

Ponteiros

A series of horizontal lines in teal and light blue colors, some solid and some dashed, extending across the bottom of the slide.

Definições

Variáveis : endereçam uma posição de memória que contém um determinado valor dependendo do seu tipo (char, int, float, double, ...)

```
void main() {  
    long a=5;  
    char ch='x';  
}
```

	endereço	valor	
a ➡	0x0100	0x00	} 5
	0x0101	0x00	
	0x0102	0x00	
	0x0103	0x05	
ch ➡	0x0104	0x78	} 'x'

Definições

Ponteiros: são variáveis cujo conteúdo é um endereço de memória.

- Assim, um ponteiro endereça uma posição de memória que contém um endereço para outra posição de memória.

```
void main() {  
    long a=5;  
    char ch='x';  
    long *aPtr = &a;  
}
```

	endereço	valor	
a →	0x0100	0x00	5
	0x0101	0x00	
	0x0102	0x00	
	0x0103	0x05	
ch →	0x0104	0x78	'c'
aPtr →	0x0105	0x00	0x00000100
	0x0106	0x00	
	0x0107	0x01	
	0x0108	0x00	

Declaração de Ponteiros

- Para declararmos um ponteiro, basta utilizar o operador *(asterisco) antes do nome da variável.
- Exemplo:

```
int *p;
```

- Ponteiros são tipados, ou seja, devem ter seu tipo declarado e somente podem apontar para variáveis do mesmo tipo.

Operadores para Ponteiros

Para trabalharmos com ponteiros, C disponibiliza os seguintes operadores:

& → Fornece o endereço de memória onde está armazenada uma variável. Lê-se “*o endereço de*”.

* → Valor armazenado na variável referenciada por um ponteiro. Lê-se “*o valor apontado por*”.

Operadores para Ponteiros

```
int main(void) {  
    long a=5;  
    char ch='x';  
    long *aPrt = &a;  
    printf("\n%lu",*aPrt);  
    printf("\n%p",aPrt);  
    printf("\n%p",&aPrt);  
}
```

	endereço	valor	
a →	0x0100	0x00	5
	0x0101	0x00	
	0x0102	0x00	
	0x0103	0x05	
ch →	0x0104	0x78	'c'
aPrt →	0x0105	0x00	0x00000100
	0x0106	0x00	
	0x0107	0x01	
	0x0108	0x00	

- O que será impresso na tela?

```
5  
0x0100  
0x0105
```

Exemplo 1

```
#include <stdio.h>
int main (void)
{
    int num,valor;
    int *p;
    num=55;
    p=&num; /* Pega o endereco de num */
    valor=*p; /* Valor é igualado a num de uma maneira indireta */
    printf ("%d\n",valor);
    printf ("Endereco para onde o ponteiro aponta: %p\n",p);
    printf ("Valor da variavel apontada: %d\n",*p);
}
```

55

Endereco para onde o ponteiro aponta: 0022FF14

Valor da variavel apontada: 55

Exemplo 2

```
#include <stdio.h>
main ()
{
    int num,*p;
    num=55;
    p=&num; /* Pega o endereco de num */
    printf ("Valor inicial: %d\n",num);
    *p=100; /* Muda o valor de num de uma maneira indireta */
    printf ("\nValor final: %d\n",num);
}
```

Valor inicial: 55
Valor final: 100

Operadores para Ponteiros

- Igualando ponteiros:

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

```
p1=p2;
```

- Repare que estamos fazendo com que p1 aponte para o mesmo lugar que p2.

- Assim a variável apontada por p1 tem o mesmo conteúdo da variável apontada por p2

```
*p1=*p2;
```

Exemplo 3

```
int main (void)
{
    int num,*p1, *p2;
    num=55;
    p1=&num; //Pega o endereco de num
    p2=p1; //p2 passa a apontar para o mesmo endereço apontado por p1
    printf ("Conteudo de p1: %p\n",p1);
    printf ("Valor apontado por p1: %d\n",*p1);
    printf ("Conteudo de p2: %p\n",p2);
    printf ("Valor apontado por p2: %d\n",*p2);
}
```

Exemplo 4

```
int main (void)
{
    int num1, num2,*p1, *p2;
    num1=55;
    p1=&num1; // Pega o endereco de num
    num2= 100;
    p2=&num2;
    *p2=*p1; // p2 recebe o valor apontado por p1
    printf ("Conteudo de p1: %p\n",p1);
    printf ("Valor apontado por p1: %d\n",*p1);
    printf ("Conteudo de p2: %p\n",p2);
    printf ("Valor apontado por p2: %d\n",*p2);
}
```

Exemplo 5

Passagem de Parâmetros por Referência

```
void troca (int *x, int *y) // troca os valores entre os dois parâmetros
{
    int temp;
    temp=*x; // salva o valor do endereço x
    *x=*y; // põe y em x
    *y=temp; // põe x em y
}
```

```
int main(void)
{
    int i, j;
    i=10;
    j=20;
    troca (&i, &j); //passa os endereços de i e j
}
```

Operadores para Ponteiros

- Ao acessar um ponteiro devemos ter cuidado: um ponteiro com um endereço de memória inválido ou nulo, ao ser referenciado, irá causar um erro de “Falha de segmentação” (segmentation fault), finalizando forçadamente a execução de seu programa.
- Coisas como essa precisam ser evitadas:

```
double* pDouble;  
*pDouble = 2.5;
```

```
long* pLong;  
int inteiro = *pLong;
```

```
char* string = NULL;  
puts(string);
```

Exercício 1

- Faça uma função denominada “pesquisa” que recebe dois parâmetros por valor: (1) um vetor de inteiros e (2) um valor que se deseja pesquisar a existência no vetor.
- A função deve retornar:
 - 1 caso encontre o valor pesquisado e
 - 0 caso contrário.
- Há ainda um terceiro parâmetro denominado pos. Este parâmetro deve retornar por referência a posição onde o valor foi encontrado no vetor.
- Sabemos que uma função tem apenas um valor de retorno, mas nesse caso a função retornará também (via referência) a posição onde foi encontrado o valor, estabelecendo uma relação entre a variável global e o parâmetro.

Exercício 1

```
#include<stdio.h>
# define tam 10
```

```
int pesquisa(int numeros[tam], int pesq, int *pos){
    *pos=0;
    while(*pos<tam && numeros[*pos]!=pesq)
    {
        *pos=*pos+1;
    }
    if(*pos<tam)
        return(1);
    else return(0);
}
```

Exercício 1

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

```
# define tam 10
```

```
int pesquisa(int numeros[tam], int pesq, int *pos){
```

```
    *pos=0;
```

```
    while(*pos<tam && numeros[*pos]!=pesq)
```

```
        *pos=*pos+1;
```

```
    if(*pos<tam)
```

```
        return(1);
```

```
    else return(0);
```

```
}
```

```
int vetor[tam], num, p, i;
```

```
int main(void) {
```

```
    for (i=0;i<10;i++) {
```

```
        printf("Digite o %iº valor do vetor...: ", i);
```

```
        scanf("%i", &vetor[i]);
```

```
    }
```

```
    printf("\n\nDigite um valor para pesquisa: ");
```

```
    scanf("%i",&num);
```

```
    if (pesquisa(vetor,num,&p)==1)
```

```
        printf("Encontrado na posicao %i",p);
```

```
    else printf("Nao encontrado!");
```

```
}
```


Operadores para Ponteiros

Incremento/Decremento:

- Apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta:

```
long *aPtr, a=5;  
aPtr=&a;  
aPtr++;
```

	endereço	valor	
a →	0x0100	0x00	5
	0x0101	0x00	
	0x0102	0x00	
	0x0103	0x05	
ch →	0x0104	0x78	'c'
aPtr →	0x0105	0x00	0x00000100
	0x0106	0x00	
	0x0107	0x01	
	0x0108	0x00	

Operadores para Ponteiros

Qual será o valor endereçado por `aPtr++` ??

- Se `aPtr` é `long`, como o `long` ocupa 4 bytes, `aPtr` irá apontar para o endereço `0x0104`
- Este é o principal motivo que nos obriga a definir um tipo para um ponteiro!!!

	endereço	valor	
a →	0x0100	0x00	5
	0x0101	0x00	
	0x0102	0x00	
	0x0103	0x05	
ch →	0x0104	0x78	'c'
aPtr →	0x0105	0x00	0x00000100
	0x0106	0x00	
	0x0107	0x01	
	0x0108	0x00	

Operadores para Ponteiros

Tipo	Num de bits	Intervalo	
		Inicio	Fim
char	8	-128	127
unsigned char	8	0	255
signed char	8	-128	127
int	16	-32.768	32.767
unsigned int	16	0	65.535
signed int	16	-32.768	32.767
short int	16	-32.768	32.767
unsigned short int	16	0	65.535
signed short int	16	-32.768	32.767
long int	32	-2.147.483.648	2.147.483.647
signed long int	32	-2.147.483.648	2.147.483.647
unsigned long int	32	0	4.294.967.295
float	32	3,4E-38	3,4E+38
double	64	1,7E-308	1,7E+308
long double	80	3,4E-4932	3,4E+4932

Exemplo 6

```
int main(void)
{
    long num;
    long *p;
    num=55;
    p=&num;
    printf ("Conteudo de p: %p\n",p);
    printf ("Valor apontado por p: %d\n",*p);
    printf ("Conteudo de p incrementado: %p\n",++p);
    printf ("Valor apontado por p incrementado: %d\n",*p);
}
```

Exemplo 7

```
int main(void)
{
    long num;
    long *p;
    num=55;
    p=&num;
    printf ("Conteudo de p: %p\n",p);
    printf ("Valor apontado por p: %d\n",*p);
    printf ("Valor apontado por p incrementado: %d\n",++(*p));
    printf ("Conteudo de p: %p\n",p);
}
```

Vetores como ponteiros

- A linguagem C enxerga vetores como ponteiros;
- Quando declaramos um vetor, C aloca memória para todas as posições necessárias conforme seu tipo:
 - `int vet[10];`
- O nome do vetor pode ser atribuído a um ponteiro. Neste caso o ponteiro irá endereçar a posição do vetor:
 - `int *p; p=vet; ou`
 - `int *p; p=&vet[0];`

Exemplo 9

```
int main(void)
{
    int vet [4];
    int *p;
    p=vet;
    for (int cont=0;cont<4;cont++) //inicializa o vetor todo com zero
    {
        *p=0;
        p++;
    }

    for (int i=0;i<4;i++)
        printf ("%d\n",vet[i]);
}
```

Exemplo 10

```
int main(void)
{
    float mat [4][4];
    float *p;
    int cont;
    p=mat[0];
    for (cont=0;cont<16;cont++)
    {
        *p=0.0;
        p++;
    }
    for (int i=0;i<4; i++)
        for (int k=0;k<4;k++)
            printf("\nLinha %d Coluna %d = %.1f", i, k, mat[1][k]);
}
```


Vetores como ponteiros

- **Importante**: um ponteiro é uma variável, mas o nome de um vetor não é uma variável;
- Isto significa que não se consegue alterar o endereço que é apontado pelo nome do vetor;
- Diz-se que um vetor é um ponteiro constante!
- Condições inválidas:
`int vet[10], *p;`
`vet++;`
`vet = p;`

Ponteiros como vetores

- Quando um ponteiro está endereçando um vetor, podemos utilizar a indexação também com os ponteiros:

- Exemplo:

```
int vet [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int *p;  
p=vet;  
printf("O terceiro elemento do vetor e: %d",p[2]);
```

- Neste caso $p[2]$ equivale a $*(p+2)$

Porque inicializar ponteiros?

- Observe o código:

```
int main(void) //Errado - Nao Execute
{
    int x,*p;
    x=13;
    *p=x; //posição de memória de p é indefinida!
}
```

- A não inicialização de ponteiros pode fazer com que ele esteja alocando um espaço de memória utilizado, por exemplo, pelo sistema operacional.

Porque inicializar ponteiros?

- No caso de vetores, é necessário sempre alocar a memória necessária para compor as posições do vetor.
- O exemplo abaixo apresenta um programa que compila, porém poderá ocasionar sérios problemas na execução. Como por exemplo utilizar um espaço de memória alocado para outra aplicação.

```
int main(void) {  
    char *pc; char str[] = "Uma string";  
    strcpy(pc, str); // pc indefinido  
}
```

Alocação dinâmica de memória

- Durante a execução de um programa é possível alocar uma certa quantidade de memória para conter dados do programa;
- A função `malloc (n)` aloca dinamicamente n bytes e devolve um ponteiro para o início da memória alocada;
- A função `free(p)` libera a região de memória apontada por `p`. O tamanho liberado está implícito, isto é, é igual ao que foi alocado anteriormente por `malloc`.

Alocação dinâmica de memória

- Os comandos abaixo alocam dinamicamente um inteiro e depois o liberam:

```
#include <stdlib.h>
int *pi;
pi = (int *) malloc (sizeof(int));
...
free(pi);
```

- A função malloc não tem um tipo específico. Assim, (int *) converte seu valor em ponteiro para inteiro. Como não sabemos necessariamente o comprimento de um inteiro (2 ou 4 bytes dependendo do compilador), usamos como parâmetro a função sizeof(int).

Alocação dinâmica de vetores

```
#include <stdlib.h>
int main(void) {
    int *v, i, n;
    scanf("%d", &n);
    //aloca n elementos para v
    v = (int *) malloc(n*sizeof(int));
    // zera o vetor v com n elementos
    for (i = 0; i < n; i++) v[i] = 0;
    ...
    // libera os n elementos de v
    free(v);
}
```

Exercício 2

- Faça uma função que receba um valor inteiro como referência e retorne o resto da divisão deste número por 10. Altere também o valor da variável passada por referência, dividindo-a por 10.
- Faça um programa que imprima invertido os nomes dos algarismos de um número inteiro. (Use a sua função!)

Ex: 234 saída: quatro três dois

Exercício 3

- Faça um programa que tenha duas matrizes de ordem 3 e dois ponteiros que são inicializados com os endereços do começo das matrizes, use o ponteiro para preencher as matrizes, uma começando pelas linhas e a outra pelas colunas, depois calcule o resultado da soma de matriz em uma terceira matriz usando um terceiro ponteiro para ela (neste último caso, pode-se começar pelas linhas ou colunas, fica a seu critério).