

Estrutura de Dados

Ponteiros Revisão

FELIPE CARVALHO PELLISON

FELIPE.CARVALHO@BARAODEMAUA.BR

A solid orange horizontal bar spanning the width of the slide at the bottom.

Ponteiro

- 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.
 - Um ponteiro é uma variável como qualquer outra do programa – sua diferença é que ela não armazena um valor inteiro, real, caractere ou booleano.
 - Ela serve para armazenar endereços de memória (são valores inteiros sem sinal).

Ponteiro

- É o asterisco (*) que **informa** ao **compilador** que aquela variável **não vai** guardar um **valor** mas sim um **endereço** para o tipo especificado.

```
int *x;
```

```
float *y;
```

```
struct *p;
```

Ponteiro

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

Ponteiro

- Ponteiros **apontam** para uma **posição** de **memória**.
- Cuidado: Ponteiros **não inicializados** **apontam** para um lugar **indefinido**.

- Exemplo `int *p;`


Memória		
posição	variável	conteúdo
119		
120	<code>int *p</code>	????
121		
122		
123		

Ponteiro

- Ponteiros **apontam** para uma **posição** de **memória**.
- Cuidado: Ponteiros **não inicializados** **apontam** para um lugar **indefinido**.

- Exemplo `int *p;`

Memória		
posição	variável	conteúdo
119		
120	<code>int *p</code>	122
121		
122	<code>int c</code>	10
123		

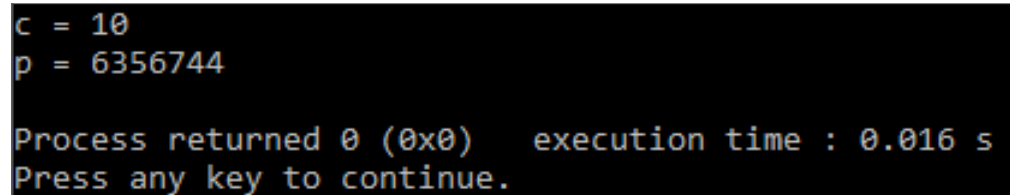


Ponteiro

- 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 &**.
- Ao armazenar o **endereço**, o **ponteiro** estará **apontando** para aquela **variável**.

Ponteiro

```
int main()
{
    int c = 10;
    int *p;
    p = &c;
    cout<<c<<endl;
    cout<<p<<endl;
    return 0;
}
```

A terminal window with a black background and yellow text. It shows the output of the C++ program: 'c = 10' and 'p = 6356744'. Below this, it says 'Process returned 0 (0x0) execution time : 0.016 s' and 'Press any key to continue.'.

```
c = 10
p = 6356744

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```


Ponteiro

- Tendo um **ponteiro armazenado** um **endereço de memória**, como **saber** o valor **guardado** dentro **dessa posição**?
- Para **acessar** o valor **guardado dentro** de uma **posição** na **memória** apontada por um ponteiro, basta usar o operador asterisco “*****” na **frente** do **nome** do ponteiro.

Ponteiro

- De modo **geral**, um **ponteiro** só pode **receber** o **endereço** de **memória** de uma **variável** do mesmo **tipo** do **ponteiro**.
- Isso ocorre porque **diferentes** tipos de **variáveis** ocupam **espaços** de **memória** de tamanhos **diferentes**.
- Na verdade, nós **podemos atribuir** a um ponteiro de **inteiro** (`int *`) o endereço de uma variável do **tipo float**. No entanto, o **compilador assume** que qualquer **endereço** que esse ponteiro **armazene obrigatoriamente** apontará para uma variável do tipo **int**.
- Isso gera problemas na interpretação dos valores.

Ponteiro

- Apenas **duas operações aritméticas** podem ser utilizadas com 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.
 - $p = p+15$: soma +15 no endereço armazenado no ponteiro.

Ponteiro

- As operações de **adição** e **subtração** no endereço **dependem** do **tipo** de dado que o ponteiro **aponta**.
- 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.

Memória		
posição	variável	conteúdo
119		
120	int a	10
121		
122		
123		
124	int b	20
125		
126		
127		
128	char c	'k'
129	char d	's'
130		

Ponteiro

- Existem **operações** que são **ilegais** com **ponteiros**.
- Dividir ou multiplicar ponteiros.
- Somar o endereço de dois ponteiros.
- Não se pode adicionar ou subtrair valores dos tipos float ou double de ponteiros.

Ponteiro

- Sobre o **conteúdo** apontado pelo ponteiro, todas as **operações** são **válidas**.
- $(*p)++$: **incrementa** o **conteúdo** apontado pelo ponteiro
- Multiplicar o conteúdo da variável apontada pelo ponteiro por 10:
 - $*p = (*p) * 10;$

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

- `>`, `<`, `>=`, `<=` para saber qual ponteiro aponta uma posição mais alta na memória.
- Exemplo: código para saber se dois ponteiros apontam para o mesmo endereço.

Ponteiro

```
int main(){  
    int *p, *q, x, y;  
    p = &x;  
    q = &y;  
    if (p == q) cout << "Ponteiros iguais"<<endl;  
    else cout << "Ponteiros diferentes"<<endl;  
    return 0  
}
```


Ponteiro

- 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;`

Ponteiro

```
int main(){
void *pp;
int *p1, p2 = 10;
p1 = &p2;
pp = &p2;

cout<<"Endereco em pp: "<<pp<<endl;

pp=&p1;

cout<<"Endereco em pp: "<<pp<<endl;

pp = p1;

cout << "Endereco em pp: "<<pp<<endl;
return 0;
}
```

```
Endereco em pp: 0060FF04
Endereco em pp: 0060FF08
Endereco em pp: 0060FF04

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Ponteiro

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

Ponteiro

```
int main(){  
    void *pp;  
    int p2 = 10;  
    pp = &p2;  
    cout<<"Conteudo: "<<* (int*) pp<<endl;  
    return 0;  
}
```

Conteudo: 10

Process returned 0 (0x0) execution time : 0.016 s
Press any key to continue.

Ponteiro

- Vetores são **agrupamentos** de **dados** do **mesmo tipo** na memória, alocados **contiguamente**.
- Quando **declaramos** um vetor, informamos ao **computador** para **reservar** uma certa **quantidade** de **memória** a fim de armazenar os elementos do vetor de forma **sequencial**.
- O **computador** nos **devolve** um **ponteiro** que **aponta** para o **começo** dessa sequência de bytes na memória.

Ponteiro

- O nome do vetor (sem índice) é apenas um **ponteiro** que **aponta** para o **primeiro elemento** do **vetor**.

```
int vet[5] = {1,2,3,4,5};
```

```
int *p;
```

```
p = vet;
```

Memória		
posição	variável	conteúdo
119		
120		
121	int *p	123
122		
123	int vet[5]	1
124		2
125		3
126		4
127		5
128		

Ponteiro

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

```
int vet[5] = {1, 2, 3, 4, 5};
```

```
int *p;
```

```
p = vet;
```

- **Neste exemplo:**
 - ***p é equivalente a vet[0];**
 - **vet[índice] é equivalente a**
*** (p+índice);**
 - **vet é equivalente a &vet[0];**
 - **&vet[índice] é equivalente a**
(vet+índice);

Ponteiro

Usando vetor

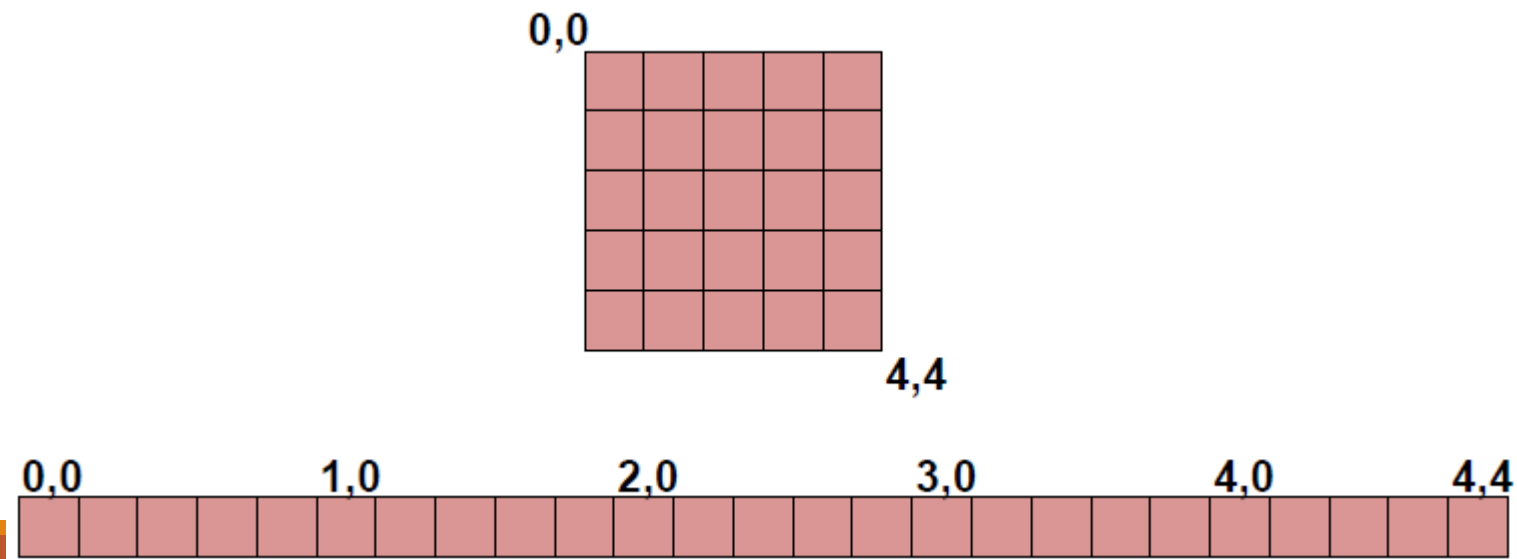
```
int main(){
    int vet[5]={1,2,3,4,5};
    int *p = vet;
    int i;
    for(i=0;i<5;i++){
        cout<<p[i];
    }
}
```

Usando ponteiro

```
int main(){
    int vet[5]={1,2,3,4,5};
    int *p = vet;
    int i;
    for(i=0;i<5;i++){
        cout<<*(p+i);
    }
}
```

Ponteiro

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



Ponteiro

Usando matriz

```
int main(){
    int mat[2][2] = {{1,2},{3,4}};
    int i,j;
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            cout<<mat[i][j];
        }
    }
}
```

Usando ponteiro

```
int main(){
    int mat[2][2] = {{1,2},{3,4}};
    int *p = &mat[0][0];
    int i;
    for(i=0;i<4;i++){
        cout<<*(p+i);
    }
}
```

Ponteiro

- A linguagem C++ permite criar **ponteiros** com **diferentes níveis** de apontamento.
- É possível criar um **ponteiro** que **aponte** para outro **ponteiro**, criando assim **vários** níveis de **apontamento**.
- Assim, um **ponteiro** poderá apontar para outro **ponteiro**, que, por sua vez, aponta para outro **ponteiro**, que aponta para um terceiro **ponteiro** e assim por **diante**.

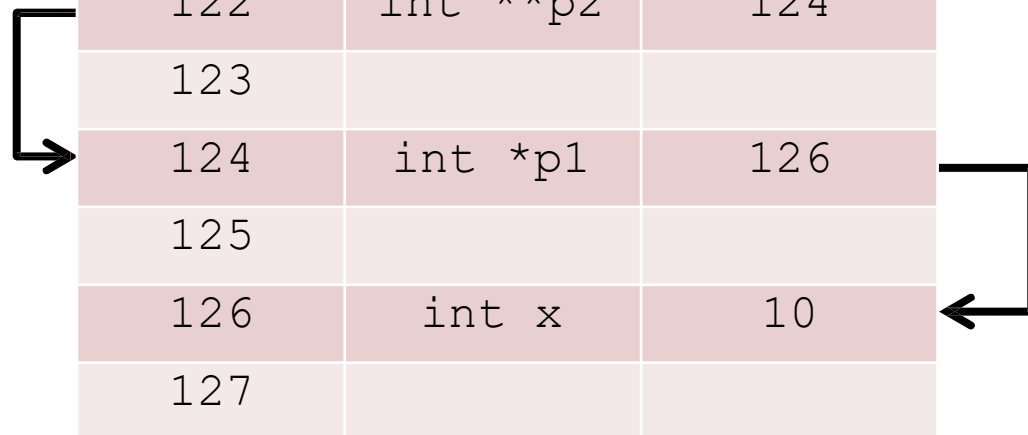
Ponteiro

- Um **ponteiro** para um **ponteiro** é como se você **anotasse** o **endereço** de um papel que tem o **endereço** da casa do seu amigo.
- Podemos declarar um ponteiro para um ponteiro com a seguinte notação `tipo_ponteiro **nome_ponteiro;`
- Acesso ao conteúdo `**nome_ponteiro` é o conteúdo final da variável apontada;
 - `*nome_ponteiro` é o conteúdo do ponteiro intermediário.

Ponteiro

```
int x = 10;  
int *p1 = &x;  
int **p2 = &p1;
```

Memória		
posição	variável	conteúdo
119		
120		
121		
122	int **p2	124
123		
124	int *p1	126
125		
126	int x	10
127		



Ponteiro

- É a **quantidade** de asteriscos (*) na declaração do ponteiro que indica o **número** de **níveis** de **apontamento** que ele possui.

```
int x; // Variável Inteira
int *p1; // Ponteiro 1 nível
int **p2; // Ponteiro 2 níveis
int ***p3; // Ponteiro 3 níveis
```

Ponteiro

```
char letra = 'a';  
char *p1;  
char **p2;  
char ***p3;
```

```
p1 = &letra;  
p2 = &p1;  
p3 = &p2;
```

Memória		
posição	variável	conteúdo
119		
120	char ***p3	122
121		
122	char **p2	124
123		
124	char *p1	126
125		
126	char letra	'a'
127		

```
graph LR
    p3[120: char ***p3] --> p2[122: char **p2]
    p2 --> p1[124: char *p1]
    p1 --> letra[126: char letra]
```


Dynamic Memory Allocation

- Às vezes é útil determinar o tamanho de uma estrutura dinamicamente (em tempo de execução).
- C++ permite que você controle a alocação (new) e desalocação (delete) de memória em um programa para todos os tipos primitivos e também definidos pelo usuário (struct).

Dynamic Memory Allocation

- Você pode usar operador **new** para alocar dinamicamente (ou seja, reservar) o valor exato de memória necessária para manter uma estrutura ou variável em tempo de execução.
- O espaço é reservado no **heap**.
- É possível **acessá-lo através do ponteiro** que o operador **new** retorna;

Sintaxe: `int *p = new int;`

Dynamic Memory Allocation

- Você pode usar operador **delete** para desalocar os espaços reservados.
- O controle dos espaços é devolvido ao **heap**.

Sintaxe:

`delete p;` //onde **p** é um ponteiro para o espaço dinamicamente alocado.

Dynamic Memory Allocation

- Alocação dinâmica de novos arrays (vetores);
- Sintaxe: `int *p = new int[n]` onde `n` é o tamanho do vetor;

Desalocação dinâmica de arrays (vetores):

- Sintaxe: `delete []p;`