

# Linguagem C – 2ª parte

18/09/2023

# Sumário

- Recap
- Ponteiros
- Gestão da memória
- Funções : call-by-value e call-by-pointer
- Validação de condições : assert(...)
- Tópicos adicionais
- Referências

# Recapitulação

Let's  
Recap

# hello.c

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

```
/* This is a  
   comment */
```

```
int main(void) {  
    // Another comment  
    printf("Hello world!\n");  
    return 0;  
}
```



A indentação torna  
o código mais legível !!

# stdio.h – printf – Output formatado

```
printf(formatting string, param1, param2, ...);
```

```
printf(“There are 220 students in AED\n”);
```

```
printf(“There are %d students in %s\n”, 220, “AED”);
```

```
int x = 10;
```

```
int y = 20;
```

```
printf(“%d + %d = %d\n”, x, y, x + y);
```

# stdio.h – scanf – Input formatado

```
scanf(formatting string, &param1, &param2, ...);
```

```
int my_num;  
char my_char;  
printf("Type a number AND a character and press enter: \n");  
scanf("%d %c", &myNum, &myChar); ←  
printf("Your number is: %d and your char is %c\n",my_num,my_char);
```

```
char first_name[30]; // Array of chars to store a string  
printf("Enter your first name:\n");  
scanf("%s", first_name); ←  
printf("Hello %s\n", first_name);
```

# Compilação e execução

- Linux

```
cc source_file.c -> ./a.out
```

```
cc -Wall source_file.c
```

```
cc -Wall my_file.c -o exec_name -> ./exec_name
```

- Windows

```
gcc source_file.c -> .\a.exe
```

```
gcc -Wall source_file.c
```

```
gcc -Wall my_file.c -o exec_name -> .\exec_name
```

# Ponteiros



# Ponteiros

- **Endereço de uma variável** : índice do 1º byte do bloco de memória que armazena a variável
- **Ponteiro** : variável que contém o endereço de outra variável

```
int i;           // Variável inteira
int* ptr;       // Ponteiro para variável inteira que
                // referencia uma localização aleatória
int* another_ptr = NULL; // Mais seguro !
```

# NULL pointer

- NULL : valor especial que representa o ponteiro não inicializado

```
int* ptr = NULL;
.
.
if(ptr == NULL) {
    printf("Cannot dereference a NULL pointer!\n");
    exit(1);
}
.
```

# Ponteiros

- **& address operator** : devolve o endereço de uma variável
- **\* dereference operator** : acede à variável apontada

```
int i = 10;
int* ptr_i = &i;      // Ponteiro para a variável i
int* another_ptr = ptr_i;
*ptr_i = 5;
*another_ptr = 20;
printf(“%d\n”, i);    // Output ?
```


# I. Horton – Exemplo – Output ?

```
#include <stdio.h>

int main(void)
{
    char multiple[] = "My string";

    char *p = &multiple[0];
    printf("The address of the first array element : %p\n", p);

    p = multiple;
    printf("The address obtained from the array name: %p\n", multiple);
    return 0;
}
```




# I. Horton – Exemplo – Output ?

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

int main(void)
{
    char multiple[] = "a string";
    char *p = multiple;

    for(int i = 0 ; i < strlen_s(multiple, sizeof(multiple)) ; ++i)
        printf("multiple[%d] = %c *(p+%d) = %c &multiple[%d] = %p p+%d = %p\n",
            i, multiple[i], i, *(p+i), i, &multiple[i], i, p+i);
    return 0;
}
```



# Gestão da memória

# C vs Java – Gestão da memória

- Em Java, objetos são alocados dinamicamente e o espaço de memória é gerido de modo automático
- Em C, as estruturas de dados são alocadas dinamicamente ou de modo estático
- O tamanho de **estruturas de dados estáticas** é fixado em tempo de compilação e não sofre qualquer alteração durante a execução de um programa
- O espaço de memória de **estruturas de dados dinâmicas** é alocado e libertado durante a execução de um programa

# Variáveis globais

- Declaradas no exterior das funções que compõem o programa
- Se necessário, inicializadas antes da execução do programa
- Espaço de memória alocado de modo estático antes da execução do programa
- Espaço alocado não é libertado antes do final da execução
- **REGRA** : usar apenas em situações particulares !!



# Variáveis locais

- Declaradas no corpo de uma função
- Espaço de memória alocado após a chamada da função (function call)
- Espaço de memória automaticamente libertado no final da execução da função
- **REGRA** : não aceder, p.ex., usando um ponteiro, a variáveis locais após o final da execução de uma função

# Heap variables – Alocação dinâmica

- Memória é alocada explicitamente usando **malloc** ou **calloc**
  - Semelhante a **new** em Java
  - `void* malloc( int )`
  - `void* calloc( int, int )`
- A memória alocada tem de ser libertada, usando **free**, para que a memória do sistema não se esgote
  - **Não** há “**garbage collection**” como em Java
  - `void free( void* )`
- Responsabilidade acrescida do programador !!

# Exemplo

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

int no_alloc_var; /* global variable counting number of allocations */

void main(void) {
    int* ptr; /* local variable of type int* */

    /* allocate space to hold an int */
    → ptr = (int*) malloc(sizeof(int));
    no_alloc_var++;

    /* check if successfull */
    if (ptr == NULL)
        exit(1); /* not enough memory in the system, exiting */

    *ptr = 4; /* use the memory allocated to store value 4 */

    → free(ptr); /* deallocate memory */
    no_alloc_var--;
}
```

# Funções

# Funções

- **Desenvolvimento modular** da solução
- Facilitar a **implementação** e a **depuração** de erros
- Permitir a **reutilização** de código
- Passagem de argumentos por valor – **Call-by-value**
  - A função recebe uma **cópia** do valor da variável passada à função
- Passagem de argumentos por ponteiro – **Call-by-pointer**
  - A função recebe um **ponteiro** para a variável passada à função
- Resultado devolvido por valor ou por ponteiro

# Exemplo

```
#include <stdio.h>

int sum(int a, int b); /* function declaration or prototype */
int psum(int* pa, int* pb);

void main(void) {
    int total=sum(2+2,5); /* call function sum with parameters 4 and 5 */

    printf("The total is %d.\n",total);
}

/* definition of function sum; has to match declaration signature */
int sum(int a, int b){ /* arguments passed by value */
    return (a+b); /* return by value */
}

int psum(int* pa, int* pb){ /* arguments passed by reference */
    return ((*a)+(*b));
}
```

# swap – Call-by-value – Output ?

```
#include <stdio.h>

void swap(int, int);

void main(void) {
    int num1=5, num2=10;
    swap(num1, num2);
    printf("num1=%d and num2=%d\n", num1, num2);
}

void swap(int n1, int n2) { /* pass by value */
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

# swap – Call-by-pointer – Output ?

```
#include <stdio.h>

void swap(int*, int*);

void main(void) {
    int num1=5, num2=10;
    int* ptr = &num1;
    swap(ptr, &num2);
    printf("num1=%d and num2=%d\n", num1, num2);
}

void swap(int* p1, int* p2) { /* pass by reference */
    int temp;
    temp = *p1;
    (*p1) = *p2;
    (*p2) = temp;
}
```



# Command-line arguments

# I. Horton – Exemplo – Output ?

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("Program name: %s\n", argv[0]);
    for(int i = 1 ; i<argc ; ++i)
        printf("Argument %d: %s\n", i, argv[i]);
    return 0;
}
```

# Validação de condições

# assert.h – assert(...)

- **Avaliar** o valor de **expressões** arbitrárias, cujo resultado é um valor inteiro
- Se **zero** (i.e., falso), **terminar a execução** do programa, com uma eventual mensagem de diagnóstico apropriada
- Utilidade ?
- Avaliar **pré-condições** (à entrada) e **pós-condições** (à saída) de funções
- Por exemplo, assegurar que **ponteiros** não são **NULL** – Guião 02

# I. Horton – Exemplo simples – Output ?

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

int main(void)
{
    int y = 5;
    for(int x = 0 ; x < 20 ; ++x)
    {
        printf("x = %d    y = %d\n", x, y);
        assert(x < y);
    }
    return 0;
}
```

# Tipos enumerados

# enum : enumerated data-types

```
enum months { JANUARY, FEBRUARY, MARCH };
```

- Cada elemento tem associado um valor inteiro
  - Valores associados por **omissão**, começando em **zero**

```
enum months {  
    JANUARY = 1,  
    FEBRUARY,    // 2  
    MARCH        // 3  
};
```

# Registos (struct)



# struct

- **Registo** agrega campos (**data members**) de vários tipos

```
struct birthday {  
    char* name,  
    unsigned int year;  
    enum months month,  
    unsigned int day  
};
```

## struct - . vs ->

```
struct birthday bday_joao = {"joao", 2000, 1, 1};  
char first_letter = bday_joao.name[0];  
bday_joao.month = FEBRUARY;
```

```
struct birthday* ptr = &bday_joao;  
printf("%s\n", ptr->name);
```

# Exemplos

```
struct point {  
    int x, y;  
} p1;  
struct point p2;
```

```
struct {  
    struct point c;  
    unsigned int rad;  
} circ1, circ2;  
// No way to declare another variable of the same struct now
```

# typedef – Sinónimo / Designação alternativa

- Definir novos tipos (auxiliares)
- Facilitar a escrita, compreensão e depuração do código

```
typedef struct {  
    struct point c;  
    unsigned int rad;  
} circle;  
  
circle circ1, circ2;
```

# Referências

# Referências

George Ferguson, *C for Java Programmers*, 2017

Ivor Horton, *Beginning C*, 4th Ed, Apress, 2006

Tomás Oliveira e Silva, *AED Lecture Notes*, 2022