

# Estrutura Sequencial e Entrada e Saída de Dados

---

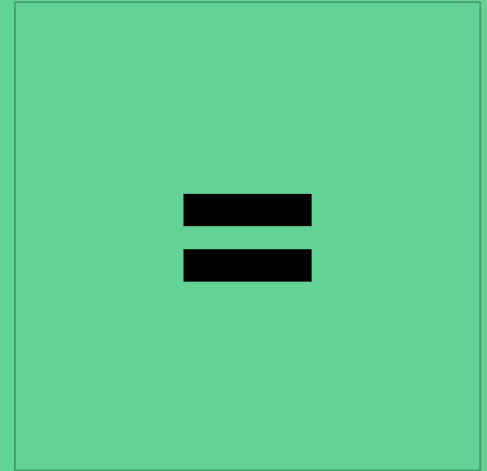
Prof. Joaquim Quinteiro Uchôa  
Profa. Juliana Galvani Gregghi  
Profa. Marluce Rodrigues Pereira  
Profa. Paula Christina Cardoso  
Prof. Renato Ramos da Silva



# Roteiro

- Atribuição de valores
- Operações aritméticas
- Biblioteca `<cmath>`
- Entrada e saída de dados
- Arquivos e streams
- Formatação de código  
(comentários e indentação)
- Estrutura sequencial

# Atribuição de valores



# Atribuição de valores

- Para que uma variável possa ser utilizada em um programa, ela deve ter um valor associado a ela.
- Em programação, o ato de associar um determinado valor a uma variável é denominado atribuição.
- **Sintaxe em pseudocódigo:**

**identificador\_da\_variável** ← **valor**

Operador de  
atribuição

# Atribuição de valores

- Para que uma variável possa ser utilizada em um programa, ela deve ter um valor associado a ela.
- Em programação, o ato de associar um determinado valor a uma variável é denominado atribuição.
- **Sintaxe em C++:**

**identificador\_da\_variável = valor ;**



Operador de  
atribuição

# Atribuição de valores

- Exemplos:

```
nome = "Janderson";  
idade = 20;  
pi = 3.14159;
```

- Estes valores permanecem associados às variáveis correspondentes até que o programa os altere por meio de uma nova atribuição ou até que o programa como um todo se encerre.

# Operação de atribuição

**identificador\_da\_variável = valor ;**

- De modo geral, o lado direito de um operador de atribuição pode ser:

- Um valor fixo. Exemplo:

idade = 20;

Variável **idade** recebe o valor fixo 20

- Uma outra variável. Exemplo:

ladoA = 7;

ladoB = ladoA;

Variável **ladoA** recebe o valor fixo 7

Variável **ladoB** recebe o valor que está armazenado na variável **ladoA**

# Operação de atribuição

**identificador\_da\_variável = valor ;**

- De modo geral, o lado direito de um operador de atribuição pode ser:
  - Uma expressão formada por valores fixos e/ou variáveis. Exemplo:

base = 2;



Variável **base** recebe o valor fixo 2

altura = 9;



Variável **altura** recebe o valor fixo 9

area = (base\*altura)/2;



Variável **area** recebe o valor resultante da multiplicação dos valores armazenados nas variáveis **base** e **altura** dividido pelo valor fixo 2



# Cuidados ao fazer uma atribuição

- (1º) Uma variável só pode ser associada a um valor (atribuição), se a variável já tiver sido declarada anteriormente.

```
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
    int idade;
    idade=20;
    return 0;
}
```



```
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
    idade=20;
    int idade;
    return 0;
}
```



# Cuidados ao fazer uma atribuição

- (2º) Números em ponto flutuante contêm um ponto para denotar as casas decimais (notação americana para separar casas decimais).

```
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
    float pi;
    pi = 3.14;
    return 0;
}
```



```
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
    float pi;
    pi = 3,14;
    return 0;
}
```



# Cuidados ao fazer uma atribuição

- (3º) Quando se atribui um valor em ponto-flutuante à uma variável inteira, ocorre uma conversão e podem ocorrer perdas.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x;
    x = 20.743;
    return 0;
}
```

Variável **x** passa a armazenar o valor inteiro 20. Neste caso, a parte decimal é perdida e o valor é truncado. Note que não há arredondamento.

# Cuidados ao fazer uma atribuição

- (4º) Quando se atribui um valor inteiro a uma variável em ponto-flutuante, ocorre uma conversão e podem ocorrer perdas.

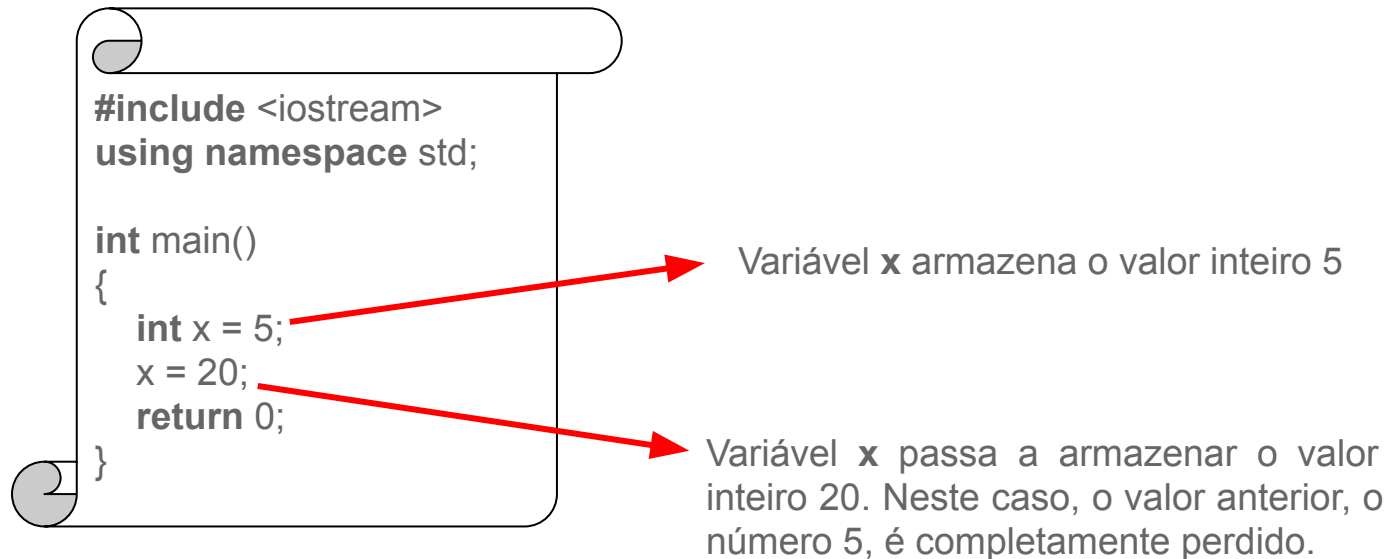
```
#include <iostream>
using namespace std;
```

```
int main()
{
    int i = 20;
    float x;
    x = i;
    return 0;
}
```

Variável **x** passa a armazenar o valor flutuante 20. Neste caso, a representação é feita usando aproximações e podem ocorrer erros caso seja necessário um valor inteiro, para comparação, por exemplo.

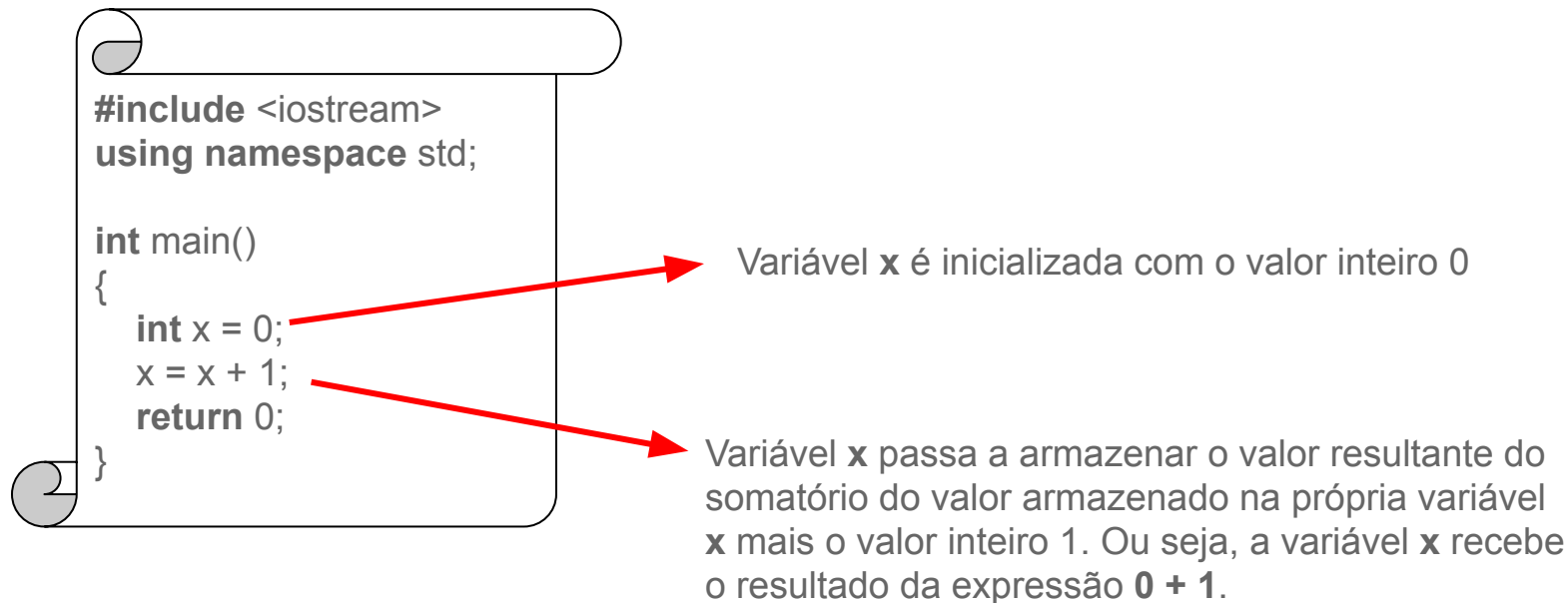
# Cuidados ao fazer uma atribuição

- (5º) Ao associarmos um valor à uma variável, qualquer dado anterior que estiver armazenado na variável é descartado.



# Cuidados ao fazer uma atribuição

- (6º) O identificador de uma variável pode aparecer simultaneamente tanto a esquerda quanto a direita do operador de atribuição.



```
#include <iostream>
using namespace std;

int main()
{
    int x = 0;
    x = x + 1;
    return 0;
}
```

Diagram illustrating variable assignment in C++ code:

- Red arrow pointing to `int x = 0;`: Variável **x** é inicializada com o valor inteiro 0
- Red arrow pointing to `x = x + 1;`: Variável **x** passa a armazenar o valor resultante do somatório do valor armazenado na própria variável **x** mais o valor inteiro 1. Ou seja, a variável **x** recebe o resultado da expressão **0 + 1**.

# Variáveis em ponto-flutuante

O exemplo de código ao lado teoricamente deveria imprimir valores de 1.1 até 22.0, pulando de 1.1 em 1.1.

*Não se preocupe com os detalhes, uma vez que você verá entrada e saída e estruturas de repetição mais à frente no curso.*

```
#include <iostream>
using namespace std;

int main() {
    float x = 1.1;
    while (x <= 22.0) {
        cout << x << endl;
        x = x + 1.1;
    }
    return 0;
}
```

# Variáveis em ponto-flutuante

O exemplo de código ao lado teoricamente deveria imprimir valores de 1.1 até 22.0, pulando de 1.1 em 1.1.

**Isso não  
ocorre!**

Não se preocupe com os detalhes que você verá em estruturas de repetição mais à frente no curso.

```
#include <iostream>
using namespace std;
```

```
int main() {
    float x = 1.1;
    while (x <= 22.0) {
        cout << x << " ";
        x = x + 1.1;
    }
    return 0;
}
```

```
1.1
2.2
3.3
4.4
5.5
6.6
7.7
8.8
9.9
11
12.1
13.2
14.3
15.4
16.5
17.6
18.7
19.8
20.9
```



# Variáveis em ponto-flutuante

O exemplo de código ao lado teoricamente deveria imprimir valores de 1.1 até 22.0, pulando de 1.1 em 1.1.

Não acontece assim os  
devido ao fato de que você verá  
então estruturas de  
repetição presente no curso.

**Isso não  
ocorre!**

**22 não é  
impresso!**

```
#include <iostream>
using namespace std;
```

```
int main() {
    float x = 1.1;
    while (x <= 22.0) {
        cout << x << " ";
        x = x + 1.1;
    }
    return 0;
}
```

1.1  
2.2  
3.3  
4.4  
5.5  
6.6  
7.7  
8.8  
9.9  
11  
12.1  
13.2  
14.3  
15.4  
16.5  
17.6  
18.7  
19.8  
20.9

# Variáveis em ponto-flutuante

Vamos modificar um pouco o código apenas para imprimir os valores com uma certa precisão.

*Novamente, não se preocupe com os detalhes, uma vez que você verá entrada e saída e estruturas de repetição mais à frente no curso.*

```
#include <iostream>
using namespace std;
```

```
int main() {
    float x = 1.1;
    cout.precision(10);
    while (x <= 22.0) {
        cout << x << endl;
        x = x + 1.1;
    }
    return 0;
}
```

# Variáveis em ponto-flutuante

Vamos modificar um pouco o código apenas para imprimir os valores com uma certa precisão.

**Valores não  
são exatos!**

Não se preocupe com a precisão da saída e com as estruturas de repetição mais à frente no curso.

```
#include <iostream>
using namespace std;
```

```
int main() {
    float x = 1.1;
    while (x <= 20) {
        cout << x << " ";
        x = x + 1.1;
    }
    return 0;
}
```

```
1.100000024
2.200000048
3.299999952
4.400000095
5.5
6.599999905
7.699999809
8.800000191
9.900000572
11.00000095
12.10000134
13.20000172
14.3000021
15.40000248
16.50000191
17.60000229
18.70000267
19.80000305
20.90000343
```

# Variáveis em ponto-flutuante

Vamos modificar um código apenas para os valores com uma precisão.

**Valores não são exatos!**

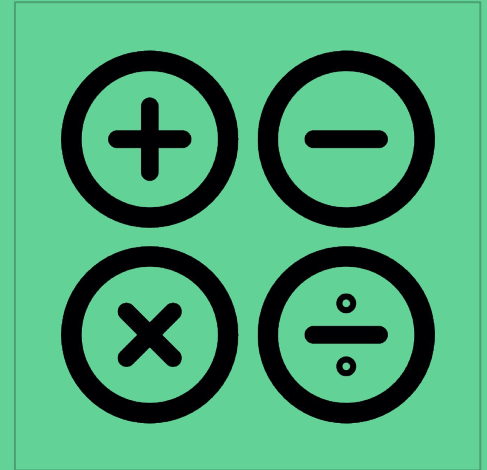
No código, as estruturas de repetição não são exatas frente no curso.

**Valores em ponto-flutuante são armazenados com aproximações, mas com uma precisão que é adequada para os problemas em que eles são utilizados.**

**Entretanto, considerá-los exatos é um erro e pode gerar erros de programação críticos.**

```
1.100000024
2.200000048
3.299999952
4.400000095
5.5
6.599999905
7.699999809
8.800000191
9.900000572
11.00000095
12.10000134
13.20000172
14.3000021
15.40000248
16.50000191
17.60000229
18.70000267
19.80000305
20.90000343
```

# Operações aritméticas



# Operações aritméticas

- As diversas operações aritméticas (ex: soma, multiplicação, etc.) são necessárias em diversos algoritmos.
- Para isso, são usados os operadores aritméticos, que podem ser divididos em duas classes: operadores unários e operadores binários.

# Operadores aritméticos unários

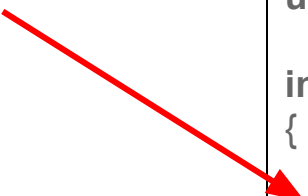
Operadores unários: possuem um único operando.

- Adição unária: +
- Subtração unária: -

## Exemplo:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x = 7;
    int y = -x;
    return 0;
}
```



# Operadores aritméticos binários

Operadores binários: possuem dois operandos em notação infixa.

- Adição: +
- Subtração: -
- Multiplicação: \*
- Divisão: /
- Resto (módulo): %

**Exemplo:**

```
#include <iostream>
using namespace std;

int main()
{
    int x = 7;
    int y = x % 5;
    return 0;
}
```

Neste exemplo, a variável **y** armazena o valor resultante do resto (módulo) da divisão de 7 por 5. Ou seja, **y** armazena o número inteiro 2.



# Particularidades dos operadores aritméticos

- Sobre números inteiros e números em ponto flutuante:
  - Com exceção do operador de resto (%), todos os demais operadores aritméticos permitem operandos inteiros ou em ponto flutuante;
    - Quando inteiros e números em ponto flutuante são misturados, o resultado da expressão sempre será um número em ponto flutuante. Exemplo: **9 + 2.5** terá como resultado o número **11.5**.
    - A divisão de dois números inteiros sempre resultará em um número também inteiro. Exemplo: **7 / 2** terá como resultado o número **3**.
  - O operador de resto (%) apenas permite operandos do tipo inteiro. Qualquer tentativa do contrário gerará um erro de sintaxe em C++.

# Divisão de números inteiros

Como comentado anteriormente, a divisão de dois números inteiros sempre resultará em um número inteiro. Exemplo:

```
#include <iostream>
using namespace std;

int main()
{
    float x = 7 / 2;
    return 0;
}
```

Mesmo a variável **x** sendo declarada como **float**, o resultado da divisão de dois números inteiros sempre será um número inteiro. Portanto, neste exemplo, a variável **x** armazena o valor **3** e não o valor **3.5**.

# Divisão de números inteiros

Para fazermos com que o resultado da divisão apresente casas decimais, é necessário garantirmos que um dos operandos da expressão seja um número em ponto flutuante. Exemplos:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    float x = 7.0 / 2;
    return 0;
}
```

```
#include <iostream>
using namespace std;
```

```
int main()
{
    float x = 7 / 2.0;
    return 0;
}
```

Em ambos os exemplos,  
a variável **x** armazena o  
valor **3.5**.

# Divisão de números inteiros

Uma outra forma para garantirmos que um dos operandos da expressão seja um número em ponto flutuante é utilizarmos estratégias de conversão para transformar um número inteiro em um em ponto flutuante.

Existem duas abordagens de conversão em C++, uma é utilizando a operação de conversão (**casting**) e a outra é utilizando uma função especial, chamada construtor, de número flutuante.

É importante ressaltar que apesar de estarmos tratando o caso de conversão de inteiro para ponto flutuante, essas duas abordagens podem funcionar em outras situações, mas deve-se verificar a disponibilidade do recurso antes.

# Divisão de números inteiros

Na primeira forma, operador de conversão (**casting**), utilizamos o tipo desejado entre parênteses, antes do valor a ser convertido. Exemplos:

```
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
```

```
    float x = (float) 7 / 2;
```

```
    return 0;
```

```
}
```

```
#include <iostream>
using namespace std;
```

```
int main()
```

```
{
```

```
    float x = 7 / (float) 2;
```

```
    return 0;
```

```
}
```

Em ambos os exemplos,  
a variável **x** armazena o  
valor **3.5**.

# Divisão de números inteiros

Na segunda forma, uso de construtor, utilizamos o tipo desejado como se fosse uma função, com o valor a ser convertido entre parênteses. Exemplos:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    float x = float(7) / 2;
    return 0;
}
```

```
#include <iostream>
using namespace std;
```

```
int main()
{
    float x = 7 / float(2);
    return 0;
}
```

Em ambos os exemplos,  
a variável **x** armazena o  
valor **3.5**.

# Precedência de operadores aritméticos

- Quando uma expressão aritmética precisa ser avaliada em um programa, o mesmo processa a expressão dando prioridade a certos operadores em detrimento de outros.
- Operações com maior prioridade (precedência) são avaliadas primeiro.  
Ordem de precedência dos operadores aritméticos:

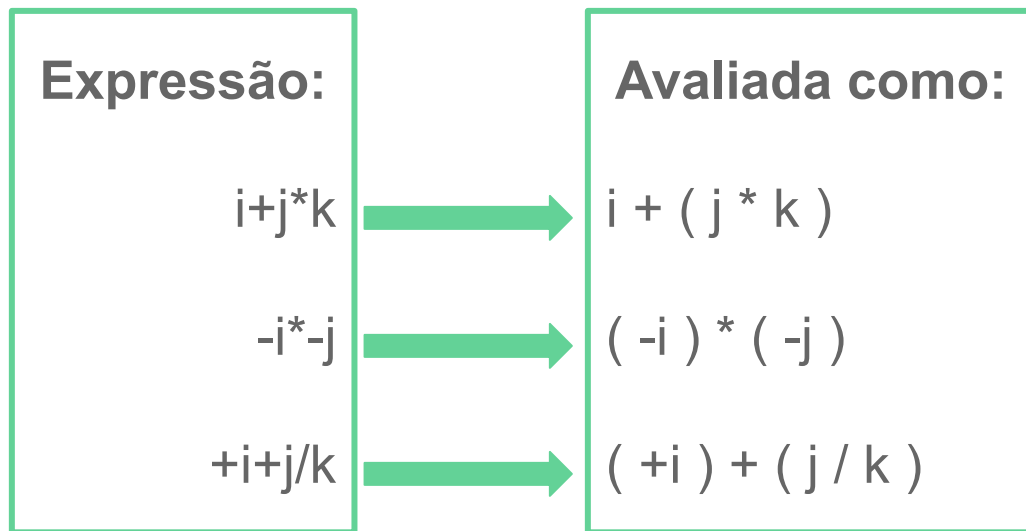
**(1º) Operadores unários: + e -**

**(2º) Multiplicação, divisão e resto: \*, / e %**

**(3º) Soma e subtração: + e -**



# Precedência de operadores aritméticos

- Exemplos:





# Associatividade de operadores aritméticos

- Quando uma expressão aritmética com operadores com o mesmo nível de precedência precisa ser avaliada em um programa, o mesmo processa a expressão seguindo as regras de associatividade dos operadores.
- Operadores aritméticos possuem duas regras possíveis de associatividade:
  - Operadores associativos à esquerda: as operações mais à esquerda são executadas primeiro; 
  - Operadores associativos à direita: as operações mais à direita são executadas primeiro. 

# Associatividade de operadores aritméticos

- Operadores associativos à esquerda:

Adição binária: +  
Subtração binária: -  
Multiplicação: \*  
Divisão: /  
Resto: %

Exemplo:

Expressão:

$a * b / c * d$

Avaliada como:

$(( (a * b) / c) * d)$

- Operadores associativos à direita:

Adição unário: +  
Subtração unária: -

Exemplo:

Expressão:

$- + i$

Avaliada como:

$-( + i)$

# Exemplos: precedência e associatividade

## EXEMPLO 1:

$$1 + \underbrace{10 * 4} - 6$$

$$\underbrace{1 + 40} - 6$$

$$\underbrace{41 - 6}$$

35

## EXEMPLO 2:

$$\underbrace{(1 + 10)} * 4 - 6$$

$$\underbrace{(11) * 4} - 6$$

$$\underbrace{44 - 6}$$

38

## EXEMPLO 3:

$$\underbrace{(1 + 10)} * \underbrace{(4 - 6)}$$

$$\underbrace{(11) * \underbrace{(4 - 6)}}$$

$$\underbrace{(11) * \underbrace{(-2)}}$$

-22

# Recomendação em precedência de operadores



- Tentar decorar a precedência de operadores é uma má ideia para programadores iniciantes.
- Sempre use parênteses (e apenas parênteses) para garantir que uma determinada expressão seja avaliada na ordem desejada.

# Operadores aritméticos e de atribuição

- Operadores aritméticos e o operador de atribuição podem ser combinados de forma a gerar uma atribuição composta.
- Uma atribuição composta pode ser criada apenas quando a variável a esquerda de um comando de atribuição também aparece em seu lado direito. Sendo assim, instruções do tipo:

**variável = variável operador valor ;**

podem ser combinadas de modo a formar instruções com a sintaxe:

**variável operador= valor ;**

# Operadores aritméticos e de atribuição

- Por exemplo, a instrução:

```
i = i + 2 ;
```

poderia ser combinada, gerando a atribuição composta:

```
i += 2 ;
```

- Outras possibilidades para operadores compostos: -=, \*=, /= e %=.

# Incremento e Decremento

- Uma das operações mais comuns em programação é aumentar ou diminuir o valor de uma variável inteira em uma unidade. Por exemplo:

```
i = i + 1 ;  
j = j - 1 ;
```

- Podemos realizar este tipo de operação por meio de atribuições compostas:

```
i += 1 ;  
j -= 1 ;
```

# Incremento e Decremento

- Na prática, estas operações são tão comuns que C++ oferece ainda um terceiro recurso para indicá-las, que são os operadores de incremento (++) e de decremento (--).
- Os operadores de incremento e decremento, implicam tanto no aumento ou diminuição de uma variável em uma unidade quanto na atribuição deste novo valor para a mesma variável.
- Nos nossos exemplos anteriores, poderíamos então fazer:

```
i++ ;  
j-- ;
```



# Biblioteca <cmath>



# Biblioteca <cmath>

- Comumente, programas que realizam cálculos matemáticos um pouco mais complexos se utilizam de funções prontas da biblioteca <cmath>.
- Bibliotecas podem ser vistas como uma coleção de subprogramas (funções) utilizados no desenvolvimento de softwares.
- Com a biblioteca <cmath> é possível encontrar funções para calcular potências, raiz quadrada, funções trigonométricas, além de constantes para números irracionais, como por exemplo, o número  $\pi$  ( $\pi$ ).

# Biblioteca <cmath>

- Para utilizar funções da biblioteca <cmath>, o programador deve:
  - (1º) Incluir a biblioteca <cmath> no início de seu próprio programa. Para isso, basta inserir a instrução no seu programa:

```
#include <cmath>
```

- (2º) Utilizar o nome da função de interesse, passando os parâmetros necessários para a mesma. De modo geral, a sintaxe será:

```
nome_da_função(parâmetros)
```

Se houver mais de um parâmetro, os mesmos devem ser separados por vírgula.

# Biblioteca <cmath>

- Alguns exemplos de funções:

| Funções | O que faz                 |
|---------|---------------------------|
| pow()   | Potenciação               |
| sqrt()  | Raiz quadrada             |
| cos()   | Cosseno                   |
| sin()   | Seno                      |
| tan()   | Tangente                  |
| ceil()  | Arredondamento para cima  |
| floor() | Arredondamento para baixo |
| abs()   | Valor absoluto (módulo)   |

# Biblioteca <cmath>

- Exemplo: cálculo da raiz quadrada de um número.

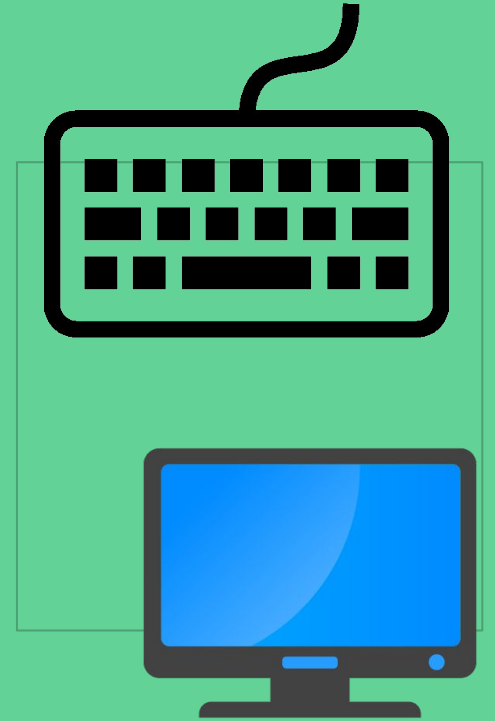
```
#include <iostream>
#include <cmath>
using namespace std;
```

```
int main()
{
    float numero = 25;
    float resultado = sqrt(numero);
    return 0;
}
```

A biblioteca <cmath> deve ser incluída antes da declaração da função main()

Outras funções da biblioteca podem ser encontradas em:  
<http://www.cplusplus.com/reference/cmath/>

# Entrada e saída de dados



# Entrada e saída de dados

- Um algoritmo geralmente realiza uma série de manipulações sobre um certo conjunto de dados.
  - Esses dados são geralmente fornecidos pelo usuário através do teclado, do mouse, podem ser buscados no disco rígido ou recebidos através de uma rede de internet, por exemplo.
  - Uma vez que as manipulações sejam realizadas, os resultados obtidos podem ser enviados para o monitor ou para uma impressora, por exemplo.



# Entrada e saída de dados

- O recebimento de dados por um algoritmo e a apresentação de resultados, independentemente dos dispositivos utilizados, são denominados Operações de Entrada e Saída.
- Em pseudocódigo, há duas funções empregadas para denotar quaisquer operações de entrada e saída:
  - Função **leia**: empregada como Operação de Entrada;
  - Função **escreva**: empregada como Operação de Saída;

```
while a number is greater than zero,  
do the following:  
  Add the number to the running total.  
once the number is greater than zero
```

[pseudocode]



# Entrada e saída de dados

```
while a number is greater than zero,  
do the following:  
Add the number to the running total.  
once the number is greater than zero
```

[pseudocode]

| Função     | Parâmetros   | Interpretação Usual  |
|------------|--|--|
| leia(x)    | Variável <b>x</b> de qualquer tipo fundamental de dados                | Um valor digitado no teclado é armazenado na variável <b>x</b> |
| escreva(x) | Variável, constante ou expressão de qualquer tipo fundamental de dados | O valor definido por <b>x</b> é apresentado no monitor.        |

# Entrada e saída de dados

- Em C++, estas duas funções são relacionadas a dois objetos distintos:



| Função     | Objeto C++ | Dispositivo acessado comumente |
|------------|------------|--------------------------------|
| leia(x)    | cin        | Teclado                        |
| escreva(x) | cout       | Monitor                        |

- Vamos ver então, a partir de agora, a sintaxe para estes dois objetos.

# Saída de dados

- Sintaxe para **cout** (para mensagens fixas):
  - Para utilizar **cout** deve-se:
    - Incluir a biblioteca para operações de entrada e saída do C++. Nome da biblioteca: **<iostream>**;
    - Utilizar o ambiente de nomeação padrão do C++. Nome do ambiente de nomeação: **std**;

```
cout << "Mensagem de texto" ;
```

Símbolos obrigatórios

Mensagens de texto devem aparecer entre símbolos de aspas

# Saída de dados

- Sintaxe para **cout** (para mensagens fixas):
  - Para utilizar **cout** deve-se:
    - Incluir a biblioteca para operações de entrada e saída do C++. Nome da biblioteca: **<iostream>**;
    - Utilizar o ambiente de nomeação: **std**;

Operador de encadeamento:  
“recebe”

**cout** << “Mensagem de texto” ;

Símbolos obrigatórios

Mensagens de texto devem  
aparecer entre símbolos de aspas

# Saída de dados

- Exemplo com **cout**: mensagem fixa

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Olá mundo!";
    return 0;
}
```




Esquecer de incluir a biblioteca **<iostream>** ou o ambiente de nomeação **std** ocasionará um erro de sintaxe quando o objeto **cout** for utilizado.

Não colocar a mensagem a ser exibida no dispositivo de saída padrão (monitor) entre os símbolos de aspas também gerará um erro de sintaxe.

# Saída de dados

- Sintaxe para **cout** (para variáveis ou expressões):

```
cout << nome_da_variável ;
```



Identificadores de variáveis não devem aparecer entre símbolos de aspas

# Saída de dados

- Exemplo com **cout**: raiz quadrada de 25

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    float numero = 25;
    cout << numero;
    cout << sqrt(numero);
    return 0;
}
```



Colocar o nome de uma variável, ou uma expressão, entre símbolos de aspas **não** irá gerar um erro de sintaxe. Contudo, neste caso, ao invés de exibir no dispositivo de saída padrão (monitor) o valor armazenado pela variável, será exibido no dispositivo o texto entre aspas.

# Saída de dados

- Sintaxe para o objeto **cout** (formatações):
  - O objeto **cout** pode exibir simultaneamente no dispositivo de saída padrão múltiplas mensagens fixas e valores de variáveis/expressões.
  - Para exibir múltiplas informações, basta separar cada uma delas por operadores <<.

```
cout << “Mensagem de texto” << nome_da_variável ;
```

- Podemos forçar uma quebra de linha no dispositivo de saída padrão por meio do manipulador **endl**.



# Saída de dados

- Exemplo: raiz quadrada de 25

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    float numero = 25;
    cout << "A raiz de " << numero << " é o valor "
         << sqrt(numero) << endl;
    return 0;
}
```

# Saída de dados

- Exemplo: modificando saída de ponto-flutuante, imprimindo a parte decimal com 10 dígitos:

```
#include <iostream>
#include <cmath>
using namespace std;
```

```
int main() {
    float numero = 25;
    cout.precision(10);
    cout.setf(ios::fixed);
    cout << "A raiz de " << numero << " é o valor "
         << sqrt(numero) << endl;
    return 0;
}
```

Informa precisão  
(casas decimais)

Informa que parte decimal  
deve ser impressa, mesmo  
que seja zero.

# Relembrando... Entrada e saída de dados

- Em C++, estas duas funções são relacionadas a dois objetos distintos:



| Função     | Objeto C++ | Dispositivo acessado comumente |
|------------|------------|--------------------------------|
| leia(x)    | cin        | Teclado                        |
| escreva(x) | cout       | Monitor                        |

- Vamos ver então, a partir de agora, a sintaxe para o objeto de entrada.

# Entrada de dados

- Sintaxe para o objeto **cin**:
  - Para utilizar o operador **cin** se deve:
    - Incluir a biblioteca para operações de entrada e saída do C++. Nome da biblioteca: **<iostream>**;
    - Utilizar o ambiente de nomeação padrão do C++. Nome do ambiente de nomeação: **std**;

```
cin >> nome_da_variável ;
```

→ Símbolos obrigatórios

→ Identificadores não devem aparecer entre símbolos de aspas

# Entrada de dados

- Sintaxe para o objeto **cin**:
  - Para utilizar o operador **cin** se deve:
    - Incluir a biblioteca para operações de entrada e saída do C++. Nome da biblioteca: **<iostream>**;
    - Utilizar o ambiente de nomeação: **std**;

Operador de encadeamento:  
“envia”



```
cin >> nome_da_variável ;
```

Símbolos obrigatórios

Identificadores não devem aparecer  
entre símbolos de aspas

# Entrada de dados

- Exemplo com **cin**: raiz quadrada de um número qualquer

```
#include <iostream>
#include <cmath>
using namespace std;
```

```
int main()
{
    float numero;
    cin >> numero;
    cout << sqrt(numero);
    return 0;
}
```



Esquecer de incluir a biblioteca **<iostream>** ou o ambiente de nomeação **std** ocasionará um erro de sintaxe quando o objeto **cin** for utilizado.

# Entrada de dados

- Sintaxe para o objeto **cin**:
  - O objeto **cin** permite o armazenamento simultâneo de múltiplos valores em múltiplas variáveis, desde que os valores sejam separados por espaços, tabulação ou linhas em branco no dispositivo de entrada padrão (teclado);
  - Para isso, é necessário que exista uma variável diferente para cada valor a ser lido;
  - Cada nova variável deve ser separada pelos símbolos >>.

```
cin >> nome_da_variável1 >> nome_da_variável2;
```

# Entrada de dados

- Exemplo com **cin**: soma de dois números quaisquer

```
#include <iostream>
using namespace std;

int main()
{
    float numero1, numero2;
    cin >> numero1 >> numero2;
    cout << numero1+numero2;
    return 0;
}
```



Inverter os símbolos `>>` do operador **cin** pelos símbolos `<<` do operador **cout** gerará um erro de sintaxe. O contrário também gerará um erro de sintaxe.



# Arquivos e streams



# Arquivos - i

O armazenamento de dados em variáveis é temporário, isto é, os dados se perdem quando o programa termina sua execução.

Para manter grandes quantidades de dados de forma permanente são usados arquivos.

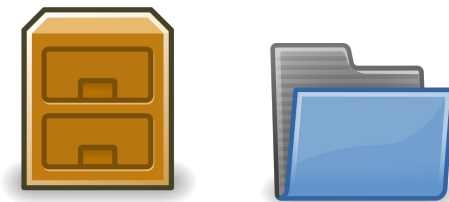
Arquivos são armazenados em dispositivos de memória secundária, tais como: discos rígidos, CD, DVD, pendrives, cartões de memória, etc.



# Arquivos - ii

Dispositivos de memória secundária podem reter grandes volumes de dados por longos períodos de tempo.

Processos de entrada e saída de dados em arquivos são executados por funções especialmente criadas para esta finalidade e disponibilizadas em bibliotecas específicas.



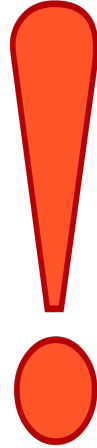
# Operações comuns em arquivos

- Abertura e fechamento de arquivos;
- Apagar um arquivo;
- Leitura e escrita;
- Indicação de que o fim de arquivo foi atingido;
- Posicionar o arquivo em um ponto determinado.

Nem sempre essas ações são fornecidas na mesma biblioteca em uma dada linguagem de programação.

# Arquivos - importante

- Para escrever ou ler, é necessário abrir o arquivo.
- Ao final das operações necessárias o programa deve fechar o arquivo.
- Quando um arquivo é fechado, o conteúdo dos buffers é descarregado para o dispositivo externo.



# Arquivos e streams

Dados em arquivos podem ser manipulados em dois diferentes tipos de fluxos de dados (**stream**):

- **Fluxo de texto:** composto por uma sequência de bytes que denotam uma sequência de caracteres. Para isso, utiliza-se geralmente caracteres da tabela ASCII ou UTF-8.
- **Fluxo binário:** composto por uma sequência de bytes lidos, sem tradução, diretamente do dispositivo externo. Os dados são gravados exatamente como estão organizados na memória do computador. Dados estão sujeitos às convenções dos programas que os geraram.

# Streams

Em C++, arquivos são entendidos como streams cujos dados estão guardados em um dispositivo de armazenamento secundário. Existem três tipos de streams para manipulação de arquivos:

- **ifstream** (para entrada/leitura de dados)
- **ofstream** (para saída/escrita de dados)
- **fstream** (para entrada e saída)

Para ter acesso, deve-se incluir a biblioteca `<fstream>`.

# Abertura de arquivos

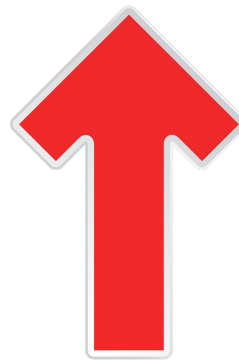
O trecho

```
ofstream arquivo("meus_dados.txt");
```

abre o arquivo “meus\_dados.txt” para a escrita de dados e é equivalente a:

```
ofstream arquivo;  
arquivo.open("meus_dados.txt");
```

A primeira opção é preferida, sempre que possível, uma vez que alguns compiladores otimizam o código.





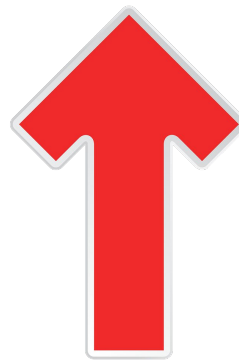
# Abertura de arquivos

Já o trecho

```
ifstream arquivo("meus_dados.txt");
```

abre o arquivo “meus\_dados.txt” para a leitura de dados e é equivalente a:

```
ifstream arquivo;  
arquivo.open("meus_dados.txt");
```



A primeira opção também é preferida, sempre que possível.

# Manipulando arquivos com *streams*

## Sobre a escrita (*ofstream*):

- Se o arquivo não existir, o mesmo será criado;
- Se o arquivo já existir, seu conteúdo será apagado.



## Sobre a leitura (*ifstream*):

- Se o arquivo não existir, o mesmo não será criado.

***A forma mais simples de utilizar arquivos é como um fluxo de texto:***

- Escrevemos no arquivo de modo similar ao uso de **cout**;
- Lemos no arquivo de modo similar ao uso de **cin**



# Escrevendo no arquivo em fluxo de texto

```
#include <fstream>

using namespace std;

int main(){
    ofstream arquivo("meus_dados.txt");
    arquivo << "Escrevendo no arquivo..."
              << endl;
    arquivo.close();
    return 0;
}
```



# Escrevendo no arquivo em fluxo de texto

```
#include <fstream>
```

```
using namespace std;
```

```
int main(){  
    ofstream arquivo("meus_dados.txt");  
    arquivo << "Escrevendo no arquivo..."  
        << endl;  
    arquivo.close();  
    return 0;  
}
```



A variável arquivo, do tipo ofstream, é associada ao arquivo meus\_dados.txt.

# Escrevendo no arquivo em fluxo de texto



```
#include <fstream>

using namespace std;

int main(){
    ofstream arquivo("meus_dados.txt");
    arquivo << "Escrevendo no arquivo";
    arquivo << endl;
    arquivo.close();
    return 0;
}
```

close() é chamado automaticamente caso o programa termine normalmente, sem uso de exit().

# Lendo do arquivo em fluxo de texto

```
#include <fstream>
using namespace std;
int main(){
    ifstream arquivo("meus_dados.txt");
    string dados;
    arquivo >> dados;
    cout << dados << endl;
    arquivo.close();
    return 0;
}
```



# Formatação de código

```
4  #include "utilities.h"
5
6  Game game;
7
8  int main(int argc, char **argv) {
9
10     game = Game();
11
12     game.initialize();
13 }
```

# Formatação de código

Você já implementou algumas linhas de código e já conhece o funcionamento básico. Já imaginou um código com 5 mil linhas? Imagine este código escrito uma forma adequada.

Agora que você está escrevendo os primeiros programas, já deve ter percebido que é possível representar um mesmo programa de diferentes formas. Também já deve ter pensado em utilizar textos para explicar algo do código produzido.

Assim, vamos ver alguns recursos para formatar o código e deixá-lo mais apresentável? Você pode usar **comentários**, bem como **indentação**, para fazer com que o código fique mais legível.



# Comentários

- **Definição:** comentários são textos que aparecem em um programa, mas não são executados, servindo para que possamos colocar lembretes e outras informações no código.
  - Usualmente são utilizados para descrever um algoritmo ou parte dele;
  - Aumentam a legibilidade ou entendimento da solução expressa pelo programa;
  - Uso comum: indicar o significado das variáveis e constantes utilizadas.

# Comentários

- Em C++, há duas formas diferentes de criar comentários:
  - **Comentário de linha:** tipo de comentário que engloba uma única linha. Sempre começam com os símbolos `//`.
  - **Comentário de bloco:** tipo de comentário que podem englobar várias linhas de texto. Sempre começam com os símbolos `/*` e terminam com os símbolos `*/`.
- Em um mesmo programa, um programador pode inserir tantos comentários quanto achar conveniente.

# Exemplo - Comentários

```
#include <iostream>
/*Meu primeiro programa de
IALG com comentários */
//by Janderson R.
using namespace std;

int main()
{
    cout << "Olá mundo!";
    return 0;
}
```



Esquecer de colocar os símbolos `*/` ao final de um comentário de bloco gerará um erro de sintaxe.

# Linhas em branco e indentação

São utilizados para aumentar a legibilidade de um algoritmo;

São utilizados para delimitar blocos de comandos, facilitando o entendimento da lógica empregada;

Estruturas compostas devem sempre ser indentadas para facilitar o entendimento de quais comandos serão executados por cada alternativa possível.

Por estrutura composta entende-se qualquer trecho de código entre { e }.

# Exemplo - indentação

```
#include <iostream>
using namespace std;
int main( ) {
    int x;
    cin >> x;
    cout << "Exemplo de indentação" << endl;
    if ( x > 10 ) {
        cout << "Outro exemplo de indentação" << endl;
    }
    return 0;
}
```

# Usar tabulação ou espaços?

Indentação pode ser conseguida com uso de tabulação (tabs) ou espaços.

Maioria dos desenvolvedores preferem espaços, pois fica independente do editor. Esta é a recomendação dos professores.

O uso de tabulação ou espaços é indiferente, mas deve-se evitar o uso de ambos em um mesmo código, pois gerará confusão.

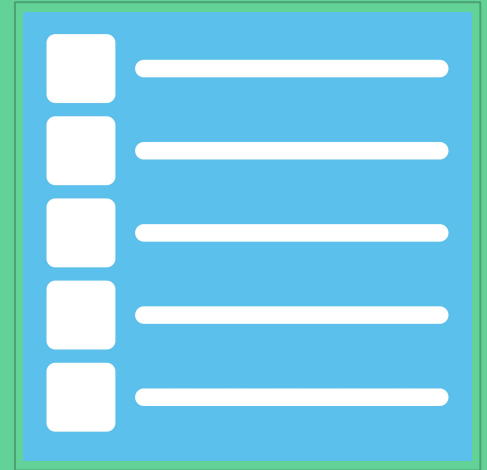
# Indentação e IDEs

A maioria das IDEs efetua indentação automaticamente.

É possível configurar a IDE para usar espaços ou tabulação por padrão.

No caso de uso de tabulação, é possível configurar o tamanho usado em cada tabulação.

# Estrutura Sequencial





# Definição - Estrutura Sequencial

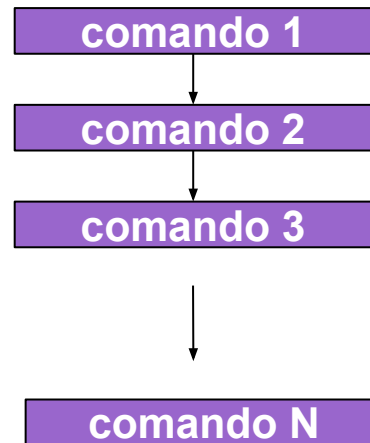
## ESTRUTURA SEQUENCIAL

Passos executados em uma sequência pré-definida.

### Pseudocódigo

```
comando 1;  
comando 2;  
comando 3;  
...  
comando N;
```

### Fluxograma



# Exemplo 1 - Cálculo de média

O algoritmo ao lado, escrito em pseudocódigo, é responsável por calcular a média aritmética entre duas avaliações realizadas por um certo aluno.

**Algoritmo** Calcula\_Media

nome: **cadeia de caracteres**

nota1, nota2, media: **real**

**Início**

**escreva**("Digite o nome do aluno: ")

**leia**(nome)

**escreva**("Digite as notas N1 e N2 do aluno: ")

**leia**(nota1, nota2)

    media  $\leftarrow$  (nota1+nota2)/2

**escreva**("A média do aluno ", nome, " é ", media)

**Fim**

# Exemplo 1 - Cálculo de média (C++)

```
//Algoritmo Calcula_Media
#include <iostream>
using namespace std;
int main() {
    string nome;
    float nota1, nota2, media;
    cout << "Digite o nome do aluno: ";
    cin >> nome;
    cout << "Digite as notas N1 e N2 do aluno: ";
    cin >> nota1 >> nota2;
    media = (nota1 + nota2)/2;
    cout << "A média do aluno " << nome << " é "
        << media << endl;
    return 0;
}
```

# Exemplo 1 - Cálculo de média (C++)

```
//Algoritmo Calcula_Media
#include <iostream>
using namespace std;
int main() {
    string nome;
    float nota1, nota2, media;
    cout << "Digite o nome do aluno: ";
    cin >> nome;
    cout << "Digite as notas N1 e N2 do aluno: ";
    cin >> nota1 >> nota2;
    media = (nota1 + nota2)/2;
    cout << "A média do aluno " << nome << " é "
         << media << endl;
    return 0;
}
```

Parênteses usados para garantir que a soma seja feita antes da adição (prioridade de operadores)

## Exemplo 2 - Área de um triângulo qualquer

Calcular a área de um triângulo dados seus três lados A, B, C. A forma de cálculo é a seguinte, usando semiperímetro:

$$P = (A + B + C)/2$$

$$Área = \sqrt{P \times (P - A) \times (P - B) \times (P - C)}$$

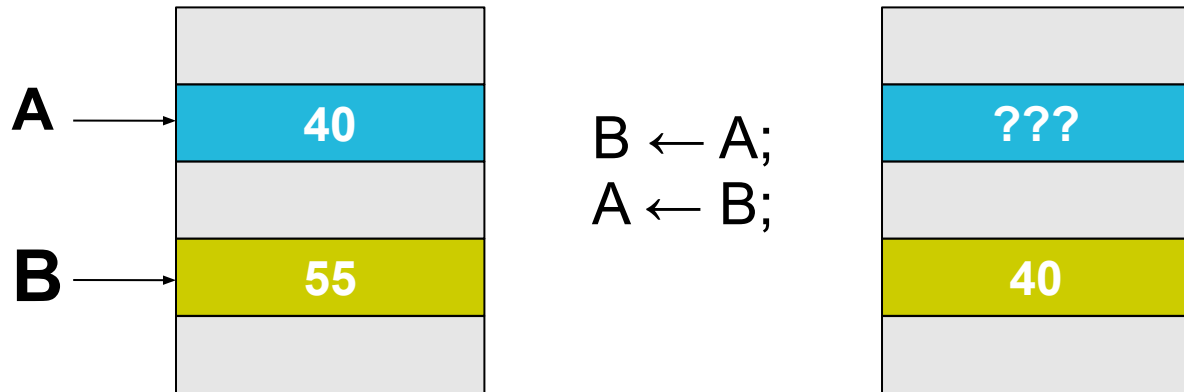
## Exemplo 2 - Área de um triângulo qualquer

Calcular a área de um triângulo dados seus três lados A, B e C.

```
Programa AREA_TRIANGULO
A, B, C: Inteiros
P, AREA: Reais
Início
    Leia(A,B,C)
     $P \leftarrow (A + B + C) / 2$ 
    AREA  $\leftarrow$ 
        RAIZ_QUADRADA( $P * (P - A) * (P - B) * (P - C)$ )
    Escreva(AREA)
Fim
```

# Exemplo 3 - Troca de Valor - i

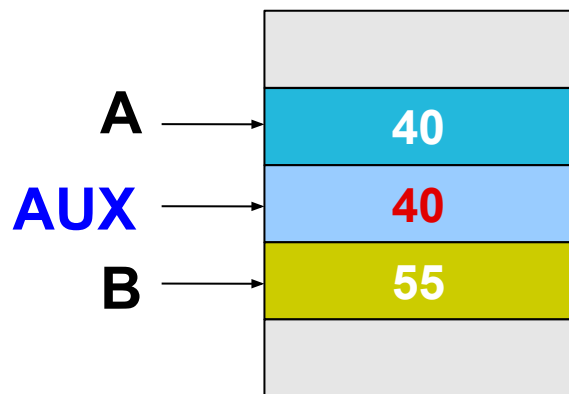
Receber o valor de duas variáveis e trocar o valor entre elas.



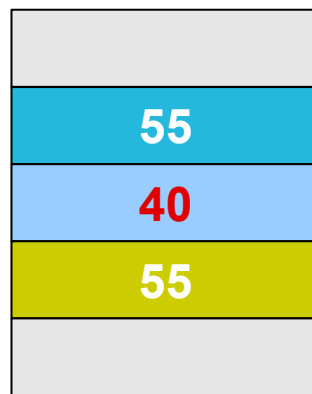
**O valor original de B  
foi perdido!!!**

# Exemplo 3 - Troca de Valor - ii

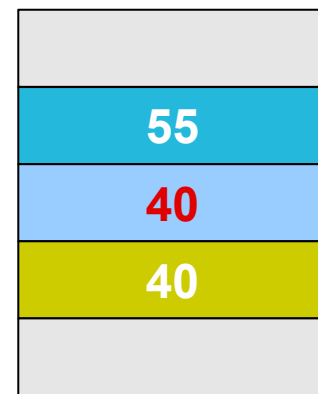
Usando uma variável auxiliar...



$AUX \leftarrow A;$



$A \leftarrow B;$



$B \leftarrow AUX;$



# Exemplo 3 - Troca de Valor - iii

Receber o valor de duas variáveis e trocar o valor entre elas.

```
programa TROCA
A,B,AUX : inteiros;
Inicio
    leia(A, B);
    escreva(A, B);
    AUX ← A;
    A ← B;
    B ← AUX;
    escreva(A, B);
Fim
```

# Sobre o Material



# Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do Departamento de Computação Aplicada - DAC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).