


Linguagem C – 2ª parte

17/09/2025

Sumário

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

Let's
RECAP

Recapitulação

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

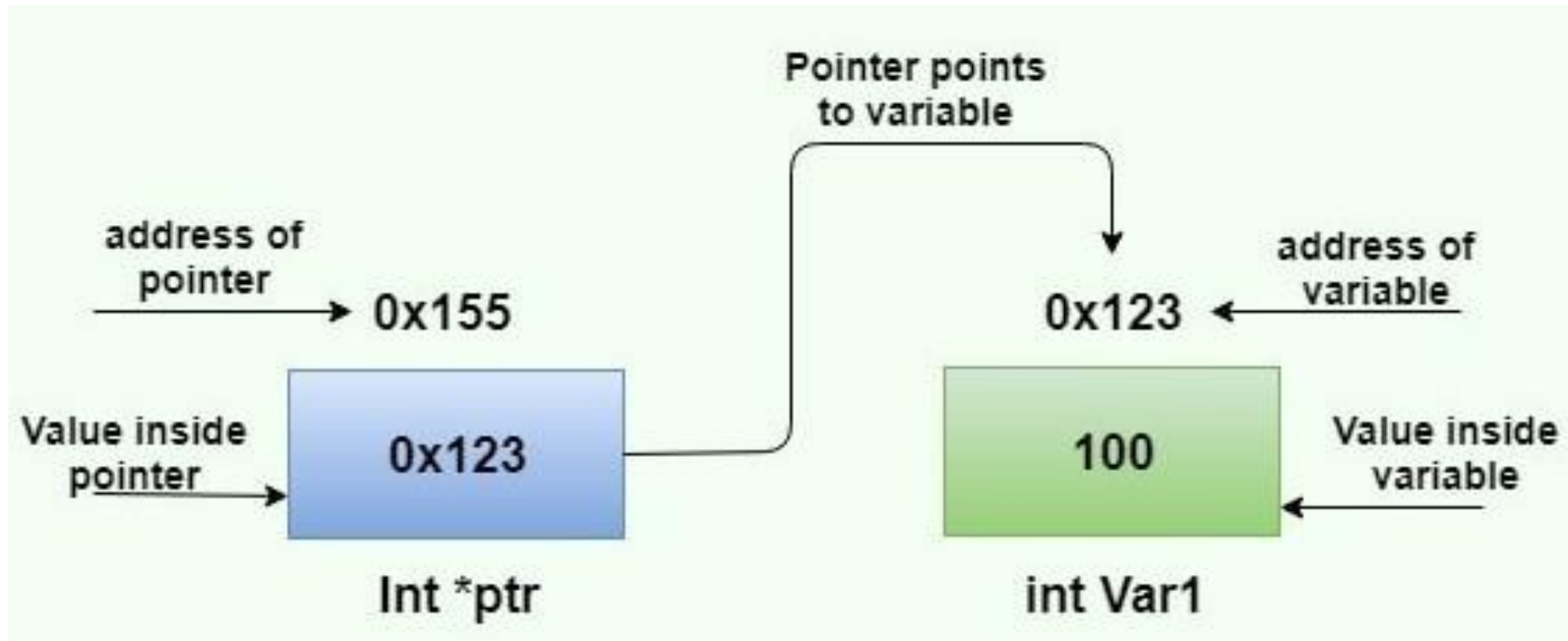
```
scanf(formatting string, &param1, &param2, ...);
```

```
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);
```

Ponteiros

Ponteiro armazena um endereço de memória



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

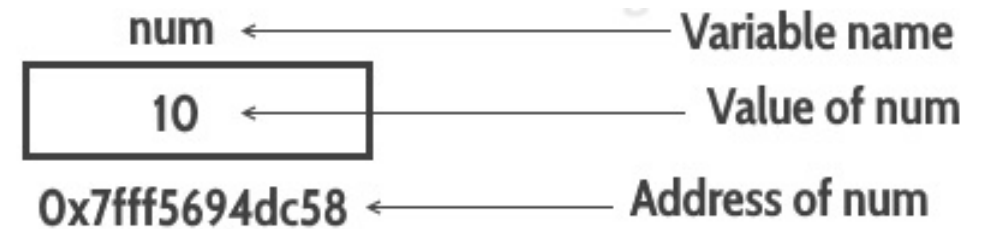
```
int* ptr = NULL;
.
.
if(ptr == NULL) {
    printf("Cannot dereference a NULL pointer!\n");
    exit(1);
}
.
```

Ponteiros – Operadores

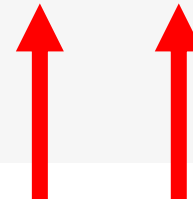
- **& address operator** : devolve o endereço de uma variável
- *** dereference operator** : acede à variável apontada (i.e., referenciada)

```
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 ?
```

& – Obter o endereço de uma variável



```
int main(void) {  
  
    int num = 10;  
    printf("Value of variable num is: %d", num);  
  
    /* To print the address of a variable we use the %p  
     * format specifier and the ampersand (&) sign just  
     * before the variable name like &num  
     */  
    printf("\nAddress of variable num is: %p", &num);  
  
    return 0;  
}
```



* – Aceder à variável apontada

```
int a = 10;  
int b = 99;  
printf("Value of a + b is: %d\n", a + b);
```

```
int* p_a = &a;  
int* p_b = &b;  
int sum = *p_a + *p_b; ←
```

```
→ *p_a = 1000;  
→ *p_b = 9999;
```

```
printf("Value of sum is: %d\n", sum);           // Output ?
```

```
printf("Value of a + b is: %d\n", a + b);       // Output ?
```

Array – Ponteiro para o 1º elemento

```
int my_array[3] = {1, 2, 3};

int* p_my_array = my_array;           // Pointer to first element !!

my_array[0] = 10;

*my_array = 20;

*p_my_array = 30;

printf("New value of first element: %d\n", my_array[0]); // Output ?
```

Array – Aritmética de ponteiros

- Aceder a cada elemento de um array usando um ponteiro
- **Atenção:** Não ultrapassar o último elemento !!

```
int my_array[3] = {1, 2, 3};
```

```
*my_array = 100;           // my_array[0]
```

```
*(my_array + 1) = 200;     // my_array[1]
```

```
*(my_array + 2) = 300;     // my_array[2]
```

```
printf("Value of second element is: %d\n", my_array[1]); // Output ?
```

Array – Iterar ao longo de um array

- Aceder a cada elemento usando um ponteiro
- Que é **sucessivamente incrementado**
- **Atenção:** Não ultrapassar o último elemento !!

```
int my_array[3] = {1, 2, 3};  
int* p = my_array;
```

```
for(int i = 0; i < 3; i++) {  
    printf("%d\n", *p);  
    p = p + 1;  
}
```

// Output ?
// Increment pointer

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 – Alocar espaço para um inteiro

```
int* ptr = NULL;
```

```
→ ptr = (int*)malloc(sizeof(int));    // Try to allocate memory

if(ptr == NULL) {                      // Memory allocation failed
    exit(1);
}

*ptr = 99;
printf("%d\n", *ptr);

→ free(ptr);                          // Free allocated memory
```

Exemplo – Alocar espaço para um array

```
int* a = NULL;
```

→

```
a = (int*)malloc(3 * sizeof(int)); // Try to allocate memory
```

```
if(a == NULL) { ↑ // Failed...  
    exit(1);  
}
```

```
a[0] = 99;
```

```
...
```

```
...
```

→

```
free(a); // Free allocated memory
```

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
 - Valor original dessa variável não é alterado !!
- Passagem de argumentos por ponteiro – Call-by-pointer
 - A função recebe um ponteiro para a variável passada à função
- Resultado pode ser devolvido por valor ou por ponteiro

Call-by-value vs Call-by-pointer

```
int sum(int a, int b) {                // Call-by-value
    return a + b;
}
int p_sum(int* p_a, int* p_b) {        // Call-by-pointer
    return (*p_a) + (*p_b);
}
...

int m = 10;
int n = 20;

int res = sum(m, n);
...
res = p_sum(&m, &n);                  // Call-by-pointer
```



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;
}
```

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

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;
}
```



Command-line arguments

Input de argumentos pela linha de comando

- Passar **argumentos ao iniciar a execução** de um programa
- Esses argumentos/parâmetros são passados para a função **main**
- Nesses casos, a função main tem **dois parâmetros de entrada**
- **argc** : valor inteiro, número de “palavras” que constituem a linha de comando
- **argv** : array de ponteiros para cadeias de caracteres, com **argc** elementos
- O **nome do programa** é referenciado por **argv[0]**
- O **primeiro argumento**, caso exista, é referenciado por **argv[1]**

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;
}
```

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  
};
```

Acesso aos campos de um registo – . vs ->

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

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



typedef

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;
```



Exercícios / Tarefas


Ponteiro para um array – Qual é o 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;
}
```




Ponteiros para um array – Qual é o 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;
}
```



String – Alocar e libertar memória – Output ?

```
int main(void) {  
    char* str = (char *)malloc(20 * sizeof(char));  
    if (str == NULL) {  
        printf("Memory allocation failed!\n");  
        return 1;  
    }  
    strcpy(str, "Hello, World!");           // Copy into the allocated memory  
    printf("String: %s\n", str);  
    free(str);  
    return 0;  
}
```

String – Aritmética de ponteiros – Output ?

```
int main(void) {  
    char str[] = "Hello, World!";  
    char *ptr = str;  
    printf("String characters using pointer arithmetic:\n");  
    while (*ptr != '\0') {                // Loop until the null character  
        printf("%c ", *ptr);  
        ptr++;                            // Move to the next character  
    }  
    printf("\n");  
    return 0;  
}
```

struct Point e ModifyPoint() – Call-by-pointer

```
struct Point {  
    int x;  
    int y;  
};
```

```
void ModifyPoint(struct Point *p) {  
    p->x = p->x + 10;  
    p->y = p->y + 20;  
}
```

Qual é o output ?

```
int main(void) {  
    struct Point p1 = {5, 10};  
    printf("Before ModifyPoint: x = %d, y = %d\n", p1.x, p1.y);  
    ModifyPoint(&p1);  
    printf("After ModifyPoint: x = %d, y = %d\n", p1.x, p1.y);  
    return 0;  
}
```


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