



Ponteiros

Definição

- Variável

- É um espaço reservado de memória usado para guardar um valor que pode ser modificado pelo programa

- Ponteiro

- É um espaço reservado de memória usado para guardar o endereço de memória de uma outra variável

Declaração

- Um ponteiro também possui um tipo.
 - `tipo_do_ponteiro *nome_do_ponteiro;`
- O asterisco (*) que informa ao compilador que aquela variável vai guardar um endereço para o tipo especificado.
 - Variável: **int x;**
 - Ponteiro: **int *x;**

Declaração

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      //Declara um ponteiro para int
05      int *p;
06      //Declara um ponteiro para float
07      float *x;
08      //Declara um ponteiro para char
09      char *y;
10      //Declara uma variável do tipo int e um ponteiro para int
11      int soma, *p2,;
12      system("pause");
13      return 0;
14  }
```

Inicialização

- **Cuidado:** Ponteiros não inicializados apontam para um lugar indefinido.
 - Ex: `int *p;`

Memória		
#	var	conteúdo
119		
120	int *p	????
121		

Inicialização

- Um ponteiro pode ter o valor especial NULL que é o endereço de nenhum lugar.
 - Ex: `int *p = NULL;`

Memória		
#	var	conteúdo
119		
120	int *p	NULL
121		

nenhum lugar
na memória

Inicialização

- Os ponteiros devem ser inicializados antes de serem usados.
 - Devemos apontá-lo para um lugar conhecido;
 - Podemos apontá-lo para uma variável que já exista no programa.

Memória		
#	var	conteúdo
119		
120	int *p	#122
121		
122	int count	10
123		



Inicialização

- O ponteiro armazena o endereço da variável para onde ele aponta.
 - Para saber o endereço de memória de uma variável do nosso programa, usamos o operador **&**.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      //Declara uma variável int contendo o valor 10
05      int count = 10;
06      //Declara um ponteiro para int
07      int *p;
08      //Atribui ao ponteiro o endereço da variável int
09      p = &count;
10
11      system("pause");
12      return 0;
13  }
```


Utilização

- Como saber o valor guardado em uma variável através de um ponteiro?
 - Use o operador asterisco “*” na frente do nome do ponteiro

Utilização

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      //Declara uma variável int contendo o valor 10
05      int count = 10;
06      //Declara um ponteiro para int
07      int *p;
08      //Atribui ao ponteiro o endereço da variável int
09      p = &count;
10      printf("Conteudo apontado por p: %d \n",*p);
11      //Atribui um novo valor à posição de memória
    apontada por p
12      *p = 12;
13      printf("Conteudo apontado por p: %d \n",*p);
14      printf("Conteudo de count: %d \n",count);
15
16      system("pause");
17      return 0;
18  }
```

Saída	Conteudo apontado por p: 10 Conteudo apontado por p: 12 Conteudo de count: 12
-------	---

Utilização

- `*p`
 - conteúdo da posição de memória apontado por **p**;
- `&count`
 - o endereço na memória onde está armazenada a variável **count**.

Operações com ponteiros

- Atribuição

- p1 aponta para o mesmo lugar que p2;

p1 = p2;

- a variável apontada por p1 recebe o mesmo conteúdo da variável apontada por p2;

***p1 = *p2;**

Operações com ponteiros

- Duas operações aritméticas possíveis: adição e subtração (apenas inteiros)
 - `p++`; soma +1 no endereço armazenado no ponteiro.
 - `p--`; subtrai 1 no endereço armazenado no ponteiro.
 - Considere um ponteiro para inteiro, **int** *. O tipo **int** ocupa um espaço de 4 bytes na memória.
 - Cada incremento/decremento adiciona/subtrai 4 bytes

Operações com ponteiros

- Operações Ilegais com ponteiros
 - Dividir ou multiplicar ponteiros;
 - Somar o endereço de dois ponteiros;
 - Não se pode adicionar ou subtrair float ou double de ponteiros.

Operações com ponteiros

- Já sobre seu conteúdo apontado, valem todas as operações
 - `(*p)++`; incrementar o conteúdo da variável apontada pelo ponteiro `p`;
 - `*p = (*p) * 15`; multiplica o conteúdo da variável apontada pelo ponteiro `p` por 15;

Operações com ponteiros

- Operações relacionais
 - `==` e `!=` para saber se dois ponteiros são iguais ou diferentes.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int *p, *p1, x, y;
05      p = &x;
06      p1 = &y;
07      if(p == p1)
08          printf("Ponteiros iguais\n");
09      else
10          printf("Ponteiros diferentes\n");
11      system("pause");
12      return 0;
13  }
```


Operações com ponteiros

- Operações relacionais
 - $>$, $<$, $>=$ e $<=$ para saber qual ponteiro aponta para uma posição mais alta na memória.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int *p, *p1, x, y;
05      p = &x;
06      p1 = &y;
07      if(p > p1)
08          printf("O ponteiro p aponta para uma posicao a
frente de p1\n");
09      else
10          printf("O ponteiro p NAO aponta para uma posicao
a frente de p1\n");
11      system("pause");
12      return 0;
13  }
```

Ponteiros Genéricos

- Normalmente, um ponteiro aponta para um tipo específico de dado.
 - Um ponteiro genérico é um ponteiro que pode apontar para qualquer tipo de dado.
- Declaração
 - **`void *nome_ponteiro;`**

Ponteiros Genéricos

- Para acessar o conteúdo de um ponteiro genérico é preciso antes convertê-lo para o tipo de ponteiro com o qual se deseja trabalhar via ***type cast***

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      void *pp;
05      int p2 = 10;
06      // ponteiro genérico recebe o endereço de um
07      inteiro
08      pp = &p2;
09      //enta acessar o conteúdo do ponteiro genérico
10      printf("Conteudo: %d\n",*pp); //ERRO
11      //converte o ponteiro genérico pp para (int *)
12      antes de acessar seu conteúdo.
13      printf("Conteudo: %d\n",*(int*)pp); //CORRETO
14      system("pause");
15      return 0;
16  }
```

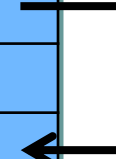
Ponteiros e Arrays

- Arrays são agrupamentos de dados do mesmo tipo na memória.
- Quando declaramos um array, informamos ao computador para reservar uma certa quantidade de memória a fim de armazenar os elementos do array de forma sequencial.
- Como resultado dessa operação, o computador nos devolve um ponteiro que aponta para o começo dessa sequência de bytes na memória.

Ponteiros e Arrays

- O nome do array (sem índice) é apenas um ponteiro que aponta para o primeiro elemento do array.
 - `int vet[5], *p;`
 - `p = vet;`

Memória		
#	var	conteúdo
123	int *p	#125
124		
125	int vet[5]	1
126		2
127		3
128		4
129		5
.		
.		
.		



Ponteiros e Arrays

- Nesse exemplo

```
int vet[5], *p;
```

```
p = vet;
```

- Temos que:

- ***p** é equivalente a **vet[0]**;
- **vet[indice]** é equivalente a ***(p+indice)**;
- **vet** é equivalente a **&vet[0]**;
- **&vet[indice]** é equivalente a **(vet + indice)**;

Ponteiros e Arrays

Exemplo: acessando arrays utilizando ponteiros

	Usando array	Usando ponteiro
01	#include <stdio.h>	#include <stdio.h>
02	#include <stdlib.h>	#include <stdlib.h>
03	int main(){	int main(){
04	int vet[5]= {1,2,3,4,5};	int vet[5]= {1,2,3,4,5};
05	int *p = vet;	int *p = vet;
06	int i;	int i;
07	for (i = 0;i < 5;i++)	for (i = 0;i < 5;i++)
08	printf("%d\n",p[i]);	printf("%d\n",*(p+i));
09	system("pause");	system("pause");
10	return 0;	return 0;
11	}	}

Ponteiros e Arrays

O valor entre colchetes é o deslocamento a partir da posição inicial. Nesse caso, **p[2]** equivale a ***(p+2)**.

```
#include <stdio.h>
#include <stdlib.h>
int main (){
    int vet[5] = {1,2,3,4,5};
    int *p;
    p = vet;
    printf ("Terceiro elemento: %d ou %d",p[2],*(p+2));
    system("pause");
    return 0;
}
```

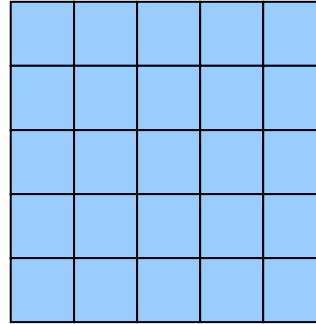

Arrays Multidimensionais

- Apesar de terem mais de uma dimensão, na memória os dados são armazenados linearmente.

- Ex.:

- `int mat[5][5];`

0,0



4,4

0,0

1,0

2,0

3,0

4,0

4,4



Ponteiros e Arrays

Exemplo: acessando um array multidimensional utilizando ponteiros

	Usando array	Usando ponteiro
01	#include <stdio.h>	#include <stdio.h>
02	#include <stdlib.h>	#include <stdlib.h>
03	int main(){	int main(){
04	int mat[2][2] = {{1,2},{3,4}};	int mat[2][2] = {{1,2},{3,4}};
	int i,j;	int * p = &mat[0][0];
05	for (i=0;i<2;i++)	int i;
06	for (j=0;j<2;j++)	for (i=0;i<4;i++)
07	printf("%d\n", mat[i][j]);	printf("%d\n", *(p+i));
08	system("pause");	system("pause");
	return 0;	return 0;
09	}	}
10		
11		

Ponteiro para ponteiro

- Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo.
- Declaração:
 - `tipo_ponteiro **nome_ponteiro;`
 - `**nome_ponteiro` é o conteúdo final da variável apontada;
 - `*nome_ponteiro` é o conteúdo do ponteiro intermediário.

Ponteiro para ponteiro

Exemplo: ponteiro para ponteiro

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int x = 10;
05      int *p = &x;
06      int **p2 = &p;
07      //Endereço em p2
08      printf("Endereco em p2: %p\n",p2);
09      //Conteudo do endereço
10      printf("Conteudo em *p2: %p\n",*p2);
11      //Conteudo do endereço do endereço
12      printf("Conteudo em **p2: %d\n",**p2);
13      system("pause");
14      return 0;
15  }
```

Memória		
#	var	conteúdo
119		
120	int **p2	#122
121		
122	int *p	#124
123		
124	int x	10
125		

Ponteiro para ponteiro

- A quantidade de asteriscos (*) na declaração do ponteiro indica o número de níveis de apontamento.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      //variável inteira
05      int x;
06      //ponteiro para um inteiro (1 nível)
07      int *p1;
08      //ponteiro para ponteiro de inteiro (2 níveis)
09      int **p2;
10      //ponteiro para ponteiro para ponteiro de inteiro(3 níveis)
11      int ***p3;
12      system("pause");
13      return 0;
14  }
```

Ponteiro para ponteiro

- Ex.:

```
char letra='a';  
char *ptrChar;  
char **ptrPtrChar;  
char ***ptrPtr;  
ptrChar = &letra;  
ptrPtrChar = &ptrChar;  
ptrPtr = &ptrPtrChar;
```

Memória		
#	var	conteúdo
119		
120	char ***ptrPtr	#122
121		
122	char **ptrPtrChar	#124
123		
124	char *ptrChar	#126
125		
126	char letra	'a'
127		

Passagem de Parâmetros

- Na linguagem C, os parâmetros de uma função são sempre passados por **valor**, ou seja, uma cópia do valor do parâmetro é feita e passada para a função.
- Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função.

Passagem por valor

```
01  include <stdio.h>
02  include <stdlib.h>
03
04  void soma_mais_um(int n){
05      n = n + 1;
06      printf("Dentro da funcao: x = %d\n",n);
07  }
08
09  int main(){
10      int x = 5;
11      printf("Antes da funcao: x = %d\n",x);
12      soma_mais_um(x);
13      printf("Depois da funcao: x = %d\n",x);
14      system("pause");
15      return 0;
16  }
```

Saída	Antes da funcao: x = 5 Dentro da funcao: x = 6 Depois da funcao: x = 5
-------	--

Passagem por referência

- Quando se quer que o valor da variável mude dentro da função, usa-se passagem de parâmetros por ***referência***.
- Neste tipo de chamada, não se passa para a função o valor da variável, mas a sua ***referência*** (seu endereço na memória)
 - Ex: função scanf()

Passagem por referência

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      int x = 5;
05      printf("Antes do scanf: x = %d\n",x);
06      printf("Digite um numero: ");
07      scanf("%d",&x);
08      printf("Depois do scanf: x = %d\n",x);
09      system("pause");
10      return 0;
11  }
```

Passagem por referência

- Para passar um parâmetro por referência, passamos o ponteiro como parâmetro:

float sqr (float *num);

- Ao se chamar a função, é necessário agora utilizar o operador “&”, igual como é feito com a função **scanf()**:

y = sqr(&x);

Passagem por referência

- No corpo da função, é necessário trabalhar corretamente com o ponteiro

Por valor

```
void soma_mais_um(int n){  
    n = n + 1;  
}
```

Por referência

```
void soma_mais_um(int *n){  
    *n = *n + 1;  
}
```

Passagem por referência

```
01  include <stdio.h>
02  include <stdlib.h>
03
04  void soma_mais_um(int *n){
05      *n = *n + 1;
06      printf("Dentro da funcao: x = %d\n",*n);
07  }
08
09  int main(){
10      int x = 5;
11      printf("Antes da funcao: x = %d\n",x);
12      soma_mais_um(&x);
13      printf("Depois da funcao: x = %d\n",x);
14      system("pause");
15      return 0;
16  }
```

Saída	Antes da funcao: x = 5 Dentro da funcao: x = 6 Depois da funcao: x = 6
-------	--

Exercício

- Crie uma função que troque o valor de dois números inteiros passados por referência.

Exercício

```
void Troca (int*a,int*b){  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Exercício

```
void troca (int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void main()
{
    int x = 2, y = 5;
    troca(x, y);
    printf("x = %d y = %d\n", x, y);
}
```

Exibe: x = 2 y = 5

Passagem por Valor

```
void troca (int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

void main()
{
    int x = 2, y = 5;
    troca(&x, &y);
    printf("x = %d y = %d\n", x, y);
}
```

Exibe: x = 5 y = 2

Passagem por
Referência

Arrays como parâmetros

- são sempre passados por referência para uma função
- Na verdade passamos só o endereço da primeira posição do array
- Precisamos passar o tamanho do array como um outro parâmetro

Arrays como parâmetros

- Na passagem de um array como parâmetro de uma função podemos declarar a função de diferentes maneiras, todas equivalentes:

```
void imprime (int *m, int n);
```

```
void imprime (int m[], int n);
```

```
void imprime (int m[5], int n);
```

Arrays como parâmetros

```
void imprime (int *m,int n){  
    int i;  
    for (i=0; i< n;i++)  
        printf ("%d \n", m[i]);  
}
```

```
int main (){  
    int n[5] = {1,2,3,4,5};  
    imprime(n,5);  
    return 0;  
}
```

Memória		
▪		
▪		
▪		
#	var	conteúdo
123	int *n	#125
124		
125		1
126		2
127		3
128		4
129		5
▪		
▪		
▪		

Arrays como parâmetros

- Para arrays de uma dimensão não é necessário especificar o número de elementos para a função.

```
void imprime (int*m, int n);
```

```
void imprime (int m[], int n);
```

- Para arrays com mais de uma dimensão, é necessário especificar o tamanho de todas as dimensões, exceto a primeira

```
void imprime (int m[][5], int n);
```

Arrays como parâmetros

- O compilador precisar saber o tamanho de cada elemento, não o número de elementos.
- Uma matriz é um array de arrays.
 - **int m[4][5]:** array de 4 elementos onde cada elemento é um array de 5 posições inteiras.
- O compilador precisa saber o tamanho de cada elemento do array.

```
int m[4][5];
```

```
void imprime (int m[][5], int n);
```