# Estruturas de Dados I

## Introdução

Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

# Ponteiros

- Ponteiros são uma ferramenta fundamental em C++.
- Permitem manipular diretamente a memória do computador. Essa é uma vantagem que o C++ possui em relação a diversas outras linguagens.
- Um ponteiro é uma variável que armazena um endereço de memória.
- Um ponteiro é declarado da seguinte forma: tipodedado *nomePonteiro.

  O * é o **operador de desreferência**.
- Por exemplo:

```
int *meuPrimeiroPonteiro;
```

- Acima, declaramos um ponteiro para inteiros de nome "meuPrimeiroPonteiro".
- Para obtermos um endereço e o atribuirmos a um ponteiro, usamos o operador de endereço &.

```
int var = 42;
int *meuPrimeiroPonteiro = &var;
```

- Podemos ler isso como "endereço para var".

# Ponteiros

- Entretanto, se estamos efetuando a atribuição de um endereço a um ponteiro durante a sua declaração inicial, não devemos usar o operador de desreferência. Devemos escrever assim:

```
int var = 42;
int *meuPrimeiroPonteiro;
meuPrimeiroPonteiro = &var;
```

- O operador de desreferência identifica um ponteiro em sua declaração inicial, mas ele possui uma utilização específica fora deste contexto.
- O operador de desreferência (*) é usado para acessar o valor armazenado na memória para a qual um ponteiro está apontando.
- Portanto se fizermos algo assim, exemplificado abaixo, teremos um erro.

```
int var = 42;
int *meuPrimeiroPonteiro;
*meuPrimeiroPonteiro = &var; // Gerará um erro.
```

- Como usamos o operador de desreferência, estamos acessando o valor apontado pelo ponteiro, em vez do ponteiro em si.
- Vamos entender melhor como funciona a memória, ponteiros, e este operador.

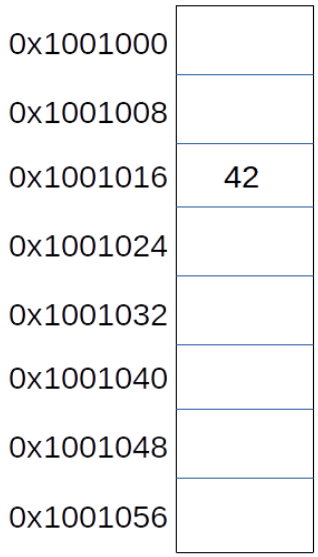
# Ponteiros

- Devem lembrar que a memória principal é formada por células, e cada variável estará armazenada em uma ou mais células.
- Cada variável, portanto, pode ser acessada por seu *endereço* na memória.
- Se declaramos, por exemplo:

```
int var = 42;
```

- Isso reserva um espaço de memória onde essa variável estará alocada.

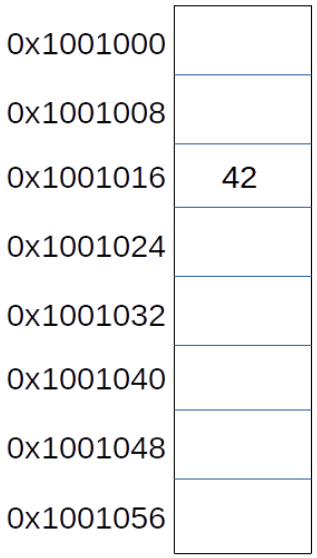
- Suponha que a variável var esteja na posição 0x1001016 da memória.
- O que está ocorrendo de fato é: um espaço de memória, de tamanho para o tipo inteiro (dada a arquitetura), está sendo reservado, e o valor 42 está sendo atribuído a ele.
- A imagem ao lado ilustra essa alocação e atribuição.



# Ponteiros

- Portanto, considerando as informações do slide anterior, ilustradas na imagem ao lado, o que esse código irá imprimir? Vamos pensar juntos.

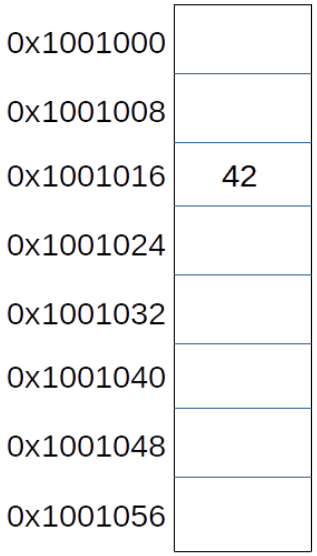
```
#include <iostream>
using namespace std;
int main() {
    int var = 42, *meuPrimeiroPonteiro;
    meuPrimeiroPonteiro = &var;
    cout << meuPrimeiroPonteiro << endl;
    return 0;
}
```



# Ponteiros

- Portanto, considerando as informações do slide anterior, ilustradas na imagem ao lado, o que esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var = 42, *meuPrimeiroPonteiro;
    meuPrimeiroPonteiro = &var;
    cout << meuPrimeiroPonteiro << endl;
    return 0;
}
```

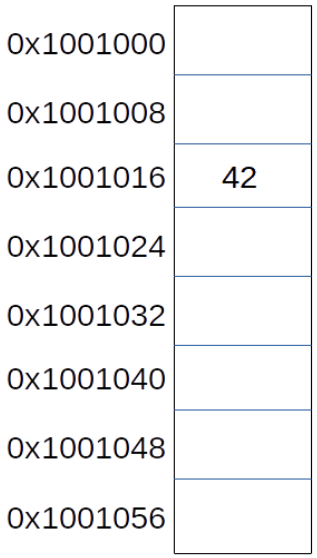



- 0x1001016, que é o endereço da variável var, que encontra-se armazenado no ponteiro meuPrimeiroPonteiro.

# Ponteiros

- Sabemos que o operador de desreferência (*) é usado para acessar o valor armazenado na memória apontado por um ponteiro.
- Considerando as mesmas informações anteriores, o esse código irá imprimir? Vamos pensar juntos.

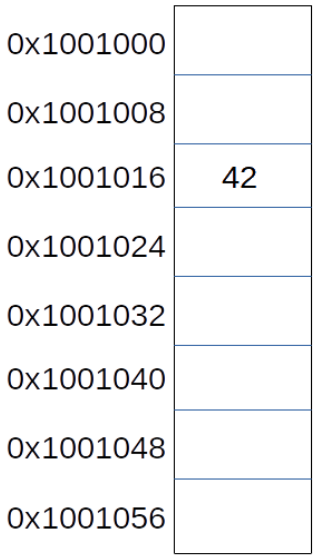
```
#include <iostream>
using namespace std;
int main() {
    int var = 42, *meuPrimeiroPonteiro;
    meuPrimeiroPonteiro = &var;
    cout << *meuPrimeiroPonteiro << endl;
    return 0;
}
```



# Ponteiros

- Sabemos que o operador de desreferência (*) é usado para acessar o valor armazenado na memória apontado por um ponteiro.
- Considerando as mesmas informações anteriores, o esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var = 42, *meuPrimeiroPonteiro;
    meuPrimeiroPonteiro = &var;
    cout << *meuPrimeiroPonteiro << endl;
    return 0;
}
```




- 42, que é o valor da variável var.

# Operador de Desreferência

- O operador de desreferência (*) permite modificarmos o valor de variáveis, já que podemos ter acesso ao endereço de memória delas através do ponteiro.
- Vamos ver mais um código. O esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var = 12, *meuPrimeiroPonteiro = &var;
    *meuPrimeiroPonteiro += 50;
    cout << var << endl;
    return 0;
}
```

# Operador de Desreferência

- O operador de desreferência (*) permite modificarmos o valor de variáveis, já que podemos ter acesso ao endereço de memória delas através do ponteiro.
- O esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var = 12, *meuPrimeiroPonteiro = &var;
    *meuPrimeiroPonteiro += 50;
    cout << var << endl;
    return 0;
}
```

- 62. O valor de var era 12, mas o modificamos através de meuPrimeiroPonteiro, somando 50.

# Operador de Endereço

- O operador de endereço (&) obtém o endereço de variáveis.
- O esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var1, var2, *meuPrimeiroPonteiro = &var1;
    meuPrimeiroPonteiro = &var2;
    cout << meuPrimeiroPonteiro << endl;
    return 0;
}
```

# Operador de Endereço

- O operador de endereço (&) obtém o endereço de variáveis.
- O esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var1, var2, *meuPrimeiroPonteiro = &var1;
    meuPrimeiroPonteiro = &var2;
    cout << meuPrimeiroPonteiro << endl;
    return 0;
}
```

- O endereço de var2.

# Ponteiros

- Ou seja, temos dois operadores muito importantes quando trabalhando com ponteiros: o operador de desreferência (*) e o operador de endereço (&).
- O primeiro permite acessarmos o conteúdo da endereço armazenado em um ponteiro. O segundo permite obtermos o endereço de uma variável.
- Para fixação, digitem e executem esse código, e observem o seu resultado.

```
#include <iostream>
using namespace std;
int main() {
    int var = 10;
    int *meuPrimeiroPonteiro = &var;
    cout << "Valor de var: " << var << endl;
    cout << "Endereço de memória de var: " << &var << endl;
    cout << "Valor apontado por meuPrimeiroPonteiro: " << *meuPrimeiroPont
    cout << "Endereço de memória armazenado em meuPrimeiroPonteiro: " << m
    *meuPrimeiroPonteiro = 20;
    cout << "Novo valor de var: " << var << endl;
    return 0;
}
```

# Ponteiros e Vetores

- Em C++, o nome de um vetor é um ponteiro constante contendo o endereço do primeiro elemento do vetor.
- Portanto, se declaramos:

```
int vetor[10];
```

- vetor é equivalente a &vetor[0].
- Exemplificando:

```
#include <iostream>
using namespace std;
int main() {
    int vetor[10];
    cout << vetor << endl;
    cout << &vetor[0] << endl;
    return 0;
}
```

- Isso imprimirá o mesmo valor duas vezes, já que ambos contém o endereço da primeira posição do vetor.

# Ponteiros

## Exercícios (10).

⇒ Ponteiros

- Escreva um programa que declare dois inteiros e dois ponteiros para esses inteiros. Realize soma, subtração, multiplicação, e divisão, entre os inteiros, usando apenas os seus respectivos ponteiros.
- Escreva um programa que solicite ao usuário dois números inteiros e, em seguida, troque seus valores usando ponteiros.
- Escreva um programa que solicite ao usuário uma string e imprima essa string verticalmente. Use o objeto *string* de C++. Precisa obrigatoriamente usar um ponteiro para obter o endereço da variável de tipo String e utilizá-lo para fazer a impressão.

# A Pilha e o Heap

- Existem dois espaços de memória que precisamos conhecer: a pilha e o heap.

## Pilha

- A pilha é uma área de memória usada para armazenar variáveis locais, parâmetros de função e informações de controle durante a execução do programa.
- A pilha segue a abordagem Last-In-First-Out (LIFO), o que significa que a última variável adicionada à pilha é a primeira a ser retirada.
- Variáveis locais são automaticamente desalocadas quando a função em que foram criadas sai do escopo.

## Heap

- O heap é uma área de memória usada para alocação dinâmica.
- Você pode alocar e liberar memória sob demanda durante a execução do programa.
- Em C++, isso é feito usando os operadores new e delete.
- A memória alocada no heap não é desalocada após saida do escopo da variável alocada nesta.

# Alocação Dinâmica de Memória

- A alocação dinâmica de memória permite que você crie variáveis durante a execução do programa.
- Isso é útil quando não sabemos o tamanho necessário para os dados em tempo de compilação, ou se este tamanho precisa ser flexível.
- Por exemplo, um programa pode perguntar quantos inteiros ele deseja armazenar, e o programa precisará alocar este espaço durante sua execução.
- Ou considere uma lista encadeada (também conhecida como lista ligada), em que precisamos criar novos elementos em tempo de execução.
- É fundamental que saibam que o C++ **não possui garbage collection**. O programador precisa efetuar a liberação de memória sempre que um elemento não for mais necessário.
- Existem dois modos de fazermos alocação dinâmica de memória em C++, com o operador **new** e a função **malloc**. Por enquanto vamos conhecer o operador new.

# Alocação Dinâmica de Memória

- Alocamos memória no heap com operador new. A declaração deve ser seguida do tipo de objeto que queremos alocar, deste modo o compilador saberá quanta memória será necessária.
- O operador new retorna um endereço de memória.
- Vamos ver alguns exemplos.
- Aqui, alocamos dinamicamente um inteiro.

```
int *var = new int;
```

- Aqui, alocamos dinamicamente um vetor de inteiros de 10 elementos.

```
int *vetor = new int[10];
```

- Como perceberam, precisamos usar ponteiros para efetuar a alocação dinâmica.
- Isso é porque um ponteiro é capaz de **apontar** para um espaço de memória alocado por nós.

# Alocação Dinâmica de Memória

- Como já aprenderam, se quisermos acessar o valor da posição alocada por um ponteiro, precisamos usar o operador de desreferência.
- Não devem ter dificuldade em dizer o que será impresso no código abaixo. Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var = 5;
    int *ponteiro = new int;
    *ponteiro = 10;
    var += *ponteiro;
    cout << var;
    return 0;
}
```

# Alocação Dinâmica de Memória

- E quanto a vetores? Já aprenderam que um vetor é um ponteiro constante. A memória apontada por ele está na pilha.
- Mas podemos criar vetores dinamicamente, e sua memória estará no heap.
- O que esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var = 5, *ponteiro;
    ponteiro = new int[10];
    for(int i = 0; i < 10; i++)
        ponteiro[i] = i+1;
    var += ponteiro[4];
    cout << var;
    return 0;
}
```

# Alocação Dinâmica de Memória

- E quanto a vetores? Já aprenderam que um vetor é um ponteiro constante. A memória apontada por ele está na pilha.
- Mas podemos criar vetores dinamicamente, e sua memória estará no heap.
- O que esse código irá imprimir? Vamos pensar juntos.

```
#include <iostream>
using namespace std;
int main() {
    int var = 5, *ponteiro;
    ponteiro = new int[10];
    for(int i = 0; i < 10; i++)
        ponteiro[i] = i+1;
    var += ponteiro[4];
    cout << var;
    return 0;
}
```

- Irá imprimir 10.

# Alocação Dinâmica de Memória

- Mas lembrem que C++ não possui coleta de lixo automática.
- É seu papel como programador liberar a memória após o seu uso.
- Para este fim, utilizamos o operador **delete**.
- A sintaxe para o seu uso é:

```
delete nomeDoPonteiro;
```

- Mas por que devemos nos preocupar com isso?

# Vazamentos de Memória

- Vazamento de memória (em inglês, *memory leaks*) é um problema comum em programação, ocorrendo quando a memória alocada dinamicamente não é liberada após não ser mais necessária.
- Isso pode levar a um esgotamento gradual dos recursos do sistema, a degradação do desempenho do programa, e até um crash do programa sendo executado.
- Vamos conhecer alguns exemplos.

```
int var = 10, *ponteiro = new int;
ponteiro = &var;
```

- Acima, perdemos a referência para um espaço de memória alocado dinamicamente.
- Ou seja, nem temos nem como desalocar este espaço posteriormente, este espaço do heap ficará ocupado durante toda a execução de nosso programa.

# Vazamentos de Memória

- Outro exemplo.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string *ponteiro;
    for(int i = 0; i < 3; i++) {
        cout << "Qual o seu nome?" << endl;
        ponteiro = new string;
        cin >> *ponteiro;
        cout << "Seu nome é: " << *ponteiro << endl;
    }
    return 0;
}
```

- Como no exemplo anterior, perdemos a referência para um espaço de memoria alocado previamente.

# Ponteiros

## Exercícios (11).

⇒ Alocação Dinâmica de Memória

- Escreva um programa que solicite ao usuário o tamanho de um vetor de inteiros e, em seguida, aloque dinamicamente memória para o vetor. Peça ao usuário para inserir os elementos e calcule a soma deles.
- Escreva um programa que leia um número indeterminado de notas de alunos até que seja inserido um valor negativo. Use alocação dinâmica para armazenar as notas em um vetor e, em seguida, calcule a média.
- Escreva um programa que solicite um número inteiro positivo $n$ ao usuário e aloque dinamicamente um vetor para armazenar os primeiros $n$ números da sequência de Fibonacci. Imprima a sequência resultante.