



PROGRAMMING

# WORKSHOP

// INTRODUÇÃO A PROGRAMAÇÃO C  
/\* 9 E 10 DE JANEIRO/2023 \*/

## Módulo 2.

- Operadores relacionais
- Ciclos
- Ponteiros
- Funções

# Operadores unários de atribuição: ++ --

Em ciclos é frequente incrementar ou decrementar uma dada variável

Operador	Equivalente	Exemplo
++	$i = i + 1$	$i++;$
--	$i = i - 1$	$i--;$

**Pós**-incremento:  $i++$

**Pós**-decremento:  $i--$

**Pré**-incremento:  $++i$

**Pré**-decremento:  $--i$

Exemplo	Corresponde
$y = x++;$	1. $y = x;$ 2. $x = x + 1;$
$y = x--;$	1. $y = x;$ 2. $x = x - 1;$
$y = ++x;$	1. $x = x + 1;$ 2. $y = x;$
$y = --x;$	1. $x = x - 1;$ 2. $y = x;$

## Exemplo – operadores ++,--

```
//operadores unarios
#include <stdio.h>
int main(){

    int k=0, i=0, j=5;

    printf(" i=%d \n",i);
    i++;
    printf(" i++ = %d \n",i);

    printf(" j=%d \n",j);
    j--;
    printf(" j-- = %d \n",j--);

    i=7;
    k = i++;
    printf(" i=%d e k=%d \n",i,k); //k= i++;
    i = 7;
    k = ++i;
    printf(" i=%d e k=%d \n",i,k); //k= ++i;

    return (1);
}
```

```
i=0
i++ = 1
j=5
j-- = 4
i=8 e k=7
i=8 e k=8
```



```
// ciclos  
// ** while  
// ** do-while  
// ** for
```

## Ciclos - introdução

Até este ponto, em num sentido *lato*, muito dos programas em C anteriores não eram mais do que uma sequência de instruções a serem executadas “linha-após-linha”.

As instruções *if-else* e *switch* permitiram, em função de uma condição ser *true* ou *false*, “saltar” uma ou mais linhas i.e., termos controlo de fluxo do programa...

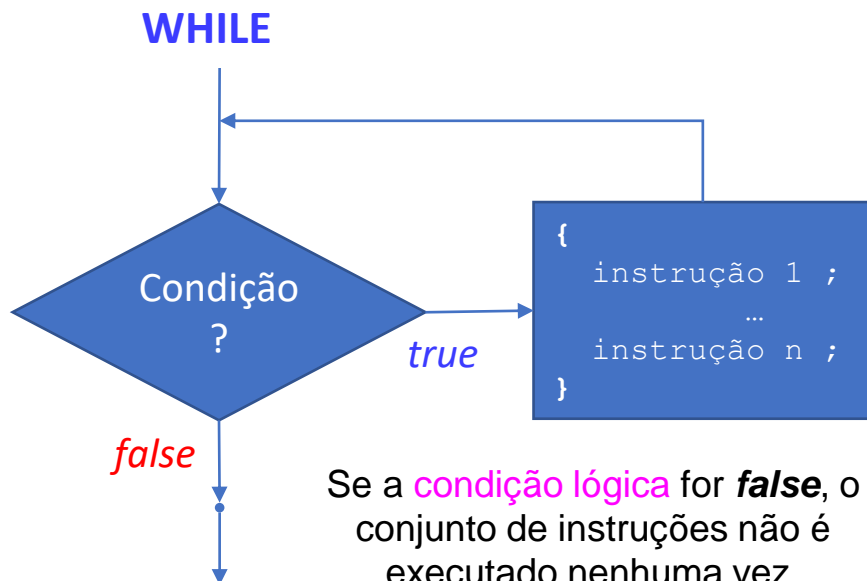
Vamos agora apresentar instruções de controlo de fluxo que são fundamentais em programação, e na resolução de problemas, nomeadamente instruções que permitem a execução de **ciclos** – o que permite repetir uma ou um conjunto de instruções.

As instruções de controlo de fluxo relacionadas com ciclos são:

- *while*
- *do-while*
- *for*

# Ciclo com a instrução **WHILE**

```
while (condição lógica)
{
    instrução 1 ;
    instrução 2 ;
    ...
    instrução n ;
}
```



```
/* exemplo: mostrar no ecrã os numeros  
1 2 3 ... 7 */
```

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

```
int main() {
```

```
    int x, n;
```

```
    n = 7;
```

```
    x=1;
```

```
    while ( x <= n ) {
```

```
        printf("x= %d \n",x);
```

```
        x = x + 1; //temos de incrementar x
```

```
    }
```

```
    return (1);
```

```
}
```

```
x= 1  
x= 2  
x= 3  
x= 4  
x= 5  
x= 6  
x= 7
```

# Exemplo de utilização do *while* – com *array*

```
/* exemplo: media com array */
#include <stdio.h>

int main(){

    unsigned int i, n=5;
    int x[]={-1, 0, 1, 2, 7};
    double mx=0;

    i = 0;
    while (i < n){ //condicao: garantir i eh menor 5 ie, nao pode passar de 4
        mx = mx + x[i];
        printf("mx= %1.2lf  \n", mx);
        i = i + 1; //precisamos incrementar i
    }

    mx=mx/n;

    printf("Apos ciclo, media= %1.2lf  \n", mx);

    return (1);
}
```

```
mx= -1.00
mx= -1.00
mx= 0.00
mx= 2.00
mx= 9.00
Apos ciclo, media= 1.80
```

# Instrução *do-while*

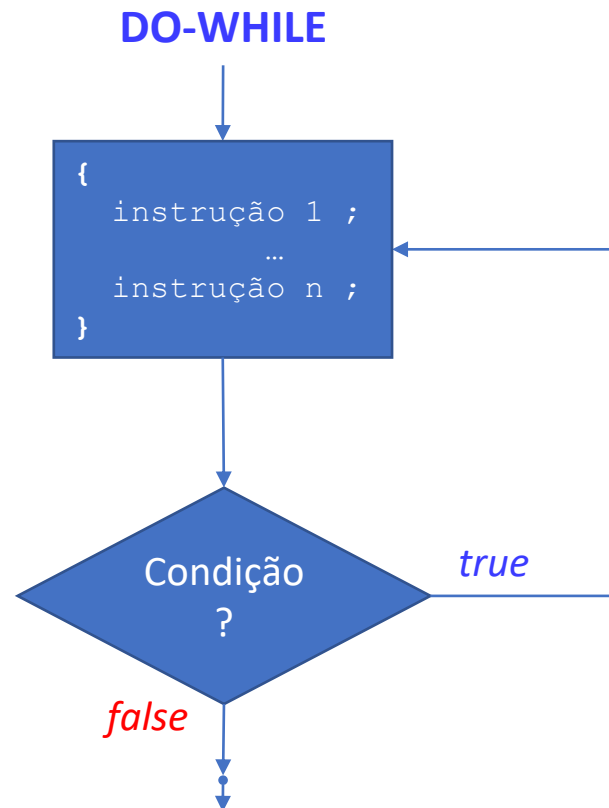
A instrução *do-while* garante que o ciclo é executado pelo menos uma vez.

Com *while*, teremos de garantir que a condição é true para que o ciclo seja executado pelo menos uma vez.

A sintaxe da instrução *do-while*:

```
do
{
    instrução1;
    instrução2;

    instruçãon;
} while (condição);
```





## Exemplo com DO-WHILE

```
#include <stdio.h>  //Exemplo com do-while
int main() {
    int x;

    do {
        printf("Digite um numero inteiro positivo: \n");
        scanf("%d", &x);
    } while (x<=0);

    printf(" Numero valido = %d \n", x);

    return (1);
}
```

Garantir que o utilizador  
digita um número positivo.

```
... //Exemplo com WHILE

printf(" Digite um numero inteiro positivo: \n");
scanf("%d", &x);

while (x<=0){
    printf(" Digite um numero inteiro positivo: \n");
    scanf("%d", &x);
}

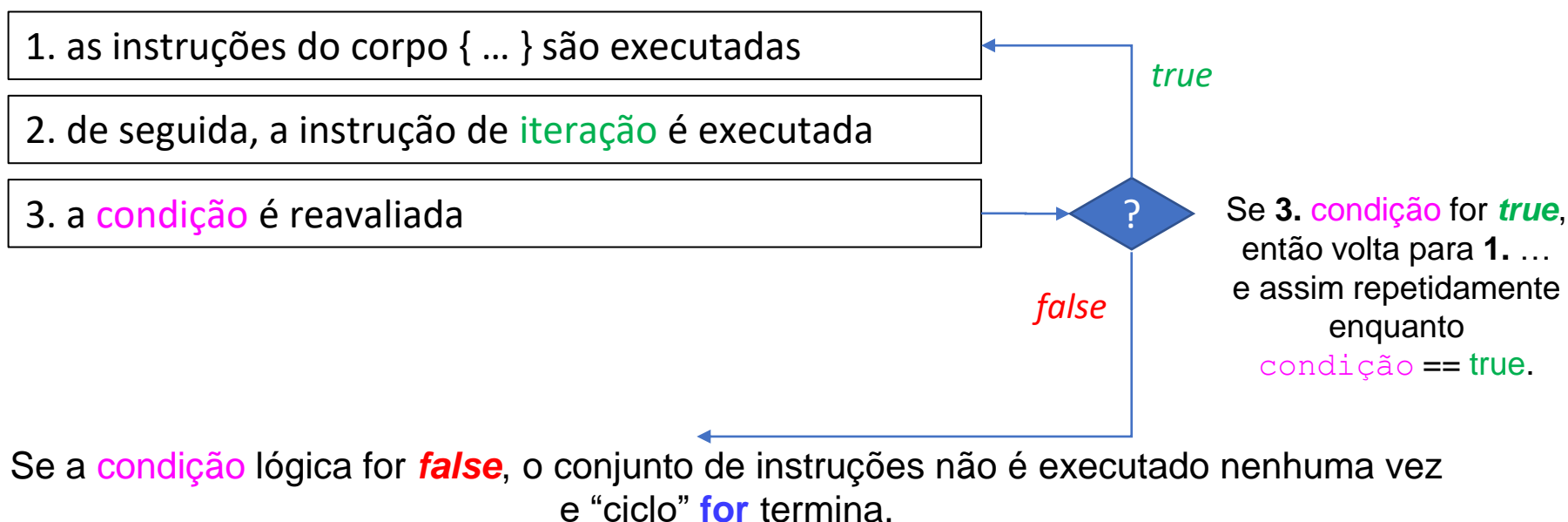
printf(" Numero valido = %d \n", x);
```

# Ciclo com a instrução **FOR**

A **instrução inicial**  
é executada uma vez!

A **condição** é avaliada após a  
**instrução inicial**. Se **condição**  
for verdadeira, então:

```
for (inst.inicial; condição; iteração) {  
    instrução 1 ;  
    instrução 2 ;  
    ...  
    instrução n ;  
}
```



# FOR é particularmente útil em **vectores/arrays**

```
#include <stdio.h> // guardar o maximo de um array num array
```

```
int main() {
```

```
    int i, max, n=5;
```

```
    int c[5], x[5]={2,1,7,10,5};
```

```
    max = x[0]; // primeiro elemento de x
```

```
    c[0] = max;
```

```
    for (i=1; i<n; i++){
```

```
        if( x[i]>max )
```

```
            max =x[i];
```

```
        c[i] = max;
```

```
    }
```

```
    for (i=0; i<n; i++)
```

```
        printf("Max no instante %d -> [%d]\n",i+1, c[i]);
```

```
    return (1);
```

```
}
```

```
Max no instante 1 -> [2]
Max no instante 2 -> [2]
Max no instante 3 -> [7]
Max no instante 4 -> [10]
Max no instante 5 -> [10]
```

# Instrução *break* e *continue* num ciclo

**Break:** permite “terminar/interromper” a execução de um ciclo.

**Continue:** permite controlo/”salto” do fluxo na execução de um ciclo

```
#include <stdio.h> // calcular soma de numeros negativos
int main(){

    int x[5]={-2,-3,10,-4}, soma=0;
    int i=0;

    for(i=0;i<5;i++){

        if (x[i]>=0) continue;
        //if (x[i]>=0) break;

        printf(" %d +",x[i]);

        soma += x[i];
    }

    printf("\n Soma de negativos = %d \n",soma);
    return (1);
}
```

-2 + -3 + -4 +  
Soma de negativos = -9

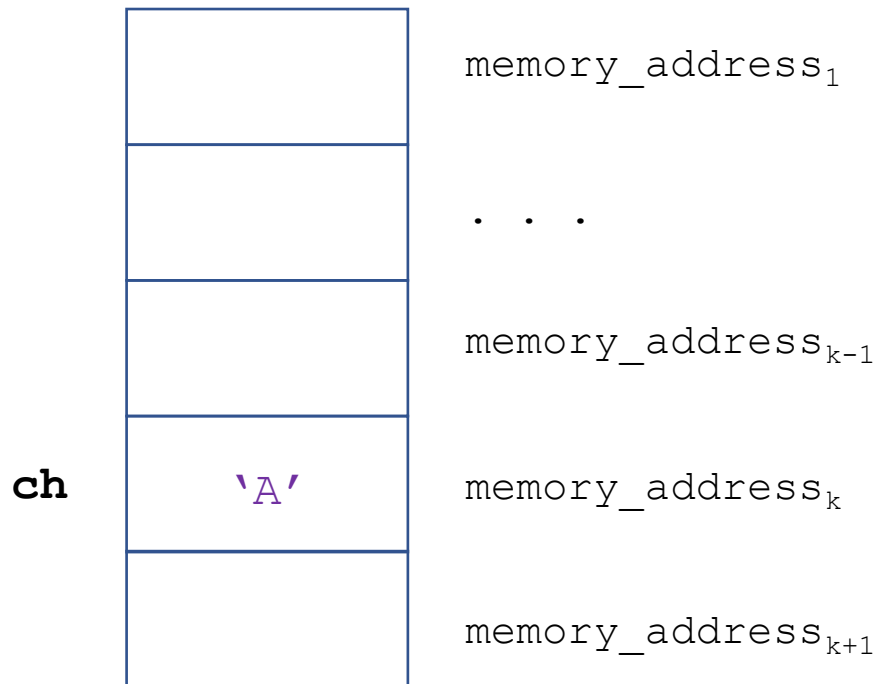
-2 + -3 +  
Soma de negativos = -5



## // Ponteiros

# Memória - visão simplificada

- A **memória RAM** (*Random Access Memory*) onde os nossos programas são carregados, pode ser vista (de forma simplificada) como um **bloco contínuo de bytes**.
- Cada um desses **bytes** ocupa uma posição bem determinada nesse bloco, posição essa que é identificada por um número (o seu endereço).



Quando fazemos: `char ch = 'A';`  
é reservado 1 byte (porque é tipo *char*)  
para armazenar 'A', e esse byte tem um  
endereço (`memory_addressk`).

# Ponteiros

Em linguagem C, um ponteiro (ou apontador ou *pointer*) é uma variável que pode conter o **endereço** de uma outra variável ou entidade (eg, *string*, *array*) para a qual aponta. Endereço aqui refere-se a uma **posição na memória**.

Declaração/sintaxe de um ponteiro

```
tipo * nome_variavel;
```



- Um ponteiro é declarado colocando o operador de declaração-prefixo “\*” antes do nome da variável.

Exemplos:

```
int *p;
```

```
double * p;
```

```
char* p;
```

- A variável `p` (ie, um ponteiro) é capaz de armazenar um endereço de memória;
- Consequentemente, com esse endereço, um ponteiro consegue obter o valor na posição de memória para onde está a apontar.
- Por exemplo: se quisermos apontar `p` para uma variável `x`, fazemos `p = &x;`
- O **tipo** é importante porque, dado um endereço de memória, a interpretação dos *bits/bytes* que aí se encontram depende do **tipo** da informação armazenada.

# Ponteiros - endereço de uma variável

O operador **&** (endereço de) permite obter o endereço de uma variável.

```
#include <stdio.h>    //Exemplo com ponteiro
int main()
{
    int x;
    int *p;

    p = &x;
    printf("Digite um numero inteiro positivo: \n");
    scanf("%d",&x); //digitar 7

    printf(" Posicao memoria de x = %p \n", &x);
    printf(" Valor de p = %p \n", p); //valor do ponteiro, nao precisa de &
    printf(" Valor na posicao memoria apontada = %d \n", *p); //seria o valor de x

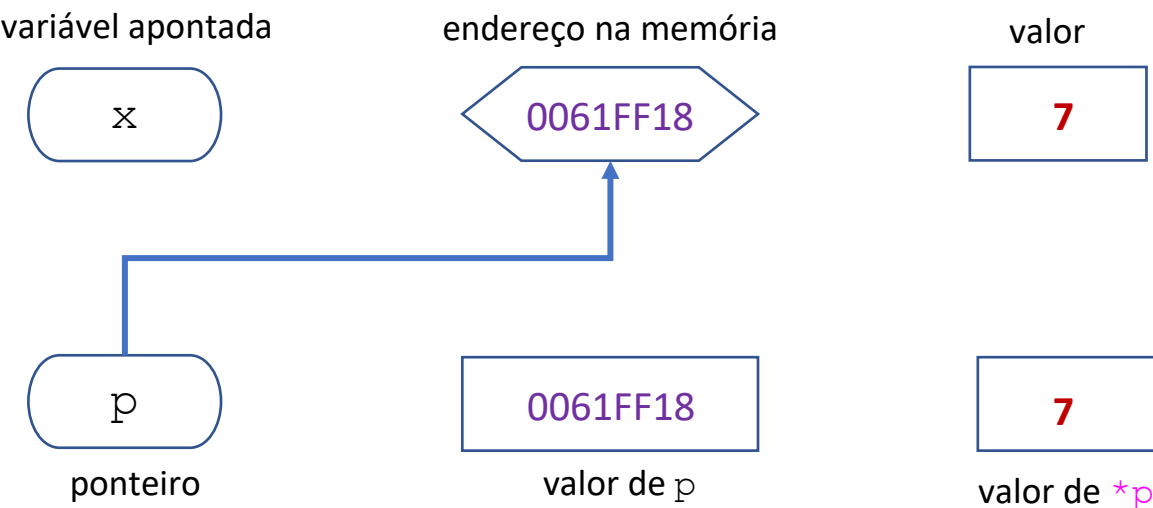
    return (1);
}
```

Para obter o valor da variável  
apontada pelo ponteiro (neste caso x)  
fazemos **\*p**

Posicao memoria de x = 0061FF18  
Valor de p = 0061FF18  
Valor na posicao memoria apontada = 7



# Ponteiros - conceito



O ponteiro `p`, ele próprio uma variável, também tem/ocupa um dado endereço na memória.

## Declaração (posição do prefixo `*`)

- Podemos declarar um ponteiro, pex. do tipo ***int***, numa das 3 formas a seguir:

```
int *p; //forma recomendada
int * p;
int* p;
```

# Ponteiros - inicialização

É boa prática inicializar sempre os ponteiros ou seja, garantir que o ponteiro está a apontar para uma variável. Em alternativa, podemos inicializar com *NULL*

A atribuição da constante *NULL* a um ponteiro sinaliza que este ainda não aponta para um endereço de memória específico.

```
int *p1 = NULL;
float *p2 = NULL;
double *p3 = NULL;
```

Podemos inicializar um ponteiro (por exemplo, apontar para uma variável de interesse) no momento da declaração:

```
int x;
int *p1 = &x; //declaracao e inicializacao
```

Ou, em alternativa, apontar após a declaração:

```
int x;
int *p1; //apenas declara ie, inda não está a apontar para nada
p1 = &x; //agora está a apontar para x
```

# Ponteiros - exemplo

```
#include <stdio.h> //Exemplo com ponteiro

int main(){
    int x = 666;
    int *p = NULL;

    printf(" Endereco de x = %p \n", &x);
    printf(" Valor de p = %p \n", p); //aponta para NULL-
nada
//printf(" Valor de *p = %p \n", &p); //?

    p = &x; //aponta para x
    printf(" Valor de p = %p \n", p);
    printf(" Valor de *p = %d \n", *p); //valor para onde
esta apontar

    return (1);
}
```

Endereco de x = **0061FF1C**

Valor de p = 00000000

Valor de p = **0061FF1C**

Valor de \*p = 666

← endereço de x

# Ponteiros – relação com Arrays

Considere uma tabela/array declarada como `int tab[10]={};`

O valor de `tab` (ou seja, do nome da tabela) corresponde ao endereço do seu 1º elemento, que é equivalente a `&tab[0]`.

```
int tab[10]={};  
printf("Valor de tab = %p \n", tab);  
printf("Endereco de tab[0] = %p \n", &tab[0]);
```

Valor de `tab` = **000000000061FDF0**

Endereco de `tab[0]` = **000000000061FDF0**

Quando compilamos um programa, o compilador “sabe” quantos *bytes* foram reservados para a tabela **tab** que, sendo do tipo inteiro e com 10 elementos, corresponde a 40 bytes.

Em termos de memória, podemos assumir que uma tabela ocupa um **bloco de memória** com os seus **elementos ordenados de forma consecutiva**.

# Tabela - endereço dos elementos

```
char tab[3]={ 'x' , 'y' , 'z' }; //char ocupa 1byte
```

		. . .
tab[0]	'x'	memory_address = 000000000061FE1D
tab[1]	'y'	memory_address = 000000000061FE1E
tab[2]	'z'	memory_address = 000000000061FE1F
		. . .

# Ponteiros – aritmética e operações

- Um ponteiro pode ser incrementado como qualquer variável.
- **Nota:** adicionar 1 a um ponteiro (eg,  $p=p+1$ ) significa que este passará a apontar para o elemento seguinte na tabela.

## Incremento de um ponteiro

Um apontador para o tipo `int` avança sempre `sizeof(int)` bytes (ie, 4 bytes) por cada unidade de incremento. Por outro lado, se for do tipo `char` então `p++` avança 1byte.

## Decremento de um ponteiro

Um apontador para o tipo `int` recua sempre `sizeof(int)` bytes por cada unidade de decremento. Se for do tipo `double`, `p--` recuará 8 bytes na memória.

**Nota:** `p[2]` equivale a `*(p+2)`. O acesso a elementos através de parêntesis retos `[]`, como nas tabelas, pode ser também ser feito através de ponteiros.



## // Funções

# Funções e “procedimentos”

- Distinção entre funções e “procedimentos”
- Passagem (por valor) de parâmetros
- Retornar valor usando uma função

Função / procedimento: tem um **nome**, **tipo**, e pode ou não ter **argumentos** ().

“Procedimento”: é uma função que não retorna nenhum valor ie, é uma função do tipo *void*.

*void* é o tipo que significa “*nada*” ou seja ausência de valor retornado pela função.

Por outro lado, funções que retornam valor podem ser do tipo *char*, *int*, *float*, *double*...

---

Em linguagem C, um programa deve possuir sempre a função *main*.

Usualmente, convém a função *main* retornar um valor que deve ser do tipo *int*.

Contudo, podemos eventualmente criar a função *main* do tipo *void* (logo, não teremos *return*).

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

    printf(" funcao MAIN do tipo int \n");

    return (1); //retorna um inteiro

}
```

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

    printf(" funcao MAIN do tipo void \n");

    //nao retorna valor algum
}
```

warning: return type of 'main' is not 'int' ...



# Variáveis locais

As variáveis declaradas dentro do corpo de uma função só são visíveis (ou seja conhecidas) dentro da função.

**Variáveis declaradas dentro do corpo de uma função** são denominadas de variáveis locais à função.

- **Variáveis declaradas dentro do corpo de uma função** são criadas quando a função é invocada e são destruídas quando a função termina.
- **Duas funções distintas podem ter variáveis locais com o mesmo nome.**

Estas são variáveis distintas sem qualquer relação entre si, que o compilador distingue pelo local onde são utilizadas.

## Procedimento (tipo *void* e sem argumentos de entrada) - exemplo

```
#include <stdio.h>

// função que imprime uma linha-tracejada; não retorna valor
void print_line() {
    unsigned int k;
    for (k=1; k<25; k++)
        printf("-");

    printf("\n");
}

int main() {
    /* a variável k na função Main,
       é diferente do k na função print_line()
       veremos isso mais adiante.
    */
    int k;
    print_line(); //chama a função

    printf("Numeros entre 1 e 5");
    for (k=1; k<=5 ; k++)
        printf("%d\n", k);

    print_line();

    return (1);
}
```

-----

Numeros entre 1 e 5

1  
2  
3  
4  
5

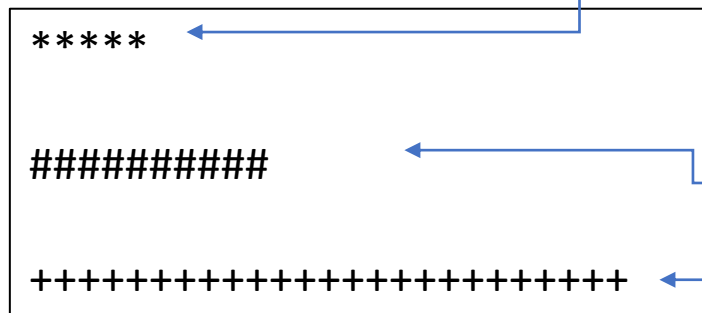
-----

# Caraterísticas de uma função

- Cada função tem um nome único, que serve para a sua invocação. O nome de uma função tem as mesmas regras do nome de uma variável.
- Uma função pode ser invocada a partir de outras funções.
- Uma função deve fazer uma tarefa bem definida (o seu nome deve ser informativo).
- Uma função deve comportar-se como uma *black-box* (não interessa à entidade que a chama saber como foi implementada): deve executar a tarefa para a qual foi implementada, sem “efeitos colaterais”.
- Uma função pode receber argumentos que determinam o seu comportamento.
- Uma função pode retornar, para a função que a invocou, um valor de saída.

# Funções (sem argumentos/parâmetros) – outro exemplo

Funções que apenas imprimem no ecrã diferentes quantidades de símbolos – também diferentes.



```
*****  
  
#####  
  
+++++
```

Observando o código à direita, verifica-se que as três funções diferem apenas no **número de iterações** e no **caractere/símbolo** que será impresso.

```
#include <stdio.h>  
  
void print_symbol_5x() {  
    for (int i=1; i<=5; i++)  
        putchar('*');  
    putchar ('\n'); }  
  
void print_symbol_10x() {  
    for (int i=1; i<=10; i++)  
        putchar('#');  
    putchar ('\n'); }  
  
void print_symbol_25x() {  
    for (int i=1; i<=25; i++)  
        putchar('+');  
    putchar ('\n'); }  
  
//-----  
  
int main() {  
  
    print_symbol_5x();  
    print_symbol_10x();  
    print_symbol_25x();  
  
    return (1);  
}
```

# Funções com parâmetros (argumentos de entrada)

```
#include <stdio.h>

void print_symbols(int n, char c)
{
    for (int i=1; i<=n; i++)
        putchar(c);

    putchar ('\n');
}

int main() {

    print_symbols(5, '*');
    print_symbols(10, '#');
    print_symbols(25, '+');

    return (1);
}
```

- Qual o nome da função?

print\_symbols

- Qual o tipo da função?

void

- Quantos parâmetros (formais) tem?

2

- Qual o tipo do parâmetro?

int e char

- Qual o nome das variáveis que vão armazenar os parâmetros?

n , c

- Qual o cabeçalho da função?

void print\_symbols(int , char );

- Qual o corpo da função?

```
for (int i=1; i<=n; i++)
    putchar(c);

    putchar ('\n');
```

# Funções que retornam um valor e **protótipos**

Uma função, uma vez terminada a sua tarefa poderá devolver um único resultado.

- A devolução é feita pela instrução **return**. O valor a devolver está escrito logo a seguir à instrução **return**.
- **O tipo do valor retornado é sempre do tipo da função.**
- Uma função pode ser invocada em qualquer expressão ou instrução válida para o tipo devolvido pela função.
- Uma função pode ser invocada dentro de outra função (diferente do **main**).
- Uma função pode conter várias instruções **return**, mas no máximo será executada uma delas.

**Valor retornado:** A seguir à instrução **return** pode estar qualquer expressão válida (incluindo chamada a uma função).

Assumindo que o programa se encontra em num único ficheiro:

**As funções podem ser colocadas em qualquer local do ficheiro, desde que seja antes de serem invocadas.**

Para não precisarmos de estar atentos à ordem precisa pela qual uma função chama outra, podemos em alternativa declarar as funções no início do ficheiro. As funções podem em seguida ser definidas por qualquer ordem no ficheiro.

A declaração de uma função consiste na escrita do seu protótipo ou assinatura o qual não é mais do que o cabeçalho seguido de um ponto e vírgula.

## **Protótipos de uma função**

O protótipo de uma função pode ser igual ao cabeçalho seguido de um ponto e vírgula; pode, eventualmente, indicar apenas os tipos dos parâmetros da função (sem a designação das variáveis), separados por vírgulas.

# Funções – exemplo utilizando Protótipo

```
#include <stdio.h>
/* Devolve a soma de dois inteiros */
int soma(int, int); /* protótipo sem parâmetros (A EVITAR: pouco legível!) */
/* Devolve o dobro de qualquer inteiro */
int dobro(int k); /* protótipo com parâmetro */

int main() {
    int n,i,total;
    printf("Introduza dois Números: ");
    scanf("%d%d",&n,&i);

    total = soma(n,i); /* Atrib. do result de função a uma var */

    printf("%d+%d=%d\n",n,i,total);
    printf("2*d=%d e 2*d=%d\n",n,dobro(n),i,dobro(i));
    if(dobro(i) > n)
        printf("O dobro de %d é maior que %d\n", i , n);
    else
        printf("O dobro de %d é menor ou igual que %d\n", i , n);
    return(0);
}

int soma(int a, int b){/* Devolve a soma de dois inteiros */
    return a+b;
}

int dobro(int k){/* Devolve o dobro de qualquer inteiro */
    return 2*k;
}
```

# Funções em C

O número e tipo de argumentos passados a uma função deve corresponder ao número e tipo dos parâmetros com que esta é definida.

Uma função, ao usar a instrução *return*, pode devolver um valor do tipo da função, e no máximo um valor.

**E se desejássemos que uma função retornasse dois ou mais valores?**

Exemplo: Dada uma tabela de números inteiros, e o seu número de elementos, pretende-se que função retorne o menor elemento, o maior elemento e ainda a média dos elementos da tabela ie, queremos que a função "retorne" 3 valores.

Tal só é possível se a função tiver acesso (via endereço/ponteiro) às variáveis onde devem ficar os resultados.



# Funções – Exemplo: limitações da passagem de parâmetros por valor

```
#include <stdio.h>
void Fun_swap(int a, int b); //prototipo

int main(){
    int a=666, b=111;

    printf("a = %d \t b = %d \n", a, b);
    Fun_swap(a, b);
    printf("No MAIN e apos executar Func_swap: ");
    printf("a = %d \t b = %d \n", a, b);

    return(0);
}
//=====
```

```
void Fun_swap(int a, int b){
    int aux = a; //salva valor de a em aux
    a = b;
    b = aux;
    printf("Na funcao Func_swap...");
    printf("a = %d \t b = %d \n", a, b);
}
```

```
a = 666          b = 111
Na funcao Func_swap...a = 111      b = 666
No MAIN e apos executar Func_swap: a = 666          b = 111
```

## Funções – Exemplo: passagem de parâmetros por endereço ie, via Ponteiro

```
#include <stdio.h>
void Fun_swap(int *x, int *y); //prototipo

int main(){
    int a=666, b=111;
    printf("a = %d | b = %d \n", a, b);
    Fun_swap(&a, &b);
    printf("Apos executar Fun_swap");
    printf("a= %d | b = %d \n", a, b);

    return(1);
}

void Fun_swap(int *x, int *y){
    int aux = *x; //salva valor de *x em aux
    *x = *y;
    *y = aux;
    printf("Fun_swap:");
    printf("*x = %d | *y = %d \n", *x, *y);
}
```

```
a = 666 | b = 111
Fun_swap:
*x = 111 | *y = 666
Apos executar Fun_swap
a= 111 | b = 666
```



## Funções - Passagem de tabelas ou *strings*

Quando uma função recebe o valor associado ao **nome da tabela**, este consiste no **endereço do primeiro elemento** da tabela (pex: `&tab[0]`).

Portanto, a função fica a com a possibilidade de alterar os valores da tabela. Ou seja, ao passar o nome de uma tabela para uma função estamos a usar a passagem por ponteiro.

Como o nome da tabela já é o ponteiro para o primeiro elemento da tabela, não precisamos de usar o operador **&** antes do nome da tabela ao invocar a função com o nome da tabela como argumento.

## Funções – Exemplo: Passagem de *arrays* e a sua capacidade

```
#include <stdio.h>

void func_replace(int tab[], int N, int x)
{
    for(int i=0;i<N;i++) tab[i]=x;
}

int main() {
    int n=6, x = 0;
    int W[6]={-1,-2,33,4,-5,0};

    x = 1;
    // O nome da tabela W corresponde a &W[0]
    func_replace(W, n, x);

    // Mostra valores da tabela
    for(int i=0; i<n; i++)
        printf("W[%d] = %d \n", i, W[i]);

    return (1);
}
```

```
W[0] = 1
W[1] = 1
W[2] = 1
W[3] = 1
W[4] = 1
W[5] = 1
```

Uma função pode devolver  
no máximo um valor

# Funções – calcula a média e a média-arredondada

```
#include <stdio.h>

int func_mean(int x[], int N, float *mu){
    float S=0.0;

    for (int i=0; i<N;i++)    S+=x[i];

    *mu = S/N;
    S = (int) (S/N + 0.5); //arredonda a media
    return S;
}

int main() {
    int tab[]={3,7,-1,2,-1,0,5};
    int n=7;
    float mean = 0.0;

    printf("Media arredondada = ");
    printf(" %d \n", func_mean(tab, n, &mean));
    printf("Media = %0.3f \n", mean);

    return (1); }
```

Alternativa:

```
func_mean(int *x, int N, float *mu);
```

```
Media arredondada = 2
Media = 2.143
```