

# Ponteiros em Linguagem C

## Linguagem de Programação Estruturada – 2016.1

### 1. CONCEITO

- Em linguagem **C**, ponteiros ou apontadores, são variáveis que armazenam o endereço de memória de outras variáveis.
- Dizemos que um ponteiro “aponta” para uma variável quando contém o endereço da mesma.
- Os ponteiros podem apontar para qualquer tipo de variável. Portanto temos ponteiros para int, float, double, etc.

### Por que usar ponteiros?

- Ponteiros são muito úteis quando uma variável tem que ser acessada em diferentes partes de um programa.
- Existem várias situações onde ponteiros são úteis, por exemplo:
  - Alocação dinâmica de memória
  - Manipulação de arrays.
  - Referência para listas, pilhas, árvores e grafos.

## 2. ENDEREÇOS

- A memória RAM de qualquer computador é uma sequência de bytes. Os bytes são numerados sequencialmente e o número de um byte é o seu endereço (= *address*).
- Cada objeto na memória do computador ocupa um certo número de bytes consecutivos.
  - Um ***char*** ocupa 1 byte.
  - Um ***int*** ocupa 4 bytes.
  - um ***double*** ocupa 8 bytes em muitos computadores.
- O número exato de bytes de um objeto é dado pelo operador ***sizeof***: a expressão ***sizeof*** (int), por exemplo, dá o número de bytes de um int no seu computador.

## 2. ENDEREÇOS

- Cada objeto na memória tem um *endereço*.
- Por exemplo, depois das declarações:

```
char c;
int i;
struct {
    int x, y;
} ponto;
int v[4];
```

os endereços das variáveis poderiam ser os seguintes:

c	89421
i	89422
ponto	89426
v[0]	89434
v[1]	89438
v[2]	89442

## 2. ENDEREÇOS

O endereço de um objeto (como uma variável, por exemplo) é dado pelo operador **&**. Se **i** é uma variável então

**&i**

é o seu endereço.

**Obs.:** Não confundir esse uso de "&" com o operador lógico **and**, que se escreve "&&" em C. No exemplo anterior, **&i** vale 89422 e **&v[3]** vale 89446.

- Um exemplo: O segundo argumento da função de biblioteca **scanf** é o endereço da variável onde deve ser depositado o objeto lido do dispositivo padrão de entrada:

```
int i;
scanf ("%d", &i);
```

## 3. Tamanhos

Qual o resultado do código abaixo?

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

int main (void) {
    typedef struct {
        int dia, mes, ano;
    } data;
    printf ("sizeof (data) = %d\n",
           sizeof (data));

    system("pause");
}
```

## 4. Ponteiros

- Um *ponteiro* (= apontador = *pointer*) é um tipo especial de variável que armazena endereços.
- Se um ponteiro *p* armazena o endereço de uma variável *i*, podemos dizer "*p* aponta para *i*" ou "*p* é o endereço de *i*". (Em termos um pouco mais abstratos, diz-se que *p* é uma referência à variável *i*.)

### Sintaxe de declaração de ponteiro

tipo \*nome\_ponteiro;

Onde temos:

- **tipo** : é o tipo de dado da variável cujo endereço o ponteiro armazena.
- O asterisco \* neste tipo de declaração determina que a variável será um ponteiro.

## 4. Declaração de ponteiros

tipo \*nome\_ponteiro;

Onde temos:

- **tipo** : é o tipo de dado da variável cujo endereço o ponteiro armazena.
- O asterisco \* neste tipo de declaração determina que a variável será um ponteiro.

É o asterisco (\*) que faz o compilador saber que a variável não vai guardar um valor, mas sim um endereço para aquele tipo especificado.

O tipo do ponteiro define que tipo de variáveis o ponteiro pode apontar.

Exemplo de declaração de ponteiro:

**int \*ptr;**



## 5. Operadores de ponteiros

& , retorna o endereço de memória da variável;

Ex: p = &cont;

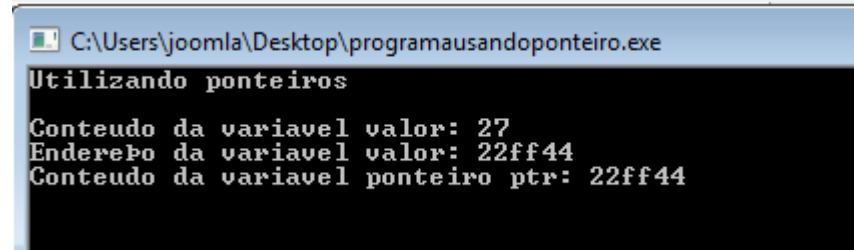
\* , retorna o valor da variável localizada no endereço que o segue;

Ex: q = \*p;

Exemplo:

```

1  #include <stdio.h>
2  #include <conio.h>
3
4
5  int main(void)
6  {
7      //valor é a variável que
8      //será apontada pelo ponteiro
9      int valor = 27;
10
11     //declaração de variável ponteiro
12     int *ptr;
13
14     //atribuindo o endereço da variável valor ao ponteiro
15     ptr = &valor;
16
17     printf("Utilizando ponteiros\n\n");
18     printf("Conteudo da variavel valor: %d\n", valor);
19     printf("Endereço da variavel valor: %x \n", &valor);
20     printf("Conteudo da variavel ponteiro ptr: %x", ptr);
21
22     getch();
23     return(0);
24 }
```



```

C:\Users\joomla\Desktop\programausandoponteiro.exe
Utilizando ponteiros
Conteudo da variavel valor: 27
Endereço da variavel valor: 22ff44
Conteudo da variavel ponteiro ptr: 22ff44
  
```

## 5. Operadores de ponteiros

Exemplo2:

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

int main()
{
    int x=4, y=7;
    int *px,*py;

    printf("&x = %p\t x = %d\n", &x , x);
    printf("&y = %p\t y = %d\n", &y , y);

    px = &x;
    py = &y;

    printf("px = %p\t*px = %d\n", px,*px);
    printf("py = %p\t*py = %d\n", py,*py);

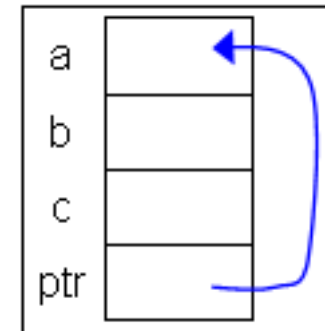
    system("PAUSE");
    return 0;
}
```

## 5. Operadores de ponteiros

Exemplo3:

```
#include <stdio.h>
#include<stdlib.h>
#include<conio.h>
int main()
{
    int a;
    int b;
    int c;
    int *ptr; // declara um ponteiro para um inteiro
              // um ponteiro para uma variável do tipo inteiro

    a = 90;
    b = 2;
    c = 3;
    ptr = &a;
    printf("Valor de ptr: %p, Conteúdo de ptr: %d\n", ptr, *ptr);
    printf("B: %d, C: %d", b, c);
    getch();
}
```



## 5. Operadores de ponteiros

Considerações Sobre o Exemplo2:

- A instrução

*int \*px, \*py*

declara **px** e **py** como ponteiros para variáveis **int**

- Quando um ponteiro não é inicializado na instrução de sua declaração, o compilador inicializa - o com o endereço zero (NULL)
- A linguagem C garante que NULL não é um endereço válido, então, antes de usá-lo, devemos atribuir a eles algum endereço válido, isto é, feito pelas instruções

*px = &x;*

*py = &y;*

- Um ponteiro pode ser inicializado na mesma instrução de sua declaração

*int \*px = &x, \*py = &y;*

## 5. Operadores de ponteiros

Se vários ponteiros são declarados em uma mesma instrução, o tipo deve ser inserido somente uma vez; o asterisco, todas as vezes

```
int* p, * p1, * p2;
```

```
int *p, *p1, *p2;
```

Inicializando ponteiros

```
int i;
```

```
int *pi=&i;
```

```
int *pj, *pi=&i, *px;
```

Ponteiros e variáveis simples declarados em uma única instrução

```
int *p, i, j, *q;
```

```
int *px=&x, i, j=5, *q;
```

## 5. Operadores de ponteiros

### CUIDADO

```
void main(void)
{
    float x, y;
    int *p;

    /* O próximo comando faz com que p (que é ponteiro
       para inteiro) aponte para um float. */
    p = &x;

    /* O próximo comando não funciona como esperado. */
    y = *p;
}
```

## 5. Operadores de ponteiros

```
/* PtrVar1.C */
/* Mostra a inicialização do ponteiro */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int x,y;
    int *px = &x;      /* Inicializa px com o endereço de x */

    *px = 14;          /* O mesmo que x = 14 */
    y = *px;           /* O mesmo que y = x */

    printf("y = %d\n", y);

    system("PAUSE");
    return 0;
}
```

- Nesse programa, usamos o ponteiro para atribuir um valor à variável **x**
- Em seguida, usamos novamente o ponteiro para atribuir esse valor a **y**
- O operador indireto resulta o nome da variável apontada

## 5. Operadores de ponteiros

Atividade:

1. Quais serão os valores de x, y e p ao final do trecho de código abaixo?

```
int x, y, *p;
y = 0;
p = &y;
x = *p;
x = 4;
(*p)++;
--x;
(*p) += x;
```



## 5. Operadores de ponteiros

Atividade:

1. Quais serão os valores de x, y e p ao final do trecho de código abaixo?

```
int x, y, *p;
y = 0;
p = &y; // *p = 0
x = *p; // x = 0
x = 4; // x = 4
(*p)++; // *p = 1, y = 1
--x; // x = 3
(*p) += x; // *p = 4, y = 4
```

Ao final, temos:

x = 3, y = 4, p apontando para y (\*p = 4).

## Linguagem de Programação Estruturada – 2016.1

### 5. Operadores de ponteiros

2.O programa abaixo possui erro(s). Qual(is)? Como deveria(am) ser?

```
void main() {  
    int x, *p;  
    x = 100;  
    p = x;  
    .....  
    printf("Valor de p: %d.\n", *p);  
}
```

## 5. Operadores de ponteiros

2.O programa abaixo possui erro(s). Qual(is)? Como deveria(am) ser?

```
void main() {  
    int x, *p;  
    x = 100;  
    p = x; //p deveria receber o endereço de x, já que p é um ponteiro (e x não).  
    Ponteiros "armazenam" o endereço para o qual eles apontam! O código correto  
    seria: p = &x;  
    printf("Valor de p: %d.\n", *p);  
}
```

## 5. Operadores de ponteiros

3.O programa abaixo possui erro(s). Qual(is)?

```
int *p1, *p2, x;
float *p3;

p1 = &x;    /* Correto */
p2 = p1;    /* Correto */
p3 = p1;    /* Incorreto. Compilador acusa "Warning". */
```

## 6. Operações com ponteiros

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

```
int main() {
```

```
    int *intPoint;
    char *chPoint;
    float *flPoint;
```

```
    printf("\n %d \n", intPoint); //endereço do ponteiro inteiro
```

```
    intPoint--;
```

```
    printf("\n %d \n", intPoint); //endereço do ponteiro deslocado em 4 bytes
```

```
    getch();
}
```

1960353332

1960353328

# Linguagem de Programação Estruturada – 2016.1

## 6. Operações com ponteiros

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

int main() {

    short *intPoint;
    char *chPoint;
    float *flPoint;

    printf("\n %d \n", intPoint); //endereço do ponteiro inteiro

    intPoint--;

    printf("\n %d \n", intPoint); //endereço do ponteiro deslocado em 2 bytes

    getch();
}
```

1960353332

1960353330

## 6. Operações com ponteiros

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

```
int main() {
```

```
    short *intPoint;
    char *chPoint;
    float *flPoint;
```

```
    printf("\n %d \n", intPoint); //endereço do ponteiro inteiro
```

```
    intPoint = intPoint + 4;
```

```
    printf("\n %d \n", intPoint); //endereço do ponteiro deslocado em 8 bytes
```

```
    getch();
}
```

```
1960353332
1960353340
```

# Linguagem de Programação Estruturada – 2016.1

## 6. Operações com ponteiros

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

int main() {

    short *intPoint;
    char *chPoint;
    float *flPoint;

    printf("\n %d \n", chPoint); //endereço do ponteiro caractere

    chPoint++;

    printf("\n %d \n", chPoint); //endereço do ponteiro deslocado em 1 byte

    getch();
}
```

2686792

2686793



# Linguagem de Programação Estruturada – 2016.1

## 6. Operações com ponteiros

```
int main()
{
    unsigned int x=5, y=6;
    unsigned int *px, *py;

    px = &x;    /* Atribuições */
    py = &y;

    if( px < py) /* Comparações */
        printf("py-px= %u\n", (py-px)); /* Subtração */
    else
        printf("px-py= %u\n", (px-py));

    printf("px = %p", px);
    printf(", *px = %u", *px);    /* Op.Indireto */
    printf(", &px = %p\n", &px); /* Op.Endereços */

    printf("py = %p", py);
    printf(", *py = %u", *py);
    printf(", &py = %p\n", &py);

    py++; /* Incremento */

    printf("py = %p", py);
    printf(", *py = %u", *py);
    printf(", &py = %p\n", &py);

    px = py + 5; /* Somar inteiros */

    printf("px = %p", px);
    printf(", *px = %u", *px);
    printf(", &px = %p\n", &px);

    printf("px-py= %u\n", (px-py));

    system("PAUSE");
    return 0;
}
```

## 7. Ponteiros e Arrays

- Vetores são variáveis de endereço contínuo na memória. Ou seja, enquanto uma variável do tipo **int** ocupa, por exemplo, um bloco de memória, um vetor de 5 posições do tipo **int** ocupa 5 blocos de memória.
- Para acessarmos cada item de um vetor usamos os índices que ficam entre colchetes ([]).
- Há duas formas de trabalhar com ponteiros para vetor.
  1. Usar índices no ponteiro como se ele fosse um vetor:

*ponteiro = vetor;*

*ponteiro[indice1] = qualquer valor para ser gravado nesse índice;*

*ponteiro[indiceN] = qualquer valor para ser gravado nesse índice;*

**Preste atenção no seguinte detalhe. Apesar de ser ponteiro, ao atribuirmos um valor ao índice não usamos o sinal de referência (\*).**

## 7. Ponteiros e Arrays

1. Usar índices no ponteiro como se ele fosse um vetor:

```

01.  #include <stdio.h>
02.  #include <stdlib.h>
03.
04.  int main (void) {
05.      int vetor [2];
06.      int *v; // ponteiro
07.      v = vetor;
08.      v[0] = 123;
09.      v[1] = 456;
10.      printf ("vetor[0] = %d\n", vetor[0]);
11.      printf ("vetor[1] = %d\n\n", vetor[1]);
12.      system ("pause");
13.  }
    
```

## 7. Ponteiros e Arrays

### 1. Outro exemplo:

```

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

int main() {

    char vetor[20] = "Teste de string";

    char *point;

    point = &vetor[6];

    printf("\n%c\n", *point);

    getch();
}

```

## 7. Ponteiros e Arrays

### 1. Outro exemplo (versão 2):

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

int main() {

    char vetor[20] = "Teste de string";

    char *point;

    point = vetor;

    point = point + 9;

    printf("\n%c\n", *point);

    getch();
}
```

## 7. Ponteiros e Arrays

O que está errado no código abaixo?

```
char nome[20], *pstr;
int val[10], *ptr;

pstr = nome;
ptr = val;

pstr = nome + 4;
ptr = val + 5;

pstr = nome++;
```

## 7. Ponteiros e Arrays

```

char nome[20], *pstr;
int val[10], *ptr;

```

```

pstr = nome;          /* Equivalente a pstr = &nome[0] */
ptr = val;            /* Equivalente a ptr = &val[0] */

```

```

pstr = nome + 4;      /* Equivalente a pstr = &nome[4] */
ptr = val + 5;        /* Equivalente a ptr = &val[5] */

```

```

pstr = nome++;        /* ATENCAO: INCORRETO !!! */
/* "nome" NÃO É UM PONTEIRO */

```

## 7. Ponteiros e Arrays

2. Usar o que chamamos de aritmética de ponteiros.

- Aritmética de ponteiros consiste em modificar o valor do ponteiro para ele indicar o próximo endereço de memória do vetor.

**Exemplificando, seria algo como:**

ponteiro = endereço do índice 0 do vetor;

$*(\text{ponteiro} + \text{indice1})$  = qualquer valor para ser gravado nesse índice;

$*(\text{ponteiro} + \text{indiceN})$  = qualquer valor para ser gravado nesse índice;

Perceba que aqui atribuímos um valor ao vetor usando um de referencia do ponteiro (\*).



## 7. Ponteiros e Arrays

Quando somamos um número ao ponteiro, o que estamos fazendo é, na verdade, somar o número necessário de bytes para o próximo endereço.

Ex.: Se tivermos um ponteiro para um vetor de inteiro, quando formos calcular o terceiro espaço faremos  $\text{*(ponteiro+3)}$ . Internamente será calculado o seguinte:  $\text{ponteiro} + 3 \times \text{o tamanho de int (4 bytes)}$ . Então,  $3 \times 4 = 12$  bytes.

```
ponteiro = 0x00001100;
ponteiro + 12 bytes;
novo ponteiro = 0x00001100C;
```

Apesar de ser uma conta simples, não precisamos nos preocupar com isso. Porque o próprio sistema cuida de executar este cálculo.

## 7. Ponteiros e Arrays

### Exemplo

```

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

int main(void)
{

    int mat[10] = {1,8,3,7,5,3,7,23,68,10};
    int *point, i;

    point = mat;
    for(i=0; i<10;i++){
        printf("%d\n", *(point+i));
    }

    getch();
}

```

## 7. Ponteiros e Arrays

### Outro Exemplo

```

01. #include <stdio.h>
02. #include <stdlib.h>
03.
04. #define MAX 10
05.
06. int main (void){
07.     int vetor [MAX], i, valor, *v;
08.     v = &vetor[0];
09.     printf ("Digite um valor para ser gravado no\n");
10.     printf ("indice\tEndereco de Memoria\n");
11.     for (i=0; i<MAX; i++) {
12.         printf("[%d]\t%p\t\t-> ", i, (v+i));
13.         scanf ("%d", &valor);
14.         getchar();
15.         *(v+i) = valor; //valor é gravado no endereço apontado pelo ponteiro
16.     }
17.     system ("cls");
18.     printf ("Os valores gravados no vetor foram:\n");
19.     for (i=0; i<MAX; i++) {
20.         printf("vetor[%d], ponteiro (%p) = %d\n", i, (v+i), vetor[i]);
21.     }
22.     system ("pause");
23. }

```

## 7. Ponteiros e Arrays

Outro Exemplo  
(Sem ponteiro):

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

int main(void)
{
    char mat[30];
    int i;

    printf("Entre com uma frase em letra maiuscula\n");
    scanf("%s", &mat);
    getchar();

    printf("Frase minuscula:");

    for(i=0; mat[i]; i++){
        printf("%c", tolower(mat[i]));
    }

    getch();
}
```

# Linguagem de Programação Estruturada – 2016.1

## 7. Ponteiros e Arrays

Outro Exemplo  
(Com ponteiro):

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

int main(void)
{
    char mat[30], *point;

    printf("Entre com uma frase em letra maiuscula\n");
    scanf("%s", &mat);
    getchar();

    printf("Frase minuscula:");

    point = mat;

    while(*point){
        printf("%c", tolower(*point++));
    }

    getch();
}
```

## 8. Ponteiros e Strings

Por definição, em C, uma string é um array de caracteres terminada com o caracter nul. Perceba, entretanto, que "nul" **\*\*não é\*** o mesmo que "NULL". 'nul' refere-se ao zero como definido pela seqüência de escape '\0'. Ela ocupa um byte de memória. NULL, por outro lado, é o nome da macro usada para iniciar ponteiros nulos.

```
my_string[0] = 'T';  
my_string[1] = 'e';  
my_string[2] = 'd';  
my_string[3] = '\0';
```

Como escrever o código acima pode demorar bastante, o C permite duas formas alternativas de chegar ao mesmo fim. Primeiro, pode-se escrever:

```
char my_string[40] = ('T', 'e', 'd', '\0');
```

Mas isto também usa mais digitação do que é conveniente. Assim, o C permite:

```
char my_string[40] = "Ted";
```

## 8. Ponteiros e Strings

É possível carregar o endereço do string em um ponteiro do tipo char.

```
void main(void)
{
    char *lista;

    lista = "Ola como vai?";
    printf("%s", lista);
}
```

## 8. Ponteiros e Strings

É possível ainda passar um ponteiro de string para uma função e na função o ponteiro \*(s+tam++) apontando (percorrendo) caracter a caracter da string.

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

int strtam(char *s);

int main(void)
{
    char *lista="1234567890";

    printf("O tamanho do string \"%s\" e %d caracteres.\n", lista, strtam(lista));
    printf("Acabou.");

    system("pause");
}

int strtam(char *s){
    int tam=0;

    while(*(s + tam++) != '\0');
    return tam-1;
}
```



## 9. Indireção Múltipla (Cadeia de Ponteiros)

**Variável (Valor)**

**Ponteiro1 → &Variável**

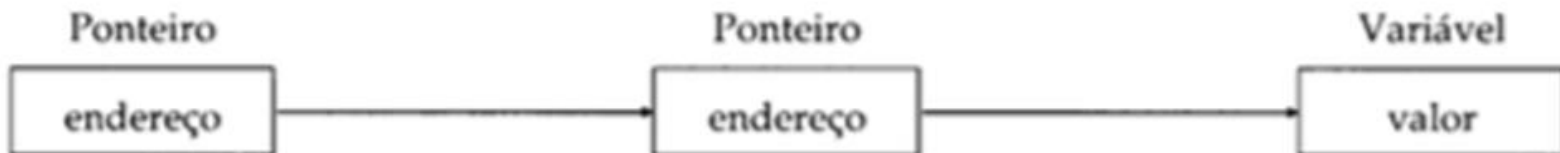
**Ponteiro2 → &Ponteiro1**

## 9. Indireção Múltipla (Cadeia de Ponteiros)

É um ponteiro de ponteiros, ou seja, é quando um ponteiro aponta para outro ponteiro que tem o valor final;



**Indireção Simples**



**Indireção Múltipla**

## 9. Indireção Múltipla (Cadeia de Ponteiros)

Exemplo:

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

int main(void)
{
    int var;
    int *point1;
    int **point2;

    var = 15;

    point1 = &var;
    point2 = &point1;

    printf("\n%d\n", **point2);

    system("pause");
}
```

## 10. Alocação Dinâmica de memória

- As declarações abaixo alocam espaço de memória para diversas variáveis. A alocação é estática (nada a ver com a palavra-chave *static*), ou seja, acontece antes que o programa comece a ser executado:

```
char c;  
int i;  
int v[10];
```

- A linguagem C oferece meios de requisitarmos espaços de memória em tempo de execução.
  - (1) uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado.
  - (2) uso de variáveis locais. Neste caso, o espaço existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Assim, a função que chama não pode fazer referência ao espaço local da função chamada.

## 10. Alocação Dinâmica de memória

- Para que a quantidade de memória a alocar só se torna conhecida durante a *execução* do programa é preciso recorrer à alocação *dinâmica* de memória. A alocação dinâmica é gerenciada pelas funções: ***malloc***, ***realloc***, ***calloc*** e ***free***, que estão na biblioteca ***stdlib***.

### 10. 1 A função malloc()

A função ***malloc*** (o nome é uma abreviatura de *memory allocation*) aloca um bloco de bytes consecutivos na memória RAM (= *random access memory*) do computador e devolve o endereço desse bloco.

No seguinte fragmento de código, ***malloc*** aloca 1 byte:

```
char *ptr;
ptr = malloc (1);
scanf ("%c", ptr);
```

## 10. Alocação Dinâmica de memória

### 10.1 A função malloc()

No seguinte fragmento de código, **malloc** aloca 1 byte:

```
char *ptr;
ptr = malloc (1);
scanf ("%c", ptr);
```

- O endereço devolvido por **malloc** é do tipo genérico void \*.
- No exemplo acima, o endereço é atribuído ao ponteiro **ptr** que é do tipo ponteiro-para-char. (A transformação do ponteiro genérico em ponteiro-para-char é automática; não é necessário escrever **ptr = (char \*) malloc (1);**.)

## 10. Alocação Dinâmica de memória

### 10. 1 A função malloc()

- Retorna um ponteiro contendo o endereço do bloco alocado, ou seja, devolve um ponteiro para o primeiro byte da região de memória alocada. Entretanto, se não há memória disponível devolve o valor nulo;
- Sintaxe:

```
void *malloc(size_t numero_de_bytes);
```

## 10. Alocação Dinâmica de memória

### 10. 1 A função malloc()

- Como estudamos bem, os ponteiros precisam saber para que tipo de variável vão apontar, pois (dentre outras coisas), podemos fazer operações matemática com ponteiros.

**Por exemplo: ptr++;**

- Se esse ponteiro apontar para um caractere, ao incrementarmos, ele pulará uma posição de endereço de memória.
- Já se apontar para um inteiro, ele pulará **int** posições de endereço, para apontar para o próximo inteiro de um vetor, por exemplo.



## 10. Alocação Dinâmica de memória

### 10. 1 A função malloc()

- Como a função malloc() serve para declarar qualquer tipo de dado, seja **int**, **float**, **double** ou uma **struct** criada por você, nos exemplos anteriores foi mostrado como void.
- Se quisermos alocar um bloco de endereços para inteiros, ao invés do void\* colocamos:

```
(int *) malloc(size_t bytes);
```

## 10. Alocação Dinâmica de memória

### 10.1 A função malloc()

Exemplo, se quiséssemos alocar 20 caracteres para conter uma string, devemos fazer:

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

int main(void)
{
    char *nome;
    nome = (char *) malloc(21);

    printf("Digite seu nome: ");
    gets(nome);

    printf("%s\n", nome);

    system("pause");
}
```

Poderíamos criar o ponteiro e logo na declaração fazer ele receber o endereço de um bloco alocado de memória:

```
char *nome = (char *) malloc(21);
```

## 10. Alocação Dinâmica de memória

### 10. 1 A função malloc()

#### Observação:

Há máquina que um inteiro ocupa 4 bytes, em outras ocupam 8 bytes.

Qual a maneira certa de alocar?

utilizar a função sizeof():

```
char *nome = (char *) malloc(21*sizeof(char));
```

## 10. Alocação Dinâmica de memória

### 10. 1 A função malloc()

**Exemplo:**

```
typedef struct {  
    int dia, mes, ano;  
} data;  
data *d;  
d = malloc (sizeof (data));  
d->dia = 31; d->mes = 12; d->ano = 2014;
```

## 10. Alocação Dinâmica de memória

### 10.1 A função malloc()

#### Resumindo:

- Devolve um ponteiro para o primeiro byte da região de memória alocada. Entretanto, se não há memória disponível devolve o valor nulo;
- **Sintaxe:** `malloc(size_t numero_de_bytes);`

```
char *p;
p = malloc(1000); /* obtém 1000 bytes */
```

```
if(!(p=malloc(100)) {
    printf("sem memória.\n");
    exit(1);
}
```

## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

A função **free** desaloca a porção de memória alocada por **malloc**. A instrução **free (ptr)** avisa ao sistema que o bloco de bytes apontado por **ptr** está livre e disponível para reciclagem. A próxima chamada de **malloc** poderá tomar posse desses bytes.

Convém não deixar ponteiros "soltos" (= *dangling pointers*) no seu programa, pois isso pode ser explorado por hackers para atacar o seu computador. Portanto, depois de cada **free (ptr)**, atribua **NULL** a **ptr**:

```
free (ptr);  
ptr = NULL;
```

## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

#### Exemplo

- Supondo que criamos uma função chamada aloca(), que não recebe nem retorna nada.
- Dentro dela criamos um ponteiro para um inteiro e em seguida alocamos 100 bytes, através da **malloc()**.
- Obviamente, quando a função termina, esse ponteiro deixa de existir, mas a memória reservada continua lá, e inacessível.
- Na nossa main() fazemos infinitas chamadas da função aloca(), que aos poucos vai reservando de 100 em 100 bytes a memória de seu computador. E aos poucos você vai notar sua máquina ficando lenta, pois vai haver cada vez menos memória, vai ficando cada vez mais 'difícil' de achar espaços para alocar até...sua máquina travar completamente.

## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

#### Exemplo

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

void aloca()
{
    int *ptr;
    ptr = (int *) malloc(100);
}

int main(void)
{
    while(1)
        aloca();

    system("pause");
}
```



## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

- Basicamente, devemos liberar a memória sempre que não formos mais usar o que foi alocado.
- Para 'consertar' o exemplo de código passado, simplesmente precisamos acrescentar **free(ptr)** ao final da função.

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

void aloca()
{
    int *ptr;
    ptr = (int *) malloc(100);

    free(ptr);
}

int main(void)
{
    while(1)
        aloca();

    system("pause");
}
```

## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

- Exemplo com **array**

```
int *v;
int n, i;
scanf ("%d", &n);
v = malloc (n * sizeof (int));
for (i = 0; i < n; ++i)
    scanf ("%d", &v[i]);
...
free (v);
```

## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

- Outro Exemplo com **array**

```
main()
{
    // alocando um vetor com 3 inteiros
    int *v = (int*) malloc( 3 * sizeof(int) );
    if (v)
    {
        v[0] = 10;
        v[1] = 20;
        v[2] = 30;
        printf("%d %d %d\n", v[0], v[1], v[2]);
        free(v);
    }
}
```

## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

#### Segurança

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

int main(void)
{
    char *senha;

    senha = (char *) malloc(21*sizeof(char));
    printf("Digite sua senha [ate 20 caracteres]: ");
    scanf("%s", senha);

    printf("Senha: %s\n", senha);
    printf("Endereço antes da free(): %d\n", &senha);

    free(senha);

    printf("Endereço depois da free(): %d\n", &senha);

    system("pause");
}
```

## 10. Alocação Dinâmica de memória

### 10. 2. A função free()

#### Resumindo:

- Devolve ao sistema memória previamente alocada;
- **Sintaxe:** free(void \*p);
- Exemplo:
 

```
int *p;
p=malloc(1000);
free(p);
```

### 10. Alocação Dinâmica de memória

#### 10. 2. A função `realloc()` - Redimensionamento

Às vezes é necessário alterar, durante a execução do programa, o tamanho do bloco de bytes alocado por `malloc`. Isso acontece, por exemplo, durante a leitura de um arquivo que se revela maior que o esperado. Nesse caso, podemos recorrer à função `realloc` para redimensionar o bloco de bytes.

## 10. Alocação Dinâmica de memória

### 10. 2. A função realloc() - Redimensionamento

A função realloc recebe o endereço de um bloco previamente alocado por malloc (ou realloc) e o número de bytes que o bloco redimensionado deve ter. A função aloca o novo bloco, transfere para ele o conteúdo do bloco original, e devolve o endereço do novo bloco.

```
int *v;
v = malloc (1000 * sizeof (int));
for (i = 0; i < 990; i++)
    scanf ("%d", &v[i]);
v = realloc (v, 2000 * sizeof (int));
for (i = 990; i < 2000; i++)
    scanf ("%d", &v[i]);
```

## 11. Ponteiro para Estruturas

- Um *struct* consiste em vários dados agrupados em apenas um. Para acessarmos cada um desses dados, usamos um ponto (.) para indicar que o nome seguinte é o nome do membro.
- Um ponteiro guarda o endereço de memória que pode ser acessado diretamente.
- Para acessarmos um membro de uma estrutura de dados usando um ponteiro, devemos utilizar a "seta" consiste de um sinal de menos e um maior (->).



## 11. Ponteiro para Estruturas

### Declaração

```
struct addr *addr_pointer;
```

### Exemplo:

```
struct bal {  
    float balance;  
    char name[80];  
} person;
```

```
struct bal *p; /* declara um ponteiro para estrutura */  
p = &person;
```

## 11. Ponteiro para Estruturas

### Declaração

```
struct addr *addr_pointer;
```

### Exemplo:

```
struct bal {
    float balance;
    char name[80];
} person;
```

```
struct bal *p; /* declara um ponteiro para estrutura */
p = &person;
```

Para acessar os elementos internos de uma estrutura através de ponteiros usa-se:

`p->balance`

## 11. Ponteiro para Estruturas

### Outro Exemplo:

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

struct data{
    short dia;
    short mes;
    int ano;
};

int main (void){
    struct data Data; //variável data do tipo struct data
    struct data *hoje; //ponteiro hoje para um tipo struct data
    hoje = &Data; //hoje aponta para o endereço de data

    //dados sendo inseridos na variável data
    hoje->dia = 25;
    hoje->mes = 4;
    hoje->ano = 2016;

    //mostrando o que está gravado no endereço contido em hoje

    printf("Data registrada:");
    printf("%d/%d/%d", hoje->dia,hoje->mes,hoje->ano);
    printf("\n\n");
    system ("pause");
}
```

## 11. Ponteiro para Estruturas

Outro Exemplo:

```
typedef struct
{
    char nome[100];
    int idade;
} pessoa;

main()
{
    // alocando uma estrutura
    pessoa *p = (pessoa*) malloc(sizeof(pessoa));
    if (p)
    {
        p->idade = 3;
        printf("%d\n", p->idade);
        free(p);
    }
}
```

## 12. Ponteiro para Funções

O C permite que acessemos variáveis e funções através de ponteiros. Um ponteiro para uma função tem a seguinte declaração:

```
tipo_de_retorno *nome_do_ponteiro();
```

ou

```
tipo_de_retorno *nome_do_ponteiro(declaração de parâmetros);
```

ou

```
tipo_de_retorno nome_da_funcao(*ponteiro);
```

## 12. Ponteiro para Funções

Exemplo1:

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

int strtam(char *s);

int main(void)
{
    char *lista="1234567890";

    printf("O tamanho do string \"%s\" e %d caracteres.\n", lista, strtam(lista));
    printf("Acabou.");

    system("pause");
}

int strtam(char *s){
    int tam=0;

    while(*(s + tam++) != '\0');
    return tam-1;
}
```

## 12. Ponteiro para Funções

Exemplo2:

**3ª questão da lista 2** - Escreva uma função que receba um vetor de inteiro  $v[0..n-1]$  (ponteiro) e os endereços de duas variáveis inteiras, digamos min e max (ponteiros), e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva uma função main que solicite ao usuário o número de elementos do vetor e o valor de cada elemento. Em seguida, use a função **mm** para informar o maior e o menor valor do vetor.

## 12. Ponteiro para Funções

Exemplo2:

```
#include <stdio.h>
#include<stdlib.h>
void mm(int *v, int n, int *min, int *max) {
    int i;
    *min = v[0];
    *max = v[0];
    for (i = 1; i < n; i++) {
        if (v[i] > *max) {
            *max = v[i];
        }
        else if (v[i] < *min) {
            *min = v[i];
        }
    }
}

int main() {
    int n, i, *vet, minimo, maximo;
    printf("Quantos numeros voce deseja digitar? ");
    scanf("%d", &n);
    vet = (int*)malloc(n * sizeof(int));
    for (i = 0; i < n; i++) {
        printf("Digite o numero de indice %d: ", i);
        scanf("%d", &vet[i]);
    }

    mm(vet, n, &minimo, &maximo);
    printf("Minimo: %d. Maximo: %d.\n", minimo, maximo);

    system ("pause");
}
```



## 12. Ponteiro para Funções

**6ª questão da lista 3** - Crie uma função que receba uma string como parâmetro (de tamanho desconhecido) e retorne uma cópia da mesma. O espaço de memória para a nova variável que irá receber a String deve ser alocado antes da copia. A assinatura da função deve ser:

```
char *strcpy(char *str);
```

## 12. Ponteiro para Funções

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

char *strcpy(char *str) {
    int n, i;
    char *nova;
    //Primeiro vamos contar quantos caracteres a string tem:
    for (n = 0; str[n] != '\0'; n++)
    //Agora vamos copiar alocar memoria para a nova string:
    nova = malloc(n * sizeof(char));
    //Com a memoria alocada, podemos copiar:
    for (i = 0; i <= n; i++) {
        nova[i] = str[i];
    }
    return nova;
}

int main(void)
{
    char *strOriginal="GilJader";

    printf("%s",strcpy(strOriginal));

    system("pause");
}
```

## 7. REFERÊNCIAS

- Márcio Alexandre Marques. Algoritmos - Lógica para Desenvolvimento de Programação de Computadores. 1ª Ed. Editora Érica, 2010.
- Sandra Rita. TREINAMENTO EM LOGICA DE PROGRAMAÇÃO, Digerati Books, 1 ed. 2009.
- SIMÃO, DANIEL HAIASHIDA; REIS, WELLINGTON JOSÉ DOS. LOGICA DE PROGRAMAÇÃO. São Paulo : EDITORA VIENA, 2015. 176p.
- Souza, Marco Antonio Furlan de *et. all*, Algoritmos e Lógica de Programação. 2 ed. São Paulo : Nobel, 2011.