8 - FUNÇÕES

8.1 - Características básicas

- É um trecho independente de código, com objetivos bem definidos.
- Programas em C, geralmente consistem em várias pequenas funções, ao invés de poucas de tamanho maior.

8.1.1 - Vantagens

- Evita que o programador tenha que escrever o mesmo código repetidas vezes. Ex.: Cálculo da distância entre dois pontos.
- Permite modularidade (dividir programa em partes):
    - Facilita a depuração (debug) e portabilidade.
    - A manutenção dos módulos é muito mais simples que em um programa completo sem funções.
    - É possível construir bibliotecas contendo partes de código pronto e testado.
- Para utilizar uma função pronta, o programador não precisa saber como esta foi feita e nem conhecer detalhes de sua implementação. É preciso saber apenas:
    - sua utilidade
    - parâmetros
    - valores retornados

8.1.2 - Forma geral de uma função

- A forma geral de uma função é:

```
<tipo_da_função>   <nome_da_função> (lista de parâmetros)
{
    corpo da função
}
```

onde

- <tipo_da_função> corresponde ao tipo retornado pela função, caso a função retorne algum valor.

- <nome_da_função> deve ser escolhido seguindo as mesmas regras de nomes de variáveis.

- (lista de parâmetros) corresponde aos valores passados na chamada à função.

- Exemplo:

```
#include <stdio.h>
main( )
{
   linha( );                                  /*chamada à função linha */
   printf (“ Um programa em C \n”);     /*chamada à função printf */
   linha( );                                  /*chamada à função linha */
}

/*  definição da função linha  */
linha( )
{
   int j;
   for (j = 0;  j < 16;  j + +)
      printf (“*”);
  printf (“\n”);
}
```

Saída do programa:

****

Um programa em C

****

- Em C só existem funções, ao contrário de Pascal, onde existem funções e procedimentos.
- Em C, funções podem ou não retornar valores.

8.2 - Chamada e Definição de Funções

- Toda função em C tem um nome
- Um programa tem pelo menos uma função: main( ). Esta determina o inicio do programa.
- main( ) pode chamar outras funções, que por sua vez também podem chamar outras, etc.
- Definição da função:

```
nome( )                 /*  sem    ;    */
{
    .
    .
    .
}
```

- Chamada da função (em qualquer outra função):

```
main( )
{
    .
    .
   nome( );          /*  com  " ; "   */
   .
   .
}
```

- O padrão ANSI exige que uma função seja declarada antes de ser utilizada por outra função, assim como ocorre com as variáveis.

- Declarar uma função ≠ definir uma função

- Uma função pode chamar a si mesma ⇒ <u>função recursiva</u>**.**

- A única função que não pode ser recursiva em um programa é a função main().

- Em C não existem funções aninhadas, ou seja, uma função definida dentro de outra função.

8.3 - Variáveis Locais

- Uma função pode ter variáveis locais:
  - só são conhecidas dentro da função onde foram declaradas
  - só existem durante a execução da função
  - Exemplo:

```
linha( )
{   int j;              /* j variável local da função linha  */
        ......
        .....
}
```

- Uma variável pode ser local a um bloco, ou seja, pode ser declarada dentro do bloco e existir somente durante a execução do mesmo.
  - Exemplo:

```
main( )
{   int i = 5;                      /*   local a main   */
    printf ("Em main( ), o endereço de i
              =  %i e o valor  =  %d \n",  &i, i);
    if (i  = = 5)
    {   int i = 2;                  /*   local ao bloco   */
        printf ("No bloco, o endereço de i é %i
                  = o valor  =  %d \n",  &i, i);
     }
     printf ("i = %d  depois do bloco \n", i);
}
```

  - Existem duas variáveis i diferentes, cada uma ocupa um endereço de memória.

- Se for declarada uma variável local, com o mesmo nome de uma variável externa, a que for declarada no bloco (ou função) tem precedência sobre a externa.
  - Exemplo:

```
int i = 5;                                  /*  global  */

main( )
{   int i = 10;       /*   local   */
    printf (“Em main( )  i = %d\n”, i );      /* imprime 10 */
    função1( );
}
função1( )
{
   printf (“Em função1( ), i = %d\n”, i);    /*  imprime 5  */
}
```

  - função1() desconhece o que foi declarado na função main().

8.4 - Argumentos

- Uma função pode ter argumentos (ou parâmetros): dados recebidos da função que a chamou.
- Argumentos funcionam de maneira semelhante a variáveis locais, são criadas quando a função começa a ser executada e são destruídas quando a função termina.
- Se uma função usa argumentos, ela deve declarar variáveis que aceitem os valores dos argumentos. Essas variáveis são os parâmetros formais da função.
- Os argumentos usados para chamar a função devem ser compatíveis com os tipos de seus parâmetros.

- Exemplo:

```
main( )
{
    abs(0);
    abs(-3);
    abs(10);  /* chamadas à função abs, parâmetros reais */
}

abs(int x)              /* definição da função */
{
    if (x < 0)
        printf (“% d”, -x);
    else
        printf (“% d”, x);
}
```

  - 0, -3 e 10 são os argumentos passados para a função abs.
  - x é parâmetro da função abs, e é uma variável local.

- A maneira de declarar parâmetros de uma função no padrão ANSI C é diferente do padrão K&R.
  - Exemplo:

```
abs(x)        /* padrão  K&R  */

 int x;  {    .....

 }
```

é escrito da seguinte maneira no padrão ANSI:

```
abs (int x);

{  .....

 }
```

- Exemplo:

```
int main( )
{
  int a, b;
  scanf (“%d   % d”,  &a,  &b);
  pot (a, b);
}

pot (int x, int y)       /*  definição de pot */
{
  int i,  potencia = 1;
  for (i = 0;  i < y;  i + +)
    potencia * = x;
  printf (“% d ^ % d = % d \n”, x,  y,  potencia);
}
```

  - *x* é cópia de *a*, *y* é cópia de *b* (*x* e *a; y* e *b* devem ser do mesmo tipo).
  - *a* e *b* são argumentos.
  - *x* e *y* são parâmetros e são variáveis locais de pot().

8.5 - Comando return

Uma função pode ou não retornar um valor. Esse retorno é feito pelo comando **return**.

Sintaxe: return (< expressão >); /* retorna um valor */

Obs: return( ); ⇒ Não retorna valor. Usado no padrão K&R.

return; ⇒ ANSI C;

O return não precisa estar presente, e também pode ser usado em funções que não retornam valor para forçar o final da execução da função.

Ex 9: Outra versão do Ex 8.

```
main( )
{   int a, b, c;
    scanf (“%d   %d”,  &a, &b);
    c = pot (a,b);                ⇒   ↓
    printf (“% d \n “, c);      ⇒   ou :  printf (“% d \n”, pot (a,b));
}

pot (int x, int y)
int x, y;
{   int i, potencia = 1;
    for (i = 0;  i < y;  i + +)
       potencia * = x;
    return (potencia);
}
```

Obs: c e potencia devem ter mesmo tipo.

8.6 - Tipo de uma Função

É o tipo do valor que ela retorna.

Ex: inteiro, real, nada (ANSI C).

8.6.1 -Funções inteiras

Se uma função não tiver seu tipo especificado, C assume que é do tipo inteiro.

No Ex. 9, pot (x, y) é equivalente a : int pot (x, y)

Isto é coerente, pois pot retorna o valor de potencia, que é um inteiro.

8.6.2 - Funções de outro tipo

O tipo retornado deve ser especificado na definição da função. Se a primeira chamada à função ocorrer antes de sua definição, então a função deve ser declarada na função que a utiliza, assim como as variáveis e constantes.

Obs 1: O padrão ANSI exige que sempre seja feita a declaração: **protótipo da função**.

Obs 2: Nos exemplos anteriores (Ex 6 a Ex 9), a declaração deveria ter sido feita, pois a definição está depois da chamada à função.

Ex 10:

```
float area (float raio);          /*   declaração da função   */
                  ↓
                ANSI C
/* ou */
float area (float);           //  declarando somente os tipos */

main( )
 {
      float raio;                          /*   declaração da variável   */
      printf (“Digite o raio da esfera:  \n”);
      scanf (“% f”, &raio);
      printf (“A área é % .2f \n”, area (raio));
 }                                  ↓
                  Chamada à função, retorna um float.

float area (float r)                 /*   definição da função   */
{
       return ( 4 * 3.14159 * r * r);
}
```

Ex 11:

```
main( )
{    char proximo( float );
     char anterior( );        /*   declaração das funções   */
     char car;
     printf (“Digite um caracter:  “);
     scanf (“% c”, &car);
     printf (“Depois de %c vem o %c e antes vem o
       %c  \n”, car,       proximo(car), anterior(car));
}

char proximo (char c)          /*   definição da função   */
{
    return (c + 1);             /*  ou,   + + c   */
}

char anterior (char c)           /*   definição da função   */
{
   return (c - 1);              /*   ou,   - - c    */
}
```

Geralmente as declarações de funções são colocadas no preâmbulo ou em um arquivo .h .

8.6.3 - Funções que não retornam valores

ANSI C “batizou” funções que não retornam valores como sendo do tipo **void**

Ex 12: Outra versão do Ex 1 (desenha linha) em ANSI C.

```
main( )
{
   void linha( );                    /*  declaração  */
   linha( );                         /*  chamada  */
   printf (“...”);
   linha( );                         /*  chamada  */
}

void linha( )                        /*  definição  */
{
   .
   .
   .
}
```

8.7 - Recursão

Função Recursiva: É parcialmente ou totalmente definida em termos de si própria.

Ex: fatorial de um número:

Se $n = 0 \rightarrow 0! = 1$

Se $n > 1 \rightarrow n*(n - 1)!$

Quando uma função recursiva é ativada, um novo conjunto de variáveis locais (incluindo os argumentos) é alocado.

Variáveis: têm o mesmo nome, porém diferentes localizações na memória.

Ex 1: Programa

```
#include <stdio.h>
long fatorial (int);                /*   declaração   */

main( )
{   int n;
    printf (“\n Digite n:  “);
    scanf (“% d”, &n);
    printf (“\n O fatorial de % d  é  % ld \n”, n,  fatorial(n) );
}

long fatorial (int num)             /*   definição   */
{
    if (num > 0)
        return (num * fatorial (num - 1) );
  else
        return (1);
}
```

Cálculo de fatorial de 5:

n = 5

1) num = 5 ↓ (fatorial (5) = 120 ¬

2) num → 5 * fatorial(4) ↓ (fatorial 4 = 24)

3) num → 4 * fatorial(3) ↓ (fatorial 3 = 6)

4) num → 3 * fatorial(2) ↓ (fatorial 2 = 2)

5) num → 2 * fatorial(1) ↓ (fatorial 1 = 1)

6) num → 1 * fatorial(0) ↓ → (fatorial 0 = 1 = num) ↓

Observações:

- Foram feitas 6 chamadas à função fatorial.
- A cada chamada, uma variável foi alocada.
- Durante a execução de fatorial 0, tinha 6 variáveis num alocadas, mais n do main.

Exemplo 2:

Sejam n e m inteiros. Calcular X(n, m) da seguinte maneira: Se n = = m ou m = = 0, X(n, m) = 1, senão X(n, m) = X(n-1, m) + X(n-1, m-1).

```
#include <stdio.h>
int X(int, int);                          /*  declaração  */
main( )
{
    int a, b;
    printf (“\n Digite a e b: \n”);
    scanf (“% d  % d”,  &a, &b);
   printf (“\n X(%d, %d) = %d \n”, a, b, X(a,b) );
}

int X(int n, int m)                        /*  definição  */
{
if ( (n = = m) | | (m = = 0) )
        return (1);
else
        return (X(n-1, m) + X(n-1, m-1));
}
```

Cálculo de X(5, 3) a = 5 b = 3

1o.

X(5,3)

2o. ↓ 9o.

X(4, 3) + X(4, 2)

3o. 4o. ↓ ↓10o. 15o.

X(3, 3) + X(3, 2) X(3,2) + X(3,1)

↓ ↓ ↓ ↓

1 5o. 6o. 11o. 12o. 16o. 19o.

X(2, 2) + X(2,1) X(2,2) + X(2,1) X(2,1) + X(2,0)

↓ ↓ ↓ ↓ ↓ ↓

1 ↓ **1** ↓ ↓ **1**

7o. 8o. 13o. 14o. 17o. 18o.

X(1,1) + X(1,0) X(1,1) + X(1,0) X(1, 1) + X(1, 0)

↓ ↓ ↓ ↓ ↓ ↓

1 **1** **1** **1** **1** **1**

Efetuando as somas, de baixo para cima, temos X(5, 3) = **10**

Observações:

Foram feitas 19 chamadas à função X.

A cada chamada eram alocadas 2 variáveis (novas).

O número máximo de variáveis (m, n) alocadas ao mesmo tempo, foram 10 (5 níveis com 2 variáveis cada, nas chamadas 7o., 8o., 13o., 14o., 17o., 18o.).

Problemas da Recursão:

Alocação de memória: Cada vez que a função é chamada, um novo conjunto de variáveis locais é alocado.

Os algoritmos recursivos em geral, são mais lentos que os não recursivos. Então, à medida do possível, optar por soluções não-recursivas.

Porém, nem sempre isto é possível e muitas vezes, a solução recursiva é mais fácil e mais simples que a não-recursiva, compensando os gastos com alocação de memória.

Solução Recursiva x Solução Iterativa

Vamos analisar a sequência de chamadas da função fatorial (exemplo 1):

fatorial 5
↓
fatorial 4
↓
fatorial 3
↓
.
.
.

∗ Apresenta estrutura linear de chamada. É mais fácil obter solução não-recursiva.

Versão não-recursiva:

```
int i;
long fatorial = 1;                /*  variável  */
if (n > 1)
      for (i = 2; i < = n; i + +)
            fatorial = fatorial * i;        /*  ou fatorial * = i  */
printf (“%l”, fatorial);
```

Sequência de chamadas do ex 2: X(5, 3)

Apresenta estrutura em árvore.

É mais fácil obter a solução recursiva.

É mais difícil obter a solução não-recursiva.

Exemplo 3: Sequência de Fibonacci

0	1	2	3	4	5	6	7
0	1	1	2	3	5	8	13

0 + 1 = 1 1 + 1 = 2 1 + 2 = 3 2 + 3 = 5 3 + 5 = 8 5 + 8 = 13

1o. número = 0

2o. número = 1

3o. número = 1

outros = soma dos dois antecessores

posição na sequência

FIB (0) = 0

FIB (1) = 1

FIB (n) = FIB (n-1) + FIB (n-2)

```
int FIB (int);
main( )
{   int n;                          /*  posição  */
    printf (“Digite posição > = 0:  “);
   scanf (“% d”, &n);
   printf (“\n FIB (% d) = % d \n”, n, FIB(n) );
}

int FIB (int n)
{   if (n < = 1)
        return (n);
    return (FIB (n-1) + FIB (n-2) );
}
```

Executar para n = 4:

1o. FIB (4) = 3

2o. FIB (3) = 2 7o. FIB (2) = 1

3o. FIB (2) = 1 6o. FIB (1) = 1 8o. FIB (1) = 1 9o. FIB(0) = 1

4o. FIB (1) = 1 5o. FIB (0) = 0

Estrutura em árvore: é mais difícil obter solução não-recursiva.

O exemplo anterior apresenta estrutura em árvore. Porém, os parâmetros se repetem com frequência, muitos cálculos são repetidos. Desta maneira é possível e aconselhável obter solução não-recursiva.

Versão Não-Recursiva Com Vetor:

```
main( )
{   int n, i;
    int FIB [NMAX];
    FIB[0] = 0;
    FIB[1] = 1;
     .
     .
     .
    if (n > 1)
       for (i = 2; i < = n; i + +)
              FIB[i] = FIB[i - 1] + FIB[i - 2];
    printf (“ . . .”, FIB[n]);
}
```

Versão Não-Recursiva Sem Vetor:

num 0 0

num 1 +1 → 1

fib 1 → +1 → 1

2 → +2

3

```
main( )
{
   int n, i, fib 0 = 0,  fib 1 = 1,  fib;
  …..
   if (n > 1)
       for (i = 2; i < = n; i + +)
       {
            fib = fib 0 + fib 1;
            fib 0 = fib 1;
            fib 1 = fib;
       }
    else
         if (n = 0)
              fib = fib 0;
       else
            fib = fib 1;
    printf (“...”, fib);
}
```

Exercício: Ler <u>x</u> e <u>y</u>:

```
float  F(float x, float y)
{
       if   (x > = y)
          return ( (x + y) / 2);
       else
          return ( F (F (x + 2, y - 1),   F ( x + 1, y - 2) ) );
                                                  ↓
                                          chama 1o.(direita)
main( )
{
   float a, b;
   printf (“% f”,   F( a, b));
}
```

Exercício:

Avaliar F (1, 10), construindo a sequência de chamadas.

8.8 - Passagem de parâmetros por valor e por referência

Exemplo 1: Programa para trocar os valores de a e b.

Versão 1:

```
void troca (int x, int y)
{
   int aux;
   x = y;
   y = aux;
}        /*  Troca x com y, mas não troca a com b.    */

main( )
{   int a, b;
    scanf (“% d   % d”, &a, &b);
    troca (a, b);
    printf (“\n Elementos trocados   % d   % d\n”, a, b);
}
```

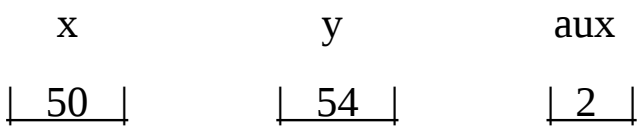
Versão 2:

```
main( )
{   int a, b;
    scanf (“% d   % d”, &a, &b);
    troca (&a, &b);
    printf (“\n   Elementos trocados % d   % d \n”, a, b);
}

void troca (int *x, int *y)
{   int aux;
    aux = *x;
    *x = *y;
    *y = aux;
}
```

Em main:

Em troca:



8.9 - Arrays como Argumentos de Funções

Exemplo: A função addconst adiciona uma constante a cada elemento do vetor:

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

void  addconst (int v[TAM], int c)  /* tipo do array e dimensão */
{     int k;
      for (k = 0;  k < TAM;  k++)
            v[k]  +=  c;
}

main( )
{   int vetor [TAM] = {2, 5, 7, 9, 11};
    int cte = 10,j;

   /*   o nome do vetor: já é endereço, vetor vai ser modificado*/
   /* const: passa por valor porque não será alterado   */

    addconst (vetor,  cte);    /* somente o nome do array */

    for (j = 0;   j < TAM  ;  j++)
       printf (“%d”,   vetor[j]);
}
```

Passagem por valor existe somente para variáveis simples. Arrays são sempre passados por referência, pois como já vimos, o nome do vetor é um apontador para o mesmo!

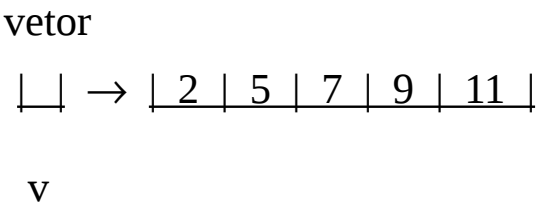
Existem diferentes maneiras de declarar o parâmetro v.

void addconst (int v[TAM], int c)

void addconst (int v[ ], int c)

void addconst (int* v, int c)

(v é um apontador para inteiros, os elementos do array estão alocados sequencialmente na memória, lembra!)



E diferentes protótipos para declarar a função addconst.

void addconst (int [TAM], int);

void addconst (int [], int);

void (int*, int);

OBS: A inclusão do tamanho de um array unidimensional na definição da função é somente por razões de legibilidade mas para arrays multi-dimensionais, todas as dimensões, exceto a primeira, devem ser especificadas.

```
#include <stdio.h>
#define   TAM   10
void inicializa_matriz(int matriz[TAM][TAM], int nlin, int ncol, int valor)
{    int i,j;
     for(i=0; i<nlin; i++)
          for(j=0; j<ncol; j++)
               matriz[i][j] = valor;
}

void imprime_matriz(int matriz[TAM][TAM], int nlin,
int ncol)
{    int i,j;
     for(i=0; i<nlin; i++)
          for(j=0; j<ncol; j++)
           printf("%d\n", matriz[i][j]);
}

main ()
{  int  M[TAM][TAM], n,m,v;
   printf (“Entre com o número de linhas: ”);
   scanf (“%d”, &n);
   printf (“\n Entre com o número de colunas: ”);
   scanf (“%d”, &m);
   printf (\n “Entre com o valor default: ”);
   scanf (“%d”, &v);
   inicializa_matriz(M, n, m, v);
   imprime_matriz (M,n,m);
}
```

No exemplo acima poderia ter sido usado

```
void inicializa_matriz(int matriz[][TAM], int nlin, int ncol, int valor)
```

Protótipos para declarar inicializa_matriz:

```
void inicializa_matriz (int [TAM] [TAM], int, int,  int);

void inicializa_matriz (int [] [TAM], int, int,  int);
```

Exemplo 2: Programa que lê uma frase do teclado, conta o número de espaços brancos e o número de letras minúsculas:

```
#include <stdio.h>
#include <string.h>
#define   TAM   80

void func (char frase[], int *p_br, int *p_mi)
{   int k = 0;
   *p_br = *p_mi = 0;
   while (k < strlen(frase))
   {
       switch (frase [k])
       {   case ‘ ’:                                  /*   branco   */
           case ‘\t’:  + + *p_br ;  break;           /*   tab   */
           default:
                   if ((frase [k] > = ‘a’) && (frase[k] < = ‘z’)
                      (*p_mi)+ +;                /*   ou     + + *p_mi   */
                     break;
       }
    k + +;
    }
}
main( )
{    char  frase [TAM];
     int brancos, minúsculas;
     printf (“Digite uma frase\n”);
     gets (frase);
     func (frase,&brancos, &minúsculas);
     printf (“O número de brancos é  % d  e o número de
            minúsculas é   % d\n”,  brancos, minúsculas);
}
```