Ficha de Trabalho N.º 3
Versão 2024/25

Objetivos: Estudo e utilização de Funções e *Arrays*.

Conceitos necessários à resolução da ficha

Funções

Funções: conceito e utilidade

É frequente em linguagens de programação designarem-se **subprogramas** os conjuntos de instruções que desempenham uma tarefa específica, aos quais é atribuído um determinado nome e cuja execução acontece quando se referenciam em qualquer ponto do programa.

Distinguem-se usualmente dois tipos de *subprogramas ou rotinas: os* **procedimentos** e as **funções.**
Uma **função** tem sempre um valor de retorno e um tipo associado, enquanto que um **procedimento** não devolve qualquer valor.

```
int minimo(int x, int y)
//devolve o menor de 2 números inteiros dados
{
    if(x < y)
        return x;
    else
        return y;
}
```

```
void linha(int nt)
//escreve uma linha de "comprimento" nt
{
    int i;
    for (i = 1; i <= nt; i++)
        printf("-");
    printf("\n");
}
```

Em C designamos por funções os dois tipos de subprogramas, embora usualmente se chamem procedimentos às funções que devolvem o valor *void*

Declaração e chamada

Declaração
SINTAXE
[tipo] **identificador** ([lista de argumentos formais, separados por vírgula (,)])
{
    bloco;
}

Chamada
SINTAXE
identificador ([lista de argumentos reais, separados por vírgula (,)]);

Exemplo

```
#include <stdio.h>
int minimo(int x, int y)
{
  if(x < y)
      return x;
    else
      return y;
}
int main()
{
   printf("\nO menor dos números 80 e 43 é: %d\n", minimo(80,43));
}
```

Declaração

Um programa em C tem de conter sempre no seu código a função **main**(), independentemente do número e variedade de funções existentes no programa.

Chamada

Algumas notas sobre declaração e chamada

Uma função tem sempre um tipo e um valor de retorno associado ao identificador, enquanto que um procedimento não devolve qualquer valor.

➢Sempre que no cabeçalho de uma função não é colocado o tipo de retorno, este é considerado do tipo *int*.

```
int minimo(int x, int y)
{
...
}
```
⟺
```
minimo(int x, int y)
{
   ...
}
```

O tipo deve ser sempre explícito.

Uma função pode ser invocada de diferentes formas:

➢*Dentro de uma atribuição:*

```
m = minimo(9,-2);
```

➢*Dentro de uma função, aproveitando o valor de retorno como parâmetro:*

```
printf("Menor valor: %d", minimo(8,18) );
```

ou

```
if (minimo(7,6) > 10)
    ...
```

➢*Tal como se invoca um procedimento, isto é, sem valor de retorno:*

```
getchar(); // aguardar que seja digitado um caractere
```

Características de uma função

1. Uma função tem um IDENTIFICADOR (nome) que serve para a sua invocação;
2. Uma função pode ser INVOCADA a partir de outras funções;
3. Como a sua designação indica, uma função deve realizar UMA ÚNICA TAREFA, bem definida;
4. Uma função deve comportar-se como uma CAIXA NEGRA: não importa como funciona, mas sim que dê o resultado esperado, sem efeitos colaterais;
5. O CÓDIGO de uma função deve ser tão GENÉRICO quanto possível, para poder ser reutilizado noutros projectos;
6. De forma a adaptar-se a situações distintas, uma função PODE RECEBER PARÂMETROS que alterem o seu comportamento;
7. Como resultado do seu trabalho, uma função PODE RETORNAR UM VALOR para a entidade que a invocou.

Identificador de uma função

1. A escolha do identificador de uma função obedece às regras anteriormente apresentadas para designação de variáveis;
2. O nome de uma função deve ser único: distinto do de qualquer variável ou de outra função;
3. Deve procurar-se que o identificador de uma função facilite a leitura e interpretação daquilo que ela realiza (Exemplo: **linha** *versus* xpto).

A instrução RETURN

Esta instrução permite terminar a execução de uma função e voltar ao programa que a invocou.

A seguir à instrução RETURN é possível colocar qualquer expressão válida em C.

```
int resto(int x, int y)
{
      return x % y;
}
```

Uma função pode ter várias instruções RETURN. Contudo, só uma delas é executada.

```
int resposta(int x)
{
   if (x > 0)
     return 1;
   else
     return 0;
}
```

Muita atenção à existência de vários *return*!!!!!

➢ **Na função *main*, a instrução RETURN faz com que o programa termine**

Variáveis locais

As variáveis declaradas dentro do próprio corpo de uma função apenas são *visíveis* (conhecidas) dentro dessa função. São por isso denominadas VARIÁVEIS LOCAIS.

A declaração de variáveis dentro de uma função deve ser realizada antes de qualquer instrução.

```
função (..........)
{
    declaração de variáveis
    instruções
}
```

```
#include <stdio.h>
void linha(int nt);
void main()
{
   int i=11;
   printf("\nBoa tarde!\n");
   linha(i);
}
void linha(int nt)
{
   int i;
   for (i=1; i<=nt; i++)
       printf("-");
   printf("\n");
}
```

Protótipo da função linha

Depois de terminada a execução de uma determinada função, todas as suas variáveis locais são destruídas.

Exemplo: Verificar se uma dada letra é vogal

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
int isvogal(char c)
{
      switch (tolower(c))
      {
         case 'a':
         case 'e':
         case 'i':
         case 'o':
         case 'u': return 1;
         default : return 0;
      }
}
void main()
{
      char letra;
      printf("\n\nDigite uma letra: ");
      scanf(" %c",&letra);
      printf("\nA letra lida, %c, ", letra);
      if (isvogal_atribuicao(letra))
            printf(" é uma vogal\n");
      else
            printf(" não é uma vogal\n");
      system ("pause");
      printf("\nConversao da letra %c em maiúscula: %c\n", letra, toupper(tolower(letra)));
      system ("pause");
}
```

Escrever um função equivalente, usando uma instrução de atribuição em vez da estrutura *switch*

```
int isvogal_atribuicao(char c)
{
    char cc = tolower(c);
    return (cc=='a' || cc== 'e' || cc== 'i' || cc== 'o' || cc== 'u');
}
```

Mais funções...

int resto (int a, int b) // Devolve o resto da divisão de a por b

```
int resto(int a, int b)
{
      return a % b;
}
```

int minimo (int a, int b) // Devolve Mínimo{a,b}

```
int minimo(int a, int b)
{
      return a<b ? a : b;
}
```

int impar (int x) // Devolve Verdade se x for impar e Falso c.c.

```
int impar(int x)
{
      return resto(x,2)!=0;
}
```

int abs (int a) // Devolve |a|

```
int abs(int a)
{
      return a>=0 ? a : -a;
}
```

int perfeito (int n) // Devolve Verdade se n for "perfeito" (igual à soma dos divisores de n, inferiores a n) e Falso c.c.

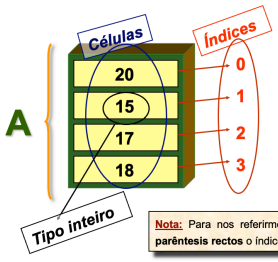
```
int perfeito(int n)
{
      int div, soma, meio;
      soma = 0;
      meio = n/2;
      for (div=1; div<=meio; div++ )
            if (resto(n,div)==0)  //se div é divisor de n
                  soma += div;
      return n==soma;
}
```

Arrays

Arrays

Conceito e utilidade

Por vezes torna-se necessário/conveniente referenciar uma zona de memória onde estão guardados vários valores de um mesmo tipo de dados, utilizando um mesmo nome. Para o efeito, utilizam-se variáveis do tipo estruturado: *Arrays*.



Array - trata-se de um tipo de variável estruturada, constituída por um conjunto de "células", identificadas univocamente por um "índice" (endereço), cujo conteúdo é constituído por um valor de determinado tipo.

Nota: Para nos referirmos a cada um dos seus valores, utilizamos o nome do Array e **entre parêntesis rectos** o índice da célula onde se encontra o valor.

Declaração de *Arrays* em C

SINTAXE
tipo identificador **[número de elementos]**

Tipo → estabelece o tipo de dados de cada componente do *array*
Identificador → nome pelo qual o *array* vai ser conhecido
Número de elementos → valor constante que indica quantos elementos tem o *array*

- Em C os índices de um *array* com *n* elementos variam sempre entre 0 e *n*-1
- O índice da primeira componente é 0 (zero) e o da *n*-ésima é *n*-1

Na declaração

```
int notaAP[100]; // notaAP é um vector com 100 inteiros

int          - tipo de cada um dos elementos do vector
notaAP       - nome do vector
100          - número de elementos do vector
notaAP[i]    - aquilo que está na i-ésima componente do vector
```

Passagem de *arrays* unidimensionais para funções

Em C, dentro de uma função, não é possível saber com quantos elementos foi declarado um *array* passado como argumento para essa função. A dimensão a considerar é da exclusiva responsabilidade do programador.

Exemplo

```
#include <stdio.h>
void inicializa (int s[ ], int n)
{
   int i;
   for (i=0; i<=n-1; i++)
         s[i] = i;
}
main()
{
   int x[15], y[20];
   inicializa(x, 15);
   inicializa(y, 20);
   printf("\n");
}
```

Qual o valor das componentes do vector x e do vector y imediatamente antes de o programa terminar?

Inicialização. Índices.

- É possível inicializar as componentes de um *array* logo na declaração. → **tipo** identificador [n] = {valor$_1$,..., valor$_n$}

Exemplo: char vogal[5] = {'a', 'e', 'i', 'o', 'u'}; ⟺

```
char vogal[5];
vogal[0]='a';
vogal[1]='e';
vogal[2]='i';
vogal[3]='o';
vogal[4]='u';
```

- Se um *array* for declarado com n elementos e apenas k (k<n) forem inicializados, os restantes ficam inicializados com o valor 0 (ESPAÇO - caractere nº zero - tratando-se do tipo char) .

Exemplo: int v[5] = {10,15,20}; ⟺ `int v[5] = {10,15,20,0,0};`

- As posições de um *array* só são inicializadas se a declaração for seguida de = {...}
- NOTA: O compilador não detecta se os índices utilizados estão fora da gama declarada.

Arrays multidimensionais

SINTAXE
tipo identificador **[dim$_1$] [dim$_2$] ... [dim$_n$];**

Array uni-dimensional - (1-D)

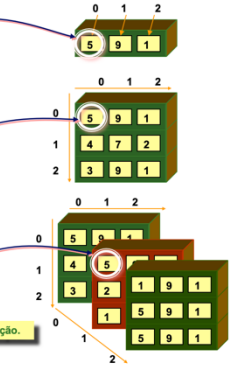
```
int UniArray [3]
UniArray[0] = 5;
```

Array bi-dimensional - (2-D)

```
int BiArray [3] [3]
BiArray[0][0] = 5;
```

Array tri-dimensional - (3-D)

```
int TriArray [3][3][3]
TriArray[0][0][1] = 5;
```



- É possível inicializar as componentes de um *array* multidimensional logo na declaração.

Exemplo: `int v[2] [3] = {{1, 2, 3}, {4, 5, 6}};`

Passagem de *arrays* multidimensionais para funções

- A passagem de *arrays* com mais de uma dimensão para uma função realiza-se indicando no cabeçalho, obrigatoriamente, o número de elementos da cada uma das dimensões (a dimensão mais à esquerda pode ser omitida, colocando apenas [ ] ou *)

Ler uma matriz

```
void leMatriz(int matriz[MAX][MAX], int tamL, int tamC)
 // Funçao para leitura de uma matriz tamLxtamC
{
     int i,j;
     for (i = 0; i < tamL; i++)
        for (j = 0; j < tamC; j++)
        {
            printf("\n(%d,%d])= ", i+1, j+1);
            scanf("%d",&matriz[i][j]);
        }
}
```

Faça uma simulação para 3x3

Problemas Propostos

1) Escreva as seguintes funções sobre o tipo char:

	Função	Devolve
a)	**int isDigit(char c)**	Verdade quando c é um dígito e Falso c.c.
b)	**int isAlpha(char c)**	Verdade quando c é uma letra e Falso c.c.
c)	**int isAlNum(char c)**	Verdade quando c é um carácter alfanumérico e Falso c.c.
d)	**char toLower(char c)**	Devolve c transformado na minúscula correspondente
e)	**char toUpper(char c)**	Devolve c transformado na maiúscula correspondente

Nota: Obtém-se acesso a estas funções através da directiva (só que com o nome todo em minúsculas) em:
#include <ctype.h> // Funções sobre o tipo char (*ctype -> char type*)

2) a) Escreva as funções a seguir indicadas de modo que devolvam os resultados descritos:

	Função	Devolve
a)	**int resto (int a, int b)**	O resto da divisão de a por b
b)	**int impar (int x)**	Verdade se x for impar e Falso c.c.
c)	**int perfeito (int n)**	Verdade se n for "perfeito" (igual à soma dos divisores de n, inferiores a n) e Falso c.c.
d)	**int primo (int n)**	Verdade se n for "primo" (apenas divisível por 1 e por n) e Falso c.c.

b) Crie um main() que permita testar as funções criadas nos exercícios 1 e 2.

3) No século I d.C. os números naturais dividiam-se em três categorias:

REDUZIDOS: os superiores à soma dos seus divisores;
ABUNDANTES: os inferiores à soma dos seus divisores;
PERFEITOS: os que são iguais à soma dos seus divisores.

NOTA: Nesta definição exclui-se o próprio número do conjunto dos seus divisores.

Escreva uma função que liste os inteiros entre a e b, a<b, classificando-os de acordo com esse critério, e que escreva também o total de cada uma das categorias.
Crie um main() eu permita testar a função criada.

4) Elabore funções que determinem:

a. O cubo de um número inteiro **n.** O número **n** deve ser pedido ao utilizador através de uma função (denominada **leitura**) e o seu **cubo** deve ser calculado através de outra função (de nome **cubo**).

b. Copiar a função cubo criada na alínea a., e alterá-la, criando a função **exponenciação**, de forma a torná-la mais genérica: calcular **x^exp.** Os números **x** e **exp** devem ser solicitados ao utilizador através da função **leitura**. De igual modo, **x^exp** deve ser calculado através da função **exponenciação**.

Obs. No final, teste e corrija as funções, criando um main() para o efeito.

5) Escreva uma função que determine o maior de dois números dados. Teste a função num pequeno programa.

6) Elabore um programa que:
   a) Leia as *n* componentes de um vetor;
   b) Escreva as n componentes de um vetor;
   c) Determine a posição em que se encontra a maior componente.

   Cada uma destas tarefas deve ser realizada por uma função e testadas / corrigidas com o auxilio de um outro pequeno programa.

7) Escreva uma função que determine o produto interno de dois vetores de n componentes inteiras.

8) Dado um vetor x com n componentes inteiras, escreva funções que permitam realizar as seguintes operações:
   a) Trocar a componente da posição p com a da posição q;
   b) Efetuar a permutação circular do vetor dado.

Obs.: Crie um main() que lhe permita testar as funções criadas nos exercícios 7 e 8.

9) Considere uma matriz quadrada com nxn elementos inteiros.
   Elabore um programa que lhe permita:
   - Ler os *n*x*n* elementos da matriz;
   - Mostrar no monitor os *n*x*n* elementos da matriz;
   - Determinar o valor mínimo da matriz;
   - Verificar se a matriz é ou não simétrica;
   - Determinar a transposta da matriz;
   - Calcular a soma de duas matrizes dadas.

   Cada tarefa deve ser realizada por uma função e deve ser testada / corrigida com o auxilio de um outro pequeno programa.