



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

- Como qualquer variável, um ponteiro também possui um tipo.
  - `tipo_do_ponteiro *nome_do_ponteiro;`
- É o asterisco (\*) que informa ao compilador que aquela variável não vai guardar um valor mas sim 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  }
```

# Declaração

- Na linguagem C, quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo.
  - Um ponteiro **int\*** aponta para um inteiro, isto é, guarda o endereço de memória onde se encontra guardada uma variável do tipo **int**

# Inicialização

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

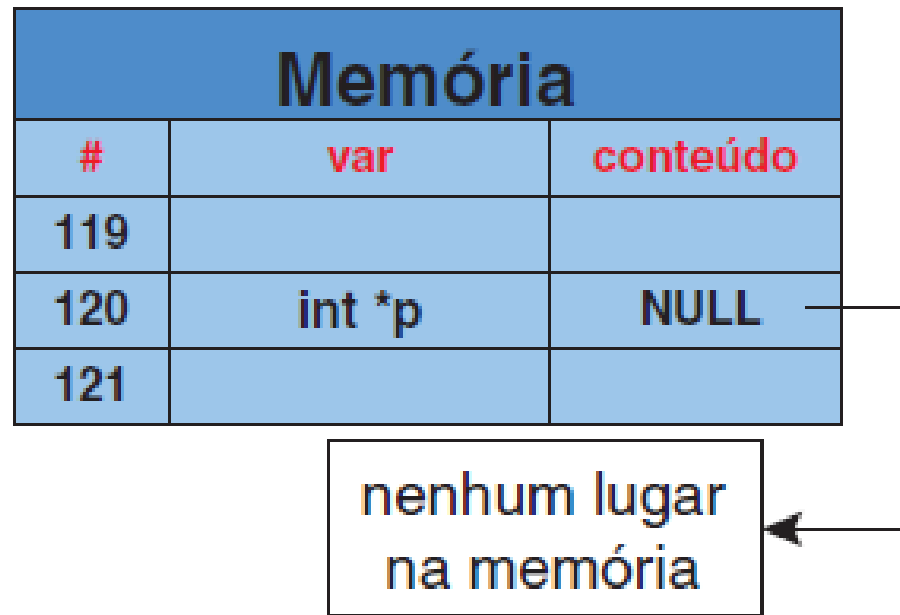
Memória		
#	var	conteúdo
119		
120	int *p	????
121		

# 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		

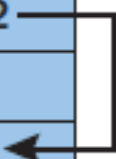
nenhum lugar  
na memória



# 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

- Apenas duas operações aritméticas podem ser utilizadas com o endereço armazenado pelo ponteiro: adição e subtração
  - podemos apenas somar e subtrair valores INTEIROS
    - `p++`; soma +1 no endereço armazenado no ponteiro.
    - `p--`; subtrai 1 no endereço armazenado no ponteiro.

# Operações com ponteiros

- As operações de adição e subtração no endereço dependem do tipo de dado que o ponteiro aponta.
  - Considere um ponteiro para inteiro, **int \***. O tipo **int** ocupa um espaço de 4 bytes na memória.
  - Assim, nas operações de adição e subtração são adicionados/subtraídos 4 bytes por incremento/decremento, pois esse é o tamanho de um inteiro na memória e, portanto, é também o valor mínimo necessário para sair dessa posição reservada de memória

# 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
09             frente de p1\n");
10      else
11          printf("O ponteiro p NAO aponta para uma posicao
12             a frente de p1\n");
13      system("pause");
14      return 0;
15  }
```

# 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
    inteiro
07      pp = &p2;
08      //enta acessar o conteúdo do ponteiro genérico
09      printf("Conteúdo: %d\n",*pp); //ERRO
10      //converte o ponteiro genérico pp para (int *)
    antes de acessar seu conteúdo.
11      printf("Conteúdo: %d\n",*(int*)pp); //CORRETO
12      system("pause");
13      return 0;
14  }
```

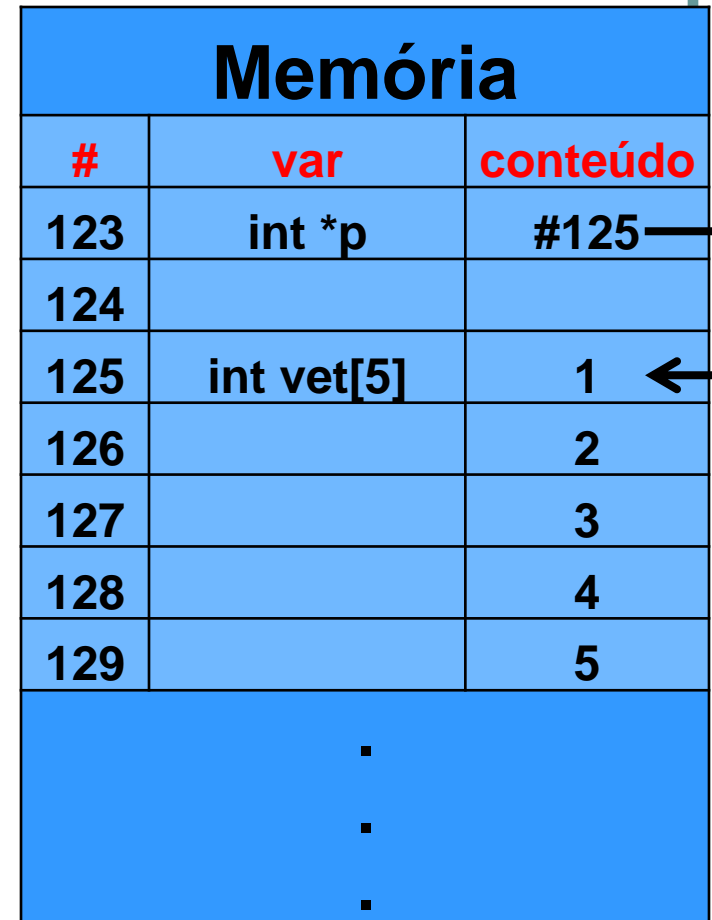
# Ponteiros e Arrays

- Ponteiros e arrays possuem uma ligação muito forte.
  - 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 seqüencial.
  - Como resultado dessa operação, o computador nos devolve um ponteiro que aponta para o começo dessa seqüência 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

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

### Usando ponteiro

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

# Ponteiros e Arrays

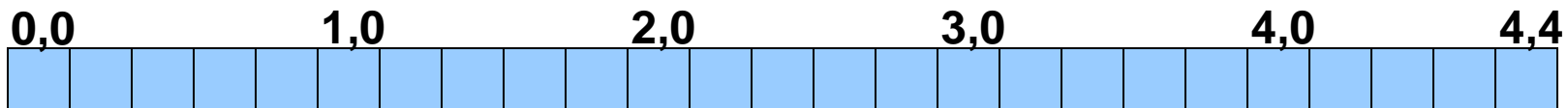
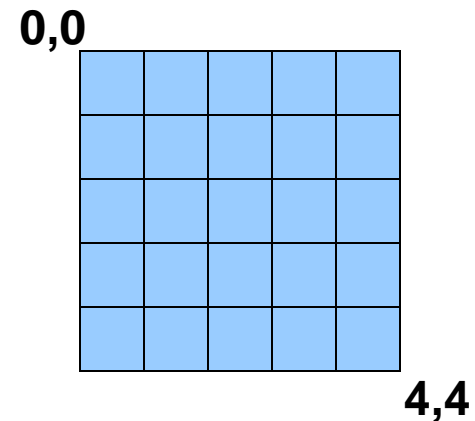
- Os colchetes [ ] substituem o uso conjunto de operações aritméticas e de acesso ao conteúdo (operador “\*”) no acesso ao conteúdo de uma posição de um array ou ponteiro.
  - O valor entre colchetes é o deslocamento a partir da posição inicial. Nesse caso, **p[2]** equivale a **\*(p+2)**.

```
#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;
}
```

# Ponteiros e Arrays

- Arrays Multidimensionais

- Apesar de terem mais de uma dimensão, na memória os dados são armazenados linearmente.
- Ex.:
- `int mat[5][5];`



# Ponteiros e Arrays

- Pode-se percorrer as várias dimensões do array como se existisse apenas uma dimensão. As dimensões mais à direita mudam mais rápido

## Exemplo: acessando um array multidimensional utilizando ponteiros

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