    char nome[30];
                    char rua[40];
                    char cidade[20];
                    unsigned long int cep;
                    };
struct endereco ficha[10];
```

```
void main()
{
        clrscr();
        int t=1;
        printf("entre com o codigo: ");
        scanf("%i",&ficha[t].nro);
        while(ficha[t].nro != 0)
        {
                printf("entre com o nome: ");
                gets(ficha[t].nome);
                printf("entre com a rua: ");
                gets(ficha[t].rua);
                printf("entre com o cep: ");
                scanf("%u",&ficha[t].cep);
                t++;
                printf("entre com o codigo: ");
                scanf("%i",&ficha[t].nro);
        }
        printf(" ----- impressao ----- \n");
        t=1;
        while(ficha[t].nro > 0)
        {
                printf("\n codigo: %i",ficha[t].nro);
                printf("\n Nome: ");
                register int i;
                for (i=0;ficha[t].nome[i];i++)
                        putchar(ficha[t].nome[i]);
                printf("\n Rua: ");
                for (i=0;ficha[t].rua[i];i++)
                        putchar(ficha[t].rua[i]);
                printf("\n Cep: %u",ficha[t].cep);
                t++;
        }
         getch();
}
```

7.3 – PASSANDO ELEMENTOS DA ESTRUTURA PARA FUNÇÕES

É possível transferir os dados armazenados em uma estrutura através de uma função.

7.3.1 – Exemplos

a)

```
#include <stdio.h>
#include <string.h>
struct lista {
                        char x ;
                        int y;
                        float z;
                };
struct lista exemplo;
void imprime (char ch);
void main ( )
```

```
{
        exemplo.x = getchar();
        imprime(exemplo.x); //passa o valor do caracter em x
}
void imprime(char ch)
{
        printf("%c",ch);
        getch( );
}
```

b) Estruturas de armazenamento com manipulação de caracteres

```
#include <stdio.h>
#include <conio.h>
struct endereco {
        char nome[30];
        char rua[40];
        char cidade[20];
        unsigned long int cep;
        };
struct endereco ficha;
void main()
{
    clrscr();
    printf("entre com o nome: ");
    scanf("%s",ficha.nome);
    printf("entre com a rua: ");
    gets("%s",ficha.rua);
    printf("entre com o cep: ");
    scanf("%u",&ficha.cep);
    printf(" ----- impressao ----- \n");
    printf(" Nome: ");
    register int i;
    for (i=0;ficha.nome[i];i++)
        putchar(ficha.nome[i]);
    printf("\n Rua: ");
    for (i=0;ficha.rua[i];i++)
        putchar(ficha.rua[i]);
    printf("\n Cep: %u",ficha.cep);
    getch();
}
```

7.3.2 – PASSANDO ESTRUTURAS INTEIRAS PARA FUNÇOES

Exemplo:

```
void funcao(struct lista par);
void main( )
{
        struct lista exe2;
        exe2.k = 5;
        funcao(exe2);
}
```

```
void funcao(struct lista par)
{
        printf("%i",par.k);
        getch( );
}
```

7.4 – PONTEIROS

Ponteiro é uma variável que contém um endereço de memória de outra variável.

7.4.1 – DECLARAÇÃO DE UMA VARIÁVEL PONTEIRO

*tipo *nome_da_variável ;*

7.4.2 – OPERADORES DE PONTEIROS

& : operador que retorna o endereço de memória de seu operando

* : operador que retorna o valor da variável que ele aponta

7.4.3 – Exemplos

a)
```
#include <stdio.h>
#include <conio.h>
void main( )
{
   int *ender , cont , val ;
   cont = 100;
   ender = &cont;  //pega o endereço de cont
   val = *ender; //pega o valor que esta no endereço ender
   printf ( " %i ",val )   // exibe o  valor  de val
   getch( );
}
```

b)
```
#include <stdio.h>
#include <conio.h>
void main( )
{
    int x,*p1,*p2;
    x = 101;
    p1 = &x;
    p2 = p1;
    printf ( " %p    %d " ,p2,*p2);
    getch( );
}
```

7.4.4 – PONTEIROS E MATRIZES

Na linguagem C há uma estreita relação entre ponteiros e matrizes. Observe o exemplo abaixo:

```
// versão usando matriz
#include <stdio.h>
#include <ctype.h>
```

```
#include <string.h>
void main( )
{
    char str[80];
    int i;
    printf( “digite uma string em letras maiúsculas” );
    gets(str);
    for (i= 0;str[i] ;i++ ) printf( “%c”,tolower(str[i]));
    getch( );
}
```

```
// versão usando ponteiro
#include <stdio.h>
#include <ctype.h>
#include <string.h>
void main( )
{
   char str[80] , *p ;
   printf( “digite uma string em letras maiúsculas” );
   gets(str);
   p = str;
   while ( *p )   printf ( “%c”,tolower(*p++) );
   getch( );
}
```

A versão deste programa usando-se ponteiro é mais rápida que a outra usando-se matriz. O compilador demora mais tempo para indexar uma matriz do que para usar o operador *.

7.4.5 – INDEXANDO UM PONTEIRO

Em C é possível indexar um ponteiro como se ele fosse uma matriz.

Exemplos:

a)
```
#include <stdio.h>
void main( )
{
        int i[5]={1,2,3,4,5};
        int *p , t;
        p = i;
        for ( t=0 ; t<5 ; t++)   printf (“%d”,p[t]);
        getch( );
}
```

b) Ponteiros e strings)
```
#include <stdio.h>
#include <conio.h>
int conta(char *s);
void main( )
{
        char  mat[20] = ”tamanho da frase”;
```

```
        int tam;
        tam = conta(mat);
        printf( "%i",tam);
        getch( );
}

int conta(char *s)
{
        int i = 0;
        while(*s)  // repete até o final  da  string
        {
                i++;
                s++;
        }
        return i ;
}
```

c) Obtendo o endereço de um elemento da matriz

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

7.5 – CHAMADA POR REFERENCIA

A linguagem C usa a chamada por valor para passar argumentos para funções. Esse método copia o valor do argumento para o parâmetro. Assim não se altera o valor das variáveis usadas para chamar a função. Pode-se alterar os valores dessas variáveis fazendo uma chamada por referência usando ponteiros. Agora o endereço da variável é passada para a função e não o seu valor.

7.5.1 – Exemplo

```
#include <stdio.h>
#include <string.h>
void leia( char  *s );
void main( )
{
        char str[80];
        leia (str);
        printf ("%s",str);
        getch( );
```

```
}
void leia( char *s )
{
        char ch;
        for (int i= 0 ; i< 80 ; i++)
        {
                ch = getchar( );
                switch (ch);
                {
                        case '\n' : return;
                        case '\b' : if (t> 0) t-- ;break ;
                        default : s[t] = ch ;
                }
        s[80] = '\0';
}
```

7.6 – Lista de Exercícios

7.6.1 – Escreva um programa cuja execução se segue abaixo:

```
c:\> digite uma frase:
c:\> carlos  <enter>
c:\> digite uma letra dessa frase :
c:\> r  <enter>
c:\> rlos
```

7.6.2 – Monte a função: **void troca (int * x , int * y) ,** cuja execução seja assim:

```
c:\> digite dois numero x e y respectivamente
c:\> 4   9  <enter>
c:\> x = 9  e  y = 4
```

7.6.3 – Escreva um programa que leia e imprima uma número. Obs: não use a função padrão **scanf**, você deve definir uma, por exemplo:

```
void leia ( int * x )
```

7.7 – MATRIZES DE PONTEIROS

Podem existir matrizes de ponteiros, como acontece com qualquer outro tipo de dado.

7.7.1 – Exemplo

```
#include <stdio.h>
#include <conio.h>
char *erro[ ] =  { "não posso abrir arquivo",
                   "erro de leitura ",
                   "erro de gravação " };
void mensagem(int i) ;
void main( )
{
        clrscr( );
        mensagem(2);
```

```
        getch( );
}
void mensagem( int i)
{
        printf( "%s",erro[i] );
}
```

7.8 – RETORNANDO PONTEIROS

7.8.1 – Exemplo:

```
#include <stdio.h>
#include <conio.h>
char *verifica( char c , char *s );
void main( )
{
        char s[80] , *p , ch;
        printf("digite uma frase e uma caracter \n ");
        gets(s);
        ch=getche( );
        p=verifica( ch,s);
        if (p) printf( "%s",p);
          else printf("nenhuma correspondência encontrada")
}

char  *verifica(char c , char *s)
{
        int k = 0;
        while ( c !=  s[k] && s[k]  != '\0' ) k++;
        if ( s[k] ) return ( &s[k] );
        else  return  '\0' ;
}
```

7.8.2 – Exercício

Faça um programa que use a função (definida por você): **char *concat(char *s1 , char * s2).** A função retorna um ponteiro que é a concatenação de s2 com s1. Exemplo de chamada:

```
char  mat1[80]="casa " ;
char  mat2[80]="grande";
char  *p;
p  = concat(mat1,mat2);
printf( "%s",p);
```

7.9 – MANIPULAÇÃO DE ARQUIVOS

7.9.1 – ARQUIVOS BINÁRIOS E ARQUIVOS ASCII EM C

A linguagem C trata os arquivos de duas maneiras diferentes : como arquivos ASCII ou como arquivos binários.

Binários: A linguagem C não interpreta o sentido do arquivo quando da leitura ou escrita de dados;

Ascii: Possui duas diferenças principais – quando encontra um ^Z (ASCII 26) no arquivo que estiver sendo lido, C

interpreta o ^Z como um caracter de fim de arquivo, e presume que o fim do arquivo foi alcançado;
- O caracter de nova linha '\n' e armazenado em disco como uma seqüência ASCII 13 10.

7.9.2 – OPCOES PARA ABERTURA DE ARQUIVOS

r	Abrir arquivo existente para leitura
w	Abrir (criar se necessário) para escrita (gravação)
a	Abrir (criar se necessário) arquivo para acréscimo
R+	Abrir arquivo para leitura e escrita
W+	Criar e abrir arquivo para leitura e escrita
A+	Abrir arquivo para leitura acréscimo
rb	Abrir arquivo binário para leitura
wb	Abrir (criar) arquivo binário para escrita
ab	Abrir (criar) arquivo binário para acréscimo
rt	Abrir arquivo texto para leitura
wt	Criar arquivo texto para escrita
at	Abrir arquivo texto para acréscimo
Rb+	Abrir arquivo binário para leitura e escrita
Wb+	Criar arquivo binário para escrita
Ab+	Abrir arquivo binário para acréscimo

7.9.3 – ABERTURA DE ARQUIVOS

Dados podem ser gravados em arquivos de três maneiras diferentes: como dados formatados, como conjunto de caracteres ou como estruturas.

No primeiro caso, poderemos utilizar a função *fprintf( )*, que é uma derivação da função *printf( )* e que possui a seguinte sintaxe:

```
fprintf(nome_do_ponteiro_arquivo," tipo_de_dados",dados)
```

Para iniciar a abertura do arquivo utilizamos a função *fopen( )* cuja sintaxe é a seguinte :

```
ponteiro_de_arquivo = fopen( "nome_do_arquivo","codigo")
```

```
#include<stdio.h>
#include <conio.h>
void main( )
{
    FILE *fp;
    char texto[ ] = {"menina bonita"};
    fp = fopen("alo.txt","w");
    if (fp == NULL)
    {
        print("O arquivo não pode ser criado");
        exit(1);
    }
    fprintf(fp,"%s",texto);
    fclose(fp);
    getch();
}
```

Neste programa declaramos uma string constante *texto* que recebeu o tipo de formatação "%s" na função fprintf, sendo

criado no diretório atual um arquivo ASCII de nome "alo.txt" com a string "menina bonita";
A função *fclose* fecha o arquivo, que não passa a receber mais nem uma ação de gravação ou leitura. Caso não seja utilizada esta função o próprio sistema trata de fechar o arquivo.

No segundo caso comumente utilizamos a função *putc* no lugar de *fprintf* sendo a sua sintaxe a seguinte:

```
puts(caracter,ponteiro_para_o_arquivo);
```

No terceiro caso é comum substituirmos estas funções pela função *fwrite* que é a função mais utilizada por programadores em C. Sua sintaxe apesar de um pouco mais elaborada não possui muitas dificuldades:

```
fwrite(estrutura,tamanho(estrutura),indice,ponteiro_de_arquivo);
```

Eis um exemplo de sua utilização:

```
#include <stdio.h>
#include <conio.h>
typedef struct
{
        char nome[20];
        float deve;
} entrada;

void main()
{
        FILE *fp;
        entrada meus_dados[15];
        strcpy(meus_dados[0].nome,"Manoel Antonio");
        meus_dados[0].deve = 0.001;
        strcpy( meus_dados[1].nome,"Jose Maria");
        meus_dados[1].deve = 13.89;
        fp = fopen("escritor.dat","w");
        fwrite(meus_dados,sizeof(meus_dados),1,fp);
        fclose(fp);
        getche( );
}
```

Exercícios

7.9.4 – LEITURA DE ARQUIVOS

Da mesma forma para a abertura de arquivos de dados podem ser lidos no modo formatado, no modo de um conjunto de caracteres ou lendo-se uma estrutura inteira. No primeiro caso utilizamos a função *fscanf,* que possui função contraria a função *fprintf,* e cuja a sintaxe é:

```
fscanf(ponteiro de arquivo,"tipo das variaveis",variaveis);
```

No segundo caso utilizamos a função *getc*, e para o terceiro caso a função *fread* de sintaxe:

```
fread(estrutura,tamanho(estrutura),indice,ponteirode_arquivo);
```

```
#include <stdio.h>
#include <conio.h>
typedef struct
{
        char nome[20];
        int deve;
} entrada;
void main()
{
     FILE *fp;
     entrada meus_dados[15];
     fp = fopen("escritor.dat","r");
     fread(&meus_dados,sizeof(meus_dados),1,fp);
     printf("%s %d",meus_dados[0].nome,meus_dados[0].deve);
     printf("%s %d",meus_dados[1].nome,meus_dados[1].deve);
     getch();
     fclose(fp);
}
```

FUNCAO *Fseek( )*

Posiciona o ponteiro de arquivo em determinada posição da estrutura, para um melhor aproveitamento de memória disponível.

```
#include <stdio.h>
#include <conio.h>
typedef struct entrada
{
        char nome[20];
        int deve;
};
void main()
{
     FILE *fp;
     int d;
     entrada meus_dados[15];
     fp = fopen("escritor.dat","r");
     puts("Digite o nro do registro a ser recuperado ");
     scanf("%d",&d);
     fseek(fp,(long)(d*sizeof(meus_dados)),0);
     fread(&meus_dados,sizeof(meus_dados),1,fp);
     printf("%s %d",meus_dados[0].nome,meus_dados[0].deve);
     printf("%s %d",meus_dados[1].nome,meus_dados[1].deve);
     getch();
     fclose(fp);
```

```
}
```

7.9.5 – Comandos

Lista de comando que são utilizados com arquivos:

Nome	Função
fopen()	Abre um arquivo
fclose()	Fecha um arquivo
fputc()	Escreve um caracter em um arquivo
fgetc()	Lê um caracter de um arquivo
fputs()	Escreve uma string em um arquivo
fgets()	Lê uma string de um arquivo
fprintf()	Escreve no arquivo, faz o que o print() faz com o console
fscanf()	Lê do arquivo, faz o que o scanf() faz com o console
fwrite()	Escreve tipos de dados maiores que um byte em arquivo
fread()	Lê tipos de dados maiores que um byte em arquivo
feof()	Devolve verdadeiro se o fim do arquivo for atingido
ferror()	Devolve verdadeiro se ocorreu um erro
remove()	Apaga um arquivo
fseek()	Posiciona o apontador no arquivo em um byte específico

Exercício

1) Utilizando-se da função fprintf, criar um programa que grava em um arquivo em disco (um arquivo de dados) que contenha o nome da pessoa e o seu numero de identidade; O programa deve possuir um menu com as opções "*entrada de dados*" e "*sair do programa*".

2) Faça uma agenda com os dados de n clientes, sendo o valor de n lido no programa. Os dados de cada cliente devem estar numa estrutura denominada CLIENTE, que possui os itens "*nome, numero de identidade, telefone, estado civil (s)olteiro ou (c)asado, cidade de origem* ". Tais dados devem ser gravados em disco.
b) Criar duas funções com os nomes *criar_arquivo* e *ler_arquivo,* sendo cada uma destas opções chamadas de um menu que contém os seguintes itens:

```
1- CONSULTAR CLIENTE
2- CADASTRAR CLIENTE
3- SAIR
```

7.9.6 – Exemplos

Programa 1

```
#include <stdio.h>
#include <conio.h>
void main()
{
        FILE *fp;
        char texto[20] = {"dados de leitura"};
        fp = fopen("alo.txt","w");
        fprintf(fp,"%s",texto);
        fclose(fp);
        getch();
}
```

Programa 2

```
#include <stdio.h>
#include <conio.h>
typedef struct
{
        char nome[20];
        float deve;
} entrada;

void main()
{
        FILE *fp;
        entrada meus_dados[15];
        strcpy(meus_dados[0].nome,"Manoel Antonio");
        meus_dados[0] = 0.01;
        strcpy(meus_dados[1].nome,"Jose Maria");
        meus_dados[1] = 13.89;
        fp = fopen("dados.txt","w");
        if (fp == NULL)
        {
          print(“O arquivo não pode ser criado”);
          exit(1);
        }
        fwrite(meus_dados,sizeof(meus_dados),1,fp);
        fclose(fp);
        getch();
}
```

programa 3

```
#include <stdio.h>
#include <conio.h>
#define n 15

typedef char alfa[n];

typedef struct
{
        int dia;
```

```
        int mes;
    } dt;

    void main()
    {
        FILE *fp;
        dt registro[15];
        registro[0].dia = 1;
        registro[0].mes = 1;
        registro[1].dia = 10;
        registro[1].mes = 10;
        fp = fopen("dados.txt","w");
        fwrite(registro,sizeof(registro),1,fp);
        fclose(fp);
        getch();
    }
```

Programa 4

```
    #include <stdio.h>
    #include <conio.h>
    #define n 15

    typedef char alfa[n];

    typedef struct
    {
        int dia;
        int mes;
    } dt;

    void main()
    {
        clrscr();
        FILE *fp;
        dt registro[15];
        fp = fopen("dados.txt","r");
        fread(&registro,sizeof(registro),1,fp);
        printf("%i  %i ",registro[0].dia,registro[0].mes);
        printf("%i  %i ",registro[1].dia,registro[1].mes);
        fclose(fp);
        getch();
    }
```

8 – ALOCAÇÃO DINÂMICA

A alocação dinâmica permite ao programador criar variáveis em tempo de execução. Isto significa que é possível alocar memória para novas variáveis quando o programa está sendo executado. O padrão C ANSI define quatro funções para o sistema de alocação dinâmica, que estão disponíveis na biblioteca **stdlib.h**.

8.1 – FUNÇÃO MALLOC

A função malloc() serve para alocar (reservar) uma área de memória. A funçao toma o número de bytes que se quer alocar (num), aloca na memória e retorna um ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função malloc() retorna um ponteiro nulo.

Sintaxe:

```
void *malloc (unsigned int num);
```

Exempo:

```
#include <stdlib.h>
main (void)
{
      int *p;
      int tam;
.     .. /* Determina o valor de tam em algum lugar do programa */
      p=(int *)malloc(tam*sizeof(int));
      if (!p)
      {
         printf (" Erro: Memoria Insuficiente ");
         exit;
      }
.     ..
      return 0;
}
```

Neste exemplo, é alocada memória suficiente para se colocar tam números inteiros. O operador **sizeof()** retorna o número de bytes de um inteiro. Ele é útil para saber o tamanho de tipos alocados. O ponteiro **void*** que **malloc()** retorna é convertido para um **int*** e é atribuído a p. na próxima linha, testa se a operação foi bem sucedida. Se não foi, p terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação foi bem sucedida, pode-se usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de p[0] a p[(tam-1)].

8.2 – FUNÇÃO CALLOC

A função calloc() também serve para alocar memória, mas possui funcionalidade diferente: A função aloca uma quantidade de memória igual a **num * size** , isto é, aloca memória suficiente para uma matriz de **num** objetos de tamanho **size**. Retorna um

ponteiro **void *** para o primeiro byte alocado. O ponteiro **void *** pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função calloc() retorna um ponteiro nulo.

Sintaxe:

```
void *calloc (unsigned int num, unsigned int size);
```

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
main (void)
{
      int *p;
      int tam;
      ...
      p=(int *)calloc(tam, sizeof(int));
      if (!p)
      {
         printf ("Erro: Memoria Insuficiente ");
         exit;
      }
.     ..
      return 0;
}
```

No exemplo, é alocada memória suficiente para se colocar **tam** números inteiros. O operador **sizeof()** retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro **void *** que calloc() retorna é convertido para um **int *** e é atribuído a **p**. A linha seguinte testa se a operação foi bem sucedida. Se não foi, **p** terá um valor nulo, o que fará com que **!p** retorne verdadeiro. Se a operação foi bem sucedida, pode-se usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de p[0] a p[(tam-1)].

8.3 – FUNÇÃO REALLOC

A função realloc() serve para realocar (reorganizar) a memória. A função modifica o tamanho da memória previamente alocada apontada por ***ptr** para aquele especificado por **num**. O valor de **num** pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque realloc() pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida. Se **ptr** for nulo, aloca **size** bytes e devolve um ponteiro; se **size** é zero, a memória apontada por **ptr** é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

Sintaxe:

```
void *realloc (void *ptr, unsigned int num);
```

8.4 – FUNÇÃO FREE

Quando É alocada memória dinamicamente é necessário que ela seja liberada quando não for mais necessária. Para isto existe a função free() que é a responsável por este procedimento.
Passar para a função free() o ponteiro que aponta para o início da memória alocada. A função sabe quanto foi alocado de memória, guardou o número de bytes alocados numa "tabela de alocação" interna.
Sintaxe:

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

Exemplo:

```
#include <stdio.h>
#include <alloc.h>
main (void)
{
      int *p;
      int tam;
      ...
      p=(int *)malloc(tam*sizeof(int));       /* alocando a memória */
      if (!p)
      {
         printf ("Erro: Memoria Insuficiente");
         exit;
      }
      ...
      free(p);          /* liberando a memória alocada */
      ...
      return 0;
}
```

8.5 – EXEMPLO DE ALOCAÇÃO DINÂMICA DE UM VETOR

A alocação dinâmica de vetores utiliza os conceitos apresentados sobre ponteiros e as funções de alocação dinâmica aqui discutidos.
Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
/* Procedimento que faz a alocação do vetor */
float *Alocar_vetor_real (int n)
{
      float *v; /* ponteiro para o vetor */
      if (n < 1) {                        /* verifica parâmetros recebidos */
         printf ("Erro: Parametro invalido \n");
         return (NULL);
      }
      /* aloca o vetor */
      v = (float *) calloc (n+1, sizeof(float));
      if (v == NULL) {
         printf ("Erro: Memoria Insuficiente");
         return (NULL);
```

```
        }
        return (v);                     /* retorna o ponteiro para o vetor */
}
/* procedimento que libera o vetor */
float *Liberar_vetor_real (int n, float *v)
{
        if (v == NULL) return (NULL);
        if (n < 1) {                    /* verifica parametros recebidos */
           printf ("Erro: Parametro invalido \n");
           return (NULL);
        }
        free(v);                        /* libera o vetor */
        return (NULL);                  /* retorna o ponteiro */
}
/* Programa principal */
void main (void)
{
        float *p;
        int tam;
        ... /* outros comandos, inclusive a inicialização de tam */
        p = Alocar_vetor_real (tam);
        ... /* outros comandos, utilizando p normalmente */
        p = Liberar_vetor_real (tam, p);
}
```