

# Breve introdução à Linguagem C++

Estrutura de Dados — QXD0010



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz  
gomes.atilio@ufc.br

Universidade Federal do Ceará

2º semestre/2023



# Tópicos

- Compilação e execução
- Elementos básicos da linguagem C++
- Estruturas de seleção e repetição
- Vetores
- Structs e enumerações
- Ponteiros
- Alocação dinâmica de memória
- Ponteiros para ponteiros
- Arrays de caracteres



# Tutoriais online para estudo rápido e consulta

- Site LearnCpp.com: <https://www.learncpp.com>
- Site cppreference.com: <https://en.cppreference.com/w/>
- Site cplusplus.com: <http://www.cplusplus.com/reference>

# Apresentação da Linguagem C++

- Linguagem **multi-paradigma**, de uso geral e nível médio.



# Apresentação da Linguagem C++

- Linguagem **multi-paradigma**, de uso geral e nível médio.

- **Bjarne Stroustrup** desenvolveu o C++ (originalmente com nome *C with Classes*) em 1983 no Bell Labs como um adicional à linguagem C.



- Desde a década de 1990 possui forte uso **acadêmico e comercial**.

# Apresentação da Linguagem C++

- Linguagem **multi-paradigma**, de uso geral e nível médio.

- **Bjarne Stroustrup** desenvolveu o C++ (originalmente com nome *C with Classes*) em 1983 no Bell Labs como um adicional à linguagem C.



- Desde a década de 1990 possui forte uso **acadêmico e comercial**.
- Novas características foram adicionadas com o tempo (funções virtuais, sobrecarga de operadores, herança múltipla, templates e tratamento de exceções).

# Apresentação da Linguagem C++

Padronizações da linguagem:

- C++98: ISO de 1998
- C++03: revisão em 2003
- Em 2011, o padrão C++11 foi lançado, adicionando vários recursos novos, ampliando ainda mais a biblioteca padrão e fornecendo mais facilidades aos programadores C++.
- C++14 foi lançada em dezembro de 2014.
- C++17 foi lançada em dezembro de 2017.
- Versão mais recente: C++20, lançada em 2020, mas nem todas as funcionalidades estão disponíveis ainda no compilador g++.
- C++23 em andamento



# Linguagens mais buscadas na internet

Aug 2023	Aug 2022	Change	Programming Language	
1	1			Python
2	2			C
3	4	▲		C++
4	3	▼		Java
5	5			C#
6	8	▲		JavaScript
7	6	▼		Visual Basic
8	9	▲		SQL
9	7	▼		Assembly language
10	10			PHP

Fonte: <https://www.tiobe.com/tiobe-index/>

# C++ inclui bibliotecas do C

- <cassert> (assert.h)
- <cctype> (ctype.h)
- <cerrno> (errno.h)
- <cfloat> (float.h)
- <climits> (limits.h)
- <cmath> (math.h)
- <cstdlib> (stdlib.h)
- <cstring> (string.h)
- <ctime> (time.h)
- <cstddef> (stddef.h)
- <ciso646> (iso646.h)
- <clocale> (locale.h)
- <csetjmp> (setjmp.h)
- <csignal> (signal.h)
- <cstdarg> (stdarg.h)
- <cstdbool> (stdbool.h)
- <stdint> (stdint.h)
- <stdio> (stdio.h)
- <cuchar> (uchar.h)
- <wchar> (wchar.h)
- <wctype> (wctype.h)

# Primeiro Programa — Hello World

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello world!\n";
5     return 0;
6 }
```

# Primeiro Programa — Hello World

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello world!\n";
5     return 0;
6 }
```

Supondo que o programa acima esteja no arquivo `programa.cpp`, para compilar no terminal do Linux e depois executar:

- `$ g++ -Wall -Wextra -pedantic-errors programa.cpp -o main`
- `$ ./main`
  - `-Wall` é a abreviação de “warn all” – ativa (quase) todos os avisos que o `g++` pode lhe informar.
  - `-Wextra` ativa alguns avisos extras que não são ativados por `-Wall`
  - `-pedantic-errors` rejeita programas que não seguem o padrão ISO C++
  - `-o` altera o nome do arquivo de saída.

# Elementos básicos da linguagem C++



# Elementos básicos da linguagem

Como em outras linguagens:

- Comentários de código
- Variáveis e Constantes
- Identificadores
- Tipos Fundamentais
- Representação numérica
- Vetores, strings, ponteiros, estruturas e enumerações
- Estruturas de controle de fluxo
- Funções
- entre outras...

# Comentários

- Um **comentário** é uma descrição inserida diretamente no código-fonte do programa e que é ignorada pelo compilador. Serve apenas para uso do programador.
- C++ permite fazer comentários de duas maneiras diferentes:  
**por linha** ou **por bloco**.

```
1 #include <iostream>
2
3 int main() {
4     /* --- Exemplo de comentario em bloco ---
5     A funcao std::cout
6     serve para
7     escrever na tela
8     */
9     std::cout << "Hello World"; // Um comentario em linha
10
11     return 0;
12 }
```

# Variáveis





# Variáveis

- Em computação, uma **variável** é uma posição de memória onde poderemos guardar determinado dado ou valor e modificá-lo ao longo da execução do programa.

# Variáveis

- Em computação, uma **variável** é uma posição de memória onde poderemos guardar determinado dado ou valor e modificá-lo ao longo da execução do programa.
- Quando criamos uma variável e armazenamos um valor dentro dela, o computador reserva um espaço associado a um endereço de memória onde podemos guardar o valor dessa variável.

# Variáveis

- Em computação, uma **variável** é uma posição de memória onde poderemos guardar determinado dado ou valor e modificá-lo ao longo da execução do programa.
- Quando criamos uma variável e armazenamos um valor dentro dela, o computador reserva um espaço associado a um endereço de memória onde podemos guardar o valor dessa variável.
- O nome da variável é chamado **identificador**.

# Variáveis

- Em computação, uma **variável** é uma posição de memória onde poderemos guardar determinado dado ou valor e modificá-lo ao longo da execução do programa.
- Quando criamos uma variável e armazenamos um valor dentro dela, o computador reserva um espaço associado a um endereço de memória onde podemos guardar o valor dessa variável.
- O nome da variável é chamado **identificador**.

```
1 #include <iostream>
2
3 int main() {
4     int x; //declara a variavel mas nao define o valor
5     std::cout << "x = " << x << "\n";
6     x = 5; //define o valor de x como sendo 5
7     std::cout << "x = " << x << std::endl;
8     return 0;
9 }
```

# Inicialização de variáveis

```
1 #include <iostream> // prog07.cpp
2
3 int main() {
4     int z = 23; // por atribuicao -- assignment
5     std::cout << "Valor de z: " << z << '\n';
6     int k = 45.89; // ok
7     std::cout << "Valor de k: " << k << '\n';
```

# Inicialização de variáveis

```
1 #include <iostream> // prog07.cpp
2
3 int main() {
4     int z = 23; // por atribuicao -- assignment
5     std::cout << "Valor de z: " << z << '\n';
6     int k = 45.89; // ok
7     std::cout << "Valor de k: " << k << '\n';
8
9     int w( 23 ); // direct initialization
10    std::cout << "Valor de w: " << w << '\n';
11    int s( 32.75 ); // ok
12    std::cout << "Valor de s: " << s << '\n';
```

# Inicialização de variáveis

```
1 #include <iostream> // prog07.cpp
2
3 int main() {
4     int z = 23; // por atribuicao -- assignment
5     std::cout << "Valor de z: " << z << '\n';
6     int k = 45.89; // ok
7     std::cout << "Valor de k: " << k << '\n';
8
9     int w( 23 ); // direct initialization
10    std::cout << "Valor de w: " << w << '\n';
11    int s( 32.75 ); // ok
12    std::cout << "Valor de s: " << s << '\n';
13
14    int y{ 23 }; // uniform initialization (C++11)
15    std::cout << "Valor de y: " << y << '\n';
16    int x{ 5.6 }; /** ERRO DE COMPILACAO **/
17    std::cout << "Valor de x: " << x << '\n';
18
19    return 0;
20 }
```

# Identificadores

A linguagem C++ estipula algumas regras para a escolha dos identificadores:

- Um **identificador** é um conjunto de caracteres que podem ser letras, números ou *underscores* (`_`).



# Identificadores

A linguagem C++ estipula algumas regras para a escolha dos identificadores:

- Um **identificador** é um conjunto de caracteres que podem ser letras, números ou *underscores* (`_`).
- O identificador deve sempre iniciar com uma letra ou o *underscore* (`_`).

# Identificadores

A linguagem C++ estipula algumas regras para a escolha dos identificadores:

- Um **identificador** é um conjunto de caracteres que podem ser letras, números ou *underscores* (`_`).
- O identificador deve sempre iniciar com uma letra ou o *underscore* (`_`).
- A linguagem C é **case-sensitive**, ou seja, uma palavra escrita utilizando caracteres maiúsculos é diferente da mesma palavra escrita com caracteres minúsculos.

# Identificadores

A linguagem C++ estipula algumas regras para a escolha dos identificadores:

- Um **identificador** é um conjunto de caracteres que podem ser letras, números ou *underscores* (`_`).
- O identificador deve sempre iniciar com uma letra ou o *underscore* (`_`).
- A linguagem C é **case-sensitive**, ou seja, uma palavra escrita utilizando caracteres maiúsculos é diferente da mesma palavra escrita com caracteres minúsculos.
- Palavras reservadas não podem ser usadas como nome de variáveis.

# Identificadores

A linguagem C++ estipula algumas regras para a escolha dos identificadores:

- Um **identificador** é um conjunto de caracteres que podem ser letras, números ou *underscores* (`_`).
- O identificador deve sempre iniciar com uma letra ou o *underscore* (`_`).
- A linguagem C é **case-sensitive**, ou seja, uma palavra escrita utilizando caracteres maiúsculos é diferente da mesma palavra escrita com caracteres minúsculos.
- Palavras reservadas não podem ser usadas como nome de variáveis.
- As **palavras reservadas** são um conjunto de 84 palavras reservadas da linguagem C++. Elas formam a sintaxe da linguagem e possuem funções específicas.

# Palavras-chave da linguagem C++

A partir do padrão C++17:

<code>alignas (C++11)</code> <code>alignof (C++11)</code> <code>and</code> <code>and_eq</code> <code>asm</code> <code>auto</code> <code>bitand</code> <code>bitor</code> <code>bool</code> <code>break</code> <code>case</code> <code>catch</code> <code>char</code> <code>char16_t (C++11)</code> <code>char32_t (C++11)</code> <code>class</code> <code>compl</code> <code>const</code> <code>constexpr (C++11)</code> <code>const_cast</code> <code>continue</code>	<code>decltype (C++11)</code> <code>default</code> <code>delete</code> <code>do</code> <code>double</code> <code>dynamic_cast</code> <code>else</code> <code>enum</code> <code>explicit</code> <code>export</code> <code>extern</code> <code>false</code> <code>float</code> <code>for</code> <code>friend</code> <code>goto</code> <code>if</code> <code>inline</code> <code>int</code> <code>long</code> <code>mutable</code>	<code>namespace</code> <code>new</code> <code>noexcept (C++11)</code> <code>not</code> <code>not_eq</code> <code>nullptr (C++11)</code> <code>operator</code> <code>or</code> <code>or_eq</code> <code>private</code> <code>protected</code> <code>public</code> <code>register</code> <code>reinterpret_cast</code> <code>return</code> <code>short</code> <code>signed</code> <code>sizeof</code> <code>static</code> <code>static_assert (C++11)</code> <code>static_cast</code>	<code>struct</code> <code>switch</code> <code>template</code> <code>this</code> <code>thread_local (C++11)</code> <code>throw</code> <code>true</code> <code>try</code> <code>typedef</code> <code>typeid</code> <code>typename</code> <code>union</code> <code>unsigned</code> <code>using</code> <code>virtual</code> <code>void</code> <code>volatile</code> <code>wchar_t</code> <code>while</code> <code>xor</code> <code>xor_eq</code>
--	---	---	--

# Tipos de dados fundamentais

- **char**: representa um caractere. Um char requer exatamente 1 byte de espaço de memória e varia de -128 a 127.

# Tipos de dados fundamentais

- **char**: representa um caractere. Um char requer exatamente 1 byte de espaço de memória e varia de -128 a 127.
- **int**: tipo de dado que armazena um inteiro. Inteiros normalmente requerem 4 bytes de espaço de memória e variam de -2.147.483.648 a 2.147.483.647.

# Tipos de dados fundamentais

- **char**: representa um caractere. Um char requer exatamente 1 byte de espaço de memória e varia de -128 a 127.
- **int**: tipo de dado que armazena um inteiro. Inteiros normalmente requerem 4 bytes de espaço de memória e variam de -2.147.483.648 a 2.147.483.647.
- **long**: armazena um inteiro e requer pelo menos 8 bytes de espaço de memória. Varia de -9.223.372.036.854.775.808 a 9,223.372.036.854.775.807.



# Tipos de dados fundamentais

- **char**: representa um caractere. Um char requer exatamente 1 byte de espaço de memória e varia de -128 a 127.
- **int**: tipo de dado que armazena um inteiro. Inteiros normalmente requerem 4 bytes de espaço de memória e variam de -2.147.483.648 a 2.147.483.647.
- **long**: armazena um inteiro e requer pelo menos 8 bytes de espaço de memória. Varia de -9.223.372.036.854.775.808 a 9,223.372.036.854.775.807.
- **bool**: representa os valores booleanos **true** e **false**. Ocupa 1 byte de memória.

# Tipos de dados fundamentais

- **char**: representa um caractere. Um char requer exatamente 1 byte de espaço de memória e varia de -128 a 127.
- **int**: tipo de dado que armazena um inteiro. Inteiros normalmente requerem 4 bytes de espaço de memória e variam de -2.147.483.648 a 2.147.483.647.
- **long**: armazena um inteiro e requer pelo menos 8 bytes de espaço de memória. Varia de -9.223.372.036.854.775.808 a 9,223.372.036.854.775.807.
- **bool**: representa os valores booleanos **true** e **false**. Ocupa 1 byte de memória.
- **float**: representa valores de ponto flutuante. Requer pelo menos 4 bytes de espaço de memória.

# Tipos de dados fundamentais

- **char**: representa um caractere. Um char requer exatamente 1 byte de espaço de memória e varia de -128 a 127.
- **int**: tipo de dado que armazena um inteiro. Inteiros normalmente requerem 4 bytes de espaço de memória e variam de -2.147.483.648 a 2.147.483.647.
- **long**: armazena um inteiro e requer pelo menos 8 bytes de espaço de memória. Varia de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807.
- **bool**: representa os valores booleanos **true** e **false**. Ocupa 1 byte de memória.
- **float**: representa valores de ponto flutuante. Requer pelo menos 4 bytes de espaço de memória.
- **double**: representa valores de ponto flutuante de precisão dupla. Requer pelo menos 8 bytes de espaço de memória.

# Tipos de dados fundamentais

**Modificadores de Tipo de Dados:** são usados para modificar o intervalo de valores que um tipo de dado fundamental pode suportar.

# Tipos de dados fundamentais

**Modificadores de Tipo de Dados:** são usados para modificar o intervalo de valores que um tipo de dado fundamental pode suportar. Os modificadores de tipo de dados disponíveis em C++ são:

# Tipos de dados fundamentais

**Modificadores de Tipo de Dados:** são usados para modificar o intervalo de valores que um tipo de dado fundamental pode suportar. Os modificadores de tipo de dados disponíveis em C++ são:

- `unsigned` (pode ser combinado com tipos inteiros)

# Tipos de dados fundamentais

**Modificadores de Tipo de Dados:** são usados para modificar o intervalo de valores que um tipo de dado fundamental pode suportar. Os modificadores de tipo de dados disponíveis em C++ são:

- `unsigned` (pode ser combinado com tipos inteiros)
- `short` (pode ser combinado com `int`)

# Tipos de dados fundamentais

**Modificadores de Tipo de Dados:** são usados para modificar o intervalo de valores que um tipo de dado fundamental pode suportar. Os modificadores de tipo de dados disponíveis em C++ são:

- **unsigned** (pode ser combinado com tipos inteiros)
- **short** (pode ser combinado com `int`)
- **long** (pode ser combinado com `int` e `double`)



# Operador sizeof

Podemos exibir o tamanho de todos os tipos de dados usando o operador `sizeof()`:

```
1 #include <iostream> // prog05.cpp
2 using namespace std;
3
4 int main() {
5     cout << "char: " << sizeof(char) << '\n';
6     cout << "bool: " << sizeof(bool) << '\n';
7     cout << "short: " << sizeof(short) << '\n';
8     cout << "short int: " << sizeof(short int) << '\n';
9     cout << "int: " << sizeof(int) << '\n';
10    cout << "long int: " << sizeof(long int) << '\n';
11    cout << "long: " << sizeof(long) << '\n';
12    cout << "long long: " << sizeof(long long) << '\n';
13    cout << "float: " << sizeof(float) << '\n';
14    cout << "double: " << sizeof(double) << '\n';
15    cout << "long double: " << sizeof(long double) << '\n';
16 }
```

# Conversão de tipos em C++

- **Conversão de tipo** é o processo de conversão de um valor de um tipo de dados para outro tipo.
- Há dois modos de se converter tipos em C++:

# Conversão de tipos em C++

- **Conversão de tipo** é o processo de conversão de um valor de um tipo de dados para outro tipo.
- Há dois modos de se converter tipos em C++:
  - **Conversão Implícita ou Coerção**: em que o compilador transforma automaticamente um tipo de dados fundamental em outro.
  - **Conversão Explícita (Casting)**: em que o programador usa um operador de *casting* para direcionar a conversão.

# Conversão de tipos Implícita

Existem dois tipos de conversão implícita:

- **Promoção Numérica:** Quando um valor de um tipo é implicitamente convertido para um tipo semelhante **maior**.

char → short → int → long → float → double

Promoções **não resultam** em perda de dados.

# Conversão de tipos Implícita

Existem dois tipos de conversão implícita:

- **Promoção Numérica:** Quando um valor de um tipo é implicitamente convertido para um tipo semelhante **maior**.

$\text{char} \rightarrow \text{short} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{float} \rightarrow \text{double}$

Promoções **não resultam** em perda de dados.

- **Rebaixamento Numérico:** Quando um valor de um tipo é implicitamente convertido para um tipo semelhante **menor**.

$\text{char} \leftarrow \text{short} \leftarrow \text{int} \leftarrow \text{long} \leftarrow \text{float} \leftarrow \text{double}$

Rebaixamentos numéricos **resultam** em perda de dados.

# Conversão Implícita — Exemplo

```
1 #include <iostream> // prog29.cpp
2 #include <iomanip>
3 using namespace std;
4
5 int main() {
6     // Promocao numerica
7     int v = 30;
8     float f = v;
9     cout << f/7 << '\n'; // Imprime o valor 4.28571
10    cout << static_cast<float>(v) / 7 << '\n'; // Imprime o
        valor 4
```

# Conversão Implícita — Exemplo

```
1 #include <iostream> // prog29.cpp
2 #include <iomanip>
3 using namespace std;
4
5 int main() {
6     // Promocao numerica
7     int v = 30;
8     float f = v;
9     cout << f/7 << '\n'; // Imprime o valor 4.28571
10    cout << static_cast<float>(v) / 7 << '\n'; // Imprime o
        valor 4
11
12    // Rebaixamento Numerico
13    int a = 35.0/4.0;
14
15    cout << a << '\n'; // Imprime o valor 8
16
17    return 0;
18 }
```

## Conversão de tipos Explícita (casting)

- No exemplo abaixo, gostaríamos que a variável `f` recebesse o valor 2.5. No entanto, ela recebe o valor 2.

```
1 int i1 = 10;  
2 int i2 = 4;  
3 float f = i1 / i2;
```



## Conversão de tipos Explícita (casting)

- No exemplo abaixo, gostaríamos que a variável `f` recebesse o valor 2.5. No entanto, ela recebe o valor 2.

```
1 int i1 = 10;  
2 int i2 = 4;  
3 float f = i1 / i2;
```

- Para avisarmos o compilador que queremos usar a divisão de ponto flutuante em vez da divisão de números inteiros, usamos um operador de conversão de tipos (cast).

## Conversão de tipos Explícita (casting)

- No exemplo abaixo, gostaríamos que a variável `f` recebesse o valor 2.5. No entanto, ela recebe o valor 2.

```
1 int i1 = 10;  
2 int i2 = 4;  
3 float f = i1 / i2;
```

- Para avisarmos o compilador que queremos usar a divisão de ponto flutuante em vez da divisão de números inteiros, usamos um operador de conversão de tipos (`cast`).
- C++ fornece algumas formas de realizar casting explícito. Aqui, precisamos do operador `static_cast`.

# Conversão explícita usando `static_cast`

- C++ possui um operador de conversão chamado `static_cast`.

# Conversão explícita usando `static_cast`

- C++ possui um operador de conversão chamado `static_cast`.
- `static_cast` aceita um único valor como entrada e gera o mesmo valor convertido para o tipo especificado dentro dos colchetes angulares.

```
1 int i1 = 10;  
2 int i2 = 4;  
3 float f = static_cast<float>(i1) / i2;
```

# Constantes

Uma **constante** é uma variável especial que permite guardar determinado dado na memória do computador, com a certeza de que ele não se alterará durante a execução do programa.

# Constantes

Uma **constante** é uma variável especial que permite guardar determinado dado na memória do computador, com a certeza de que ele não se alterará durante a execução do programa.

A fim de declarar uma constante, basta colocar a palavra-chave **const** antes do tipo de variável:

# Constantes

Uma **constante** é uma variável especial que permite guardar determinado dado na memória do computador, com a certeza de que ele não se alterará durante a execução do programa.

A fim de declarar uma constante, basta colocar a palavra-chave **const** antes do tipo de variável:

```
1 #include <iostream> // prog04.cpp
2
3 int main() {
4     const double gravidade {9.7}; // declarando constante
5     std::cout << gravidade << '\n';
6
7     std::cout << "Digite sua idade: ";
8     int idade;
9     std::cin >> idade;
10
11     const int idadeUsuario { idade }; // declarando constante
12     std::cout << idadeUsuario << '\n';
13     return 0;
14 }
```

# Constantes

O C++ suporta dois tipos de constantes:

- **Constantes em tempo de compilação:** são aquelas cujos valores de inicialização podem ser resolvidos em tempo de compilação.  
Exemplo: **gravidade**, do exemplo anterior.



# Constantes

O C++ suporta dois tipos de constantes:

- **Constantes em tempo de compilação:** são aquelas cujos valores de inicialização podem ser resolvidos em tempo de compilação.  
Exemplo: **gravidade**, do exemplo anterior.
  - Constantes deste tipo permitem que o compilador realize otimizações.

# Constantes

O C++ suporta dois tipos de constantes:

- **Constantes em tempo de compilação:** são aquelas cujos valores de inicialização podem ser resolvidos em tempo de compilação.  
Exemplo: `gravidade`, do exemplo anterior.
  - Constantes deste tipo permitem que o compilador realize otimizações.
- **Constantes em tempo de execução:** são aquelas cujos valores de inicialização só podem ser resolvidos em tempo de execução.  
Exemplo: `idadeUsuario`, do exemplo anterior.

# Namespaces



# Namespaces

- Um **namespace** é uma região declarativa que fornece um escopo para os identificadores (os nomes de tipos, funções, variáveis, etc) dentro dele.

# Namespaces

- Um **namespace** é uma região declarativa que fornece um escopo para os identificadores (os nomes de tipos, funções, variáveis, etc) dentro dele.
- São usados para organizar o código em grupos lógicos a fim de evitar colisões de nomes que podem ocorrer especialmente quando sua base de código inclui várias bibliotecas.

# Namespaces

- Um **namespace** é uma região declarativa que fornece um escopo para os identificadores (os nomes de tipos, funções, variáveis, etc) dentro dele.
- São usados para organizar o código em grupos lógicos a fim de evitar colisões de nomes que podem ocorrer especialmente quando sua base de código inclui várias bibliotecas.
- Exemplo: namespace **std** (standard)

```
1 #include <iostream> // prog60.cpp
2
3 int main() {
4     std::cout << "Hello world\n";
5     return 0;
6 }
```

# Namespaces

- Um **namespace** é uma região declarativa que fornece um escopo para os identificadores (os nomes de tipos, funções, variáveis, etc) dentro dele.
- São usados para organizar o código em grupos lógicos a fim de evitar colisões de nomes que podem ocorrer especialmente quando sua base de código inclui várias bibliotecas.
- Exemplo: namespace **std** (standard)

```
1 #include <iostream> // prog60.cpp
2
3 int main() {
4     std::cout << "Hello world\n";
5     return 0;
6 }
```

- O símbolo **::** é chamado **operador de resolução de escopo**.

# Namespaces

- Os identificadores fora do namespace podem acessar os membros das seguintes formas:



# Namespaces

- Os identificadores fora do namespace podem acessar os membros das seguintes formas:
  - usando o nome totalmente qualificado para cada identificador.

**Exemplo:** `std::cout`

# Namespaces

- Os identificadores fora do namespace podem acessar os membros das seguintes formas:
  - usando o nome totalmente qualificado para cada identificador.  
**Exemplo:** `std::cout`
  - por meio de uma declaração `using` para um único identificador.  
**Exemplo:** `using std::cout;`

# Namespaces

- Os identificadores fora do namespace podem acessar os membros das seguintes formas:
  - usando o nome totalmente qualificado para cada identificador.  
**Exemplo:** `std::cout`
  - por meio de uma declaração `using` para um único identificador.  
**Exemplo:** `using std::cout;`
  - por meio de uma diretiva `using` para todos os identificadores no namespace.  
**Exemplo:** `using namespace std;`

# Namespaces

- Os identificadores fora do namespace podem acessar os membros das seguintes formas:
  - usando o nome totalmente qualificado para cada identificador.  
**Exemplo:** `std::cout`
  - por meio de uma declaração **using** para um único identificador.  
**Exemplo:** `using std::cout;`
  - por meio de uma diretiva **using** para todos os identificadores no namespace.  
**Exemplo:** `using namespace std;`
- **Boa prática:** Use os prefixos de namespace **explicitamente** a fim de acessar os identificadores definidos no namespace.

# Definindo seu próprio namespace

- A palavra-chave `namespace` é usada para declarar um escopo que contém um conjunto de objetos relacionados.

```
1 // Arquivo mymath.hpp
2
3 namespace math {
4     int sum(int x, int y) { return x+y; }
5     int sub(int x, int y) { return x-y; }
6     int mul(int x, int y) { return x*y; }
7     int div(int x, int y) { return x/y; }
8 }
```

## Definindo seu próprio namespace (Cont.)

```
1 #include <iostream> //prog61.cpp
2 #include "mymath.hpp"
3
4 using namespace math;
5
6 int main() {
7     int a{ 5 }, b { 10 };
8     std::cout << math::sum(a,b) << '\n';
9     std::cout << math::sub(a,b) << '\n';
10    std::cout << math::mul(a,b) << '\n';
11    std::cout << math::div(a,b) << '\n';
12    return 0;
13 }
```

# Declarações using – Exemplo

```
1 #include <iostream> //prog63.cpp
2
3 namespace math {
4     int sum(int x, int y) { return x+y; }
5     int sub(int x, int y) { return x-y; }
6     int mul(int x, int y) { return x*y; }
7     int div(int x, int y) { return x/y; }
8 }
9
10 int main() {
11     using std::cout; // using declaration
12     using std::endl; // using declaration
13
14     int a{ 5 }, b{ 4 };
15     cout << math::sum(a,b) << endl;
16     cout << math::sub(a,b) << endl;
17     cout << math::mul(a,b) << endl;
18     cout << math::div(a,b) << endl;
19     return 0;
20 }
```

# Diretiva using

```
1 #include <iostream> //prog64.cpp
2 using std::cout;
3 using std::endl;
4
5 namespace math {
6     int sum(int x, int y) { return x+y; }
7     int sub(int x, int y) { return x-y; }
8     int mul(int x, int y) { return x*y; }
9     int div(int x, int y) { return x/y; }
10 }
11
12 int main() {
13     using namespace math;
14     int a{ 5 }, b{ 4 };
15     cout << sum(a,b) << endl;
16     cout << sub(a,b) << endl;
17     cout << mul(a,b) << endl;
18     cout << div(a,b) << endl; // Erro de compilacao
19     return 0;
20 }
```

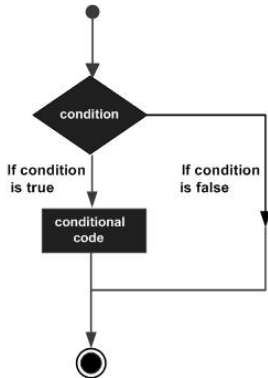
- `div_t div(int numer, int denom);` retorna um struct e está definido sob o namespace `std`



## Estruturas de Seleção

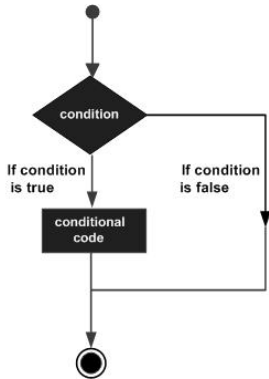


# Estruturas de Seleção — If.. else



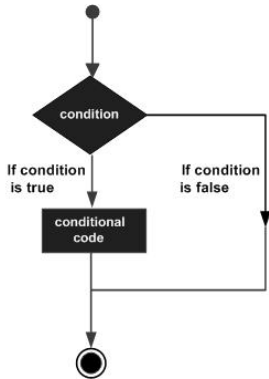
```
1 int c;  
2 std::cin >> c;  
3  
4 if (c == 1) { // If  
5     std::cout << "igual a 1";  
6 }
```

# Estruturas de Seleção — If.. else



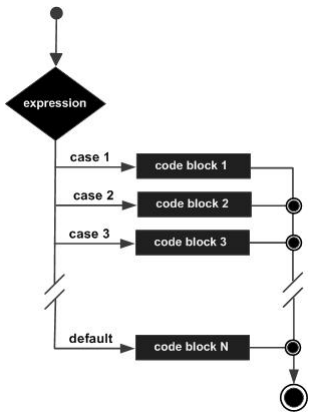
```
1 int c;  
2 std::cin >> c;  
3  
4 if (c == 1) { // If  
5     std::cout << "igual a 1";  
6 }  
7  
8 if (c == 2) { // If .. else  
9     std::cout << "igual a 2";  
10 } else {  
11     std::cout << "nao eh 1 nem 2";  
12 }
```

# Estruturas de Seleção — If.. else



```
1 int c;
2 std::cin >> c;
3
4 if (c == 1) { // If
5     std::cout << "igual a 1";
6 }
7
8 if (c == 2) { // If .. else
9     std::cout << "igual a 2";
10 } else {
11     std::cout << "nao eh 1 nem 2";
12 }
13
14 if (c == 3) { // If else encadeado
15     std::cout << "igual a 3";
16 } else if (c == 4) {
17     std::cout << "igual a 4";
18 } else {
19     std::cout << "nao eh 1,2,3,4";
20 }
```

# Estruturas de Seleção — Switch



```
1  int c;  
2  std::cin >> c;  
3  
4  switch (c) // int, char, short, long  
5  {  
6      case 1:  
7          std::cout << 1 << '\n';  
8          break;  
9      case 2:  
10         std::cout << 2 << '\n';  
11         break;  
12     case 3:  
13         std::cout << 3 << '\n';  
14         break;  
15     case 4:  
16         std::cout << 4 << '\n';  
17         break;  
18     default:  
19         std::cout << 5 << '\n';  
20         break;  
21 }
```

# Alternativa para o if-else

## Operador ternário

O **operador ternário** avalia uma condição de teste e executa um bloco de código com base no resultado da condição.

Sua sintaxe é:

```
condition ? expression1 : expression2 ;
```

- se **condition** for **true** então **expression1** é executado;
- se **condition** for **false** então **expression2** é executado.

# Operador ternário – Exemplo

```
1 #include <iostream> // prog150.cpp
2 #include <string>
3 using namespace std;
4
5 int main() {
6     double nota;
7
8     cout << "Digite sua nota: ";
9     cin >> nota;
10
11     string result = (nota >= 7) ? "passou" : "reprovou";
12
13     cout << "Voce " << result << " no exame.\n";
14
15     return 0;
16 }
```

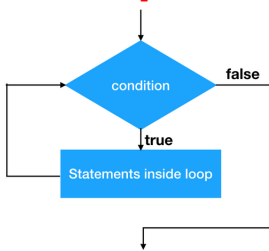
# Estruturas de Repetição





# Estruturas de repetição (loops)

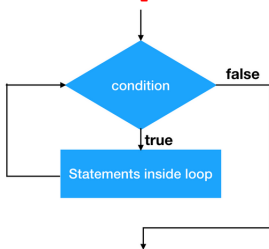
## while loop



```
1 int contador = 0;
2
3 while (contador < 10) {
4     std::cout << contador << " ";
5     contador++;
6 }
```

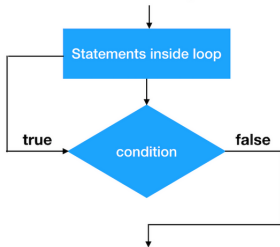
# Estruturas de repetição (loops)

## while loop



```
1 int contador = 0;
2
3 while (contador < 10) {
4     std::cout << contador << " ";
5     contador++;
6 }
```

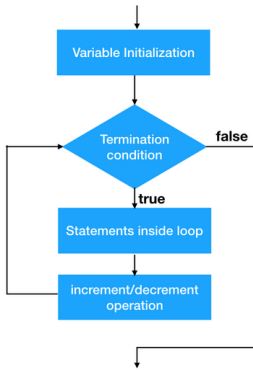
## do..while loop



```
1 do {
2     std::cout << contador << " ";
3     contador++;
4 } while (contador < 10);
```

# Estruturas de repetição (loops)

## for loop



```
1 for (int i = 0; i < 10; i++)  
2     std::cout << i << " ";
```

# Entrada/Saída em C++

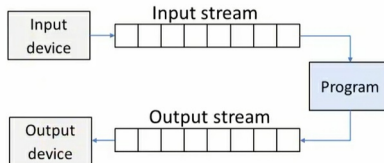


# Streams

- Um **stream** (fluxo) é uma sequência de bytes que podem ser acessados sequencialmente.
  - Em C++, Entrada e Saída são implementados com streams.

# Streams

- Um **stream** (fluxo) é uma sequência de bytes que podem ser acessados sequencialmente.
  - Em C++, Entrada e Saída são implementados com streams.
- Existem dois tipos de streams:
  - **Input Stream**: usadas para gerenciar dados de entrada vindos de um teclado, arquivo, rede, etc.
    - **Dispositivo de entrada padrão**: teclado
  - **Output Stream**: usadas para gerenciar dados de saída enviados para um terminal, arquivo, impressora, etc.
    - **Dispositivo de saída padrão**: terminal



# Streams

- Cabeçalho responsável por I/O: `iostream`.
  - Esse cabeçalho define os tipos: `istream` (input stream) e `ostream` (output stream), assim como dois objetos, chamados `std::cin` (common input) e `std::cout` (common output).

# Streams

- Cabeçalho responsável por I/O: `iostream`.
  - Esse cabeçalho define os tipos: `istream` (input stream) e `ostream` (output stream), assim como dois objetos, chamados `std::cin` (common input) e `std::cout` (common output).
- Os objetos `std::cin` e `std::cout` são usados em conjunto com os operadores `>>` (operador de extração) e `<<` (operador de inserção), respectivamente.



# Streams

- Cabeçalho responsável por I/O: `iostream`.
  - Esse cabeçalho define os tipos: `istream` (input stream) e `ostream` (output stream), assim como dois objetos, chamados `std::cin` (common input) e `std::cout` (common output).
- Os objetos `std::cin` e `std::cout` são usados em conjunto com os operadores `>>` (operador de extração) e `<<` (operador de inserção), respectivamente.

- O formato geral para ler dados do teclado é:

`std::cin >> variavel1 >> variavel2 >> ... >> variavelN`

- O formato geral para mostrar dados na tela é:

`std::cout << item1 << item2 << ... << itemN`

- Os dados na saída podem ser variáveis ou constantes literais de qualquer tipo nativo.

# Formatação da Saída

- O cabeçalho `iomanip` fornece um conjunto de funções chamadas `manipuladores`, que podem ser usadas para formatar a saída.
- Os manipuladores podem ser usados como uma cadeia em uma declaração:

```
std::cout << manip1 << manip2 << manip3 << item;
```

# Alguns manipuladores e o seu significado

## Números inteiros e manipulação de base

Manipulador	Significado
<code>std::dec</code>	usa a base decimal
<code>std::oct</code>	usa a base octal
<code>std::hex</code>	usa a base hexadecimal
<code>std::setbase(int base)</code>	seta a base para 8, 10 ou 16
<code>std::showbase</code>	Faz com que a base do número seja escrita antes do número: 0 para octal, 0x para hexadecimal. Resetado com <code>noshowbase</code> .

# Alguns manipuladores e o seu significado

## Números de ponto-flutuante

Manipulador	Significado
<code>std::scientific</code>	usa notação científica
<code>std::fixed</code>	usa notação de ponto fixo
<code>std::setprecision(int d)</code>	seta o número de casas decimais para d
<code>std::showpoint</code>	Força números de ponto flutuante serem impressos com ponto. Geralmente usado com <code>fixed</code> para garantir um certo número de casas decimais, mesmo que sejam zeros. Resetado com <code>noshowpoint</code> .
<code>std::showpos</code>	Faz com que números positivos sejam precedidos pelo sinal +. Resetado com <code>noshowpos</code> .

# Exemplo 1

```
1 // Controlling precision of floating-pointing values
2 #include <iostream> // manipulators1.cpp
3 #include <iomanip>
4 #include <cmath>
5 using namespace std;
6
7 int main() {
8     double root2{sqrt(2.0)}; // calculate square root of 2
9     cout << "sqrt(2) = " << root2 << '\n';
10    cout << scientific << "sqrt(2) = " << root2 << '\n';
11
12    cout << "Square root of 2 with precisions 0-9.\n";
13    cout << fixed; // use fixed point notation
14
15    // set precision for each digit, then display square root
16    for(int places{0}; places <= 9; ++places) {
17        cout << setprecision(places) << root2 << "\n";
18    }
19 }
```

## Exemplo 2

```
1 // Controlling precision of floating-pointing values
2 #include <iostream> // manipulators2.cpp
3 #include <iomanip>
4 using namespace std;
5
6 int main() {
7     double number1{4.0 / 2.0};
8     double number2{4.0 / 3.0};
9
10    cout << showpos; // show plus sign on positive numbers
11
12    cout << number1 << "\n"
13         << number2 << "\n\n" << flush;
14
15    cout << fixed << setprecision(2);
16
17    cout << number1 << "\n"
18         << number2 << endl;
19 }
```

# Outros manipuladores

Manipulador	Significado
<code>std::setw(int n)</code>	seta a largura do campo para n
<code>std::setfill(char c)</code>	faz o caractere de preenchimento ser o c
<code>std::left</code>	Ajusta a saída para a esquerda
<code>std::right</code>	Ajusta a saída para a direita (padrão)
<code>std::boolalpha</code>	Faz com que valores booleanos sejam impressos como as palavras <code>true</code> ou <code>false</code> . Resetado pelo manipulador <code>noboolalpha</code> .
<code>std::endl</code>	Insere uma nova linha e chama <code>std::flush</code>

## Exemplo 3

```
1 #include <iostream> // manipulators3.cpp
2 #include <iomanip>
3 using namespace std;
4
5 int main() {
6     int number {12345};
7
8     cout << setfill('*'); // sticky manipulator
9
10    cout << "\"" << left
11         << setw(10)
12         << number << "\"" << endl;
13
14    cout << "\"" << number << "\"" << endl;
15
16    cout << "\"" << right
17         << setw(10)
18         << number << "\"" << endl;
19 }
```



# Estados de erro de uma Stream

- Todo objeto stream contém um conjunto de variáveis de erro que representam o estado de uma stream.

# Estados de erro de uma Stream

- Todo objeto stream contém um conjunto de variáveis de erro que representam o estado de uma stream.
- Por exemplo, a stream `std::cin` contém as seguintes variáveis:
  - `failbit`: possui valor `1` se e somente se o tipo de dado da entrada estiver errado.
  - `eofbit`: possui valor `1` se e somente se tiver chegado ao final da leitura.
  - `badbit`: possui valor `1` se e somente se a operação falhar de modo irreversível. Exemplo: falha de disco no momento da leitura.

# Estados de erro de uma Stream

- Todo objeto stream contém um conjunto de variáveis de erro que representam o estado de uma stream.
- Por exemplo, a stream `std::cin` contém as seguintes variáveis:
  - `failbit`: possui valor `1` se e somente se o tipo de dado da entrada estiver errado.
  - `eofbit`: possui valor `1` se e somente se tiver chegado ao final da leitura.
  - `badbit`: possui valor `1` se e somente se a operação falhar de modo irreversível. Exemplo: falha de disco no momento da leitura.
- `std::cin` possui as funções `fail()`, `eof()` e `bad()`, que devolvem o valor dessas variáveis.

## Estados de erro de uma Stream

- `std::cin` possui a função `good()` que retorna `1` quando todas as três funções `fail()`, `eof()` e `bad()` retornam `0`.

# Estados de erro de uma Stream

- `std::cin` possui a função `good()` que retorna `1` quando todas as três funções `fail()`, `eof()` e `bad()` retornam `0`.
- `std::cin` possui a função `clear()`, que limpa o estado de erro da stream `std::cin`, setando o valor de `goodbit` para `1` e os valores dos demais bits para `0`.
  - Deste modo, usando a função `clear()`, é possível continuar usando a stream, mesmo depois de ocorrer um erro.

## std::cin — Extração e descarte de caracteres

O objeto `std::cin` possui a função-membro `ignore`

```
istream& ignore (int n = 1, int delim = EOF)
```

- Essa função extrai caracteres da sequência de entrada e os descarta, até que `n` caracteres tenham sido extraídos ou um caractere igual a `delim` tenha sido lido.

## std::cin — Extração e descarte de caracteres

O objeto `std::cin` possui a função-membro `ignore`

```
istream& ignore (int n = 1, int delim = EOF)
```

- Essa função extrai caracteres da sequência de entrada e os descarta, até que `n` caracteres tenham sido extraídos ou um caractere igual a `delim` tenha sido lido.
- Como primeiro argumento dessa função, geralmente se usa:  
`std::numeric_limits<std::streamsize>::max()`

## std::cin — Extração e descarte de caracteres

O objeto `std::cin` possui a função-membro `ignore`

```
istream& ignore (int n = 1, int delim = EOF)
```

- Essa função extrai caracteres da sequência de entrada e os descarta, até que `n` caracteres tenham sido extraídos ou um caractere igual a `delim` tenha sido lido.
- Como primeiro argumento dessa função, geralmente se usa:  
`std::numeric_limits<std::streamsize>::max()`
- Como segundo argumento dessa função, geralmente se usa: `'\\n'`



# Exemplo

```
1 #include <iostream> //errorTreatment.cpp
2 #include <limits>
3
4 void ignore_line() {
5     std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
6         '\n');
7 }
8
9 int readInt() {
10     while(true) {
11         int value;
12         std::cin >> value;
13         if(std::cin.fail()) {
14             std::cerr << "fail: enter a valid integer\n";
15             std::cin.clear();
16             ignore_line();
17         }
18         else {
19             ignore_line();
20             return value;
21         }
22     }
```

## Exemplo (cont.)

```
24 {  
25     int x = readInt();  
26     std::cout << "x = " << x << '\n';  
27 }
```

# Vetores



# Vetores (Arrays)

- Um **vetor (array)** é uma estrutura de dados que nos permite acessar muitas variáveis do mesmo tipo por meio de um único identificador.
- Em quais casos na programação pode ser necessário o uso de um vetor?

# Declarando e inicializando vetores

<code>int numbers[10];</code>	Um array de 10 inteiros.
<code>const int SIZE = 10; int numbers[SIZE];</code>	É uma boa ideia usar uma variável constante para o tamanho.
<code>int size = 10; int numbers[size];</code>	<b>Atenção:</b> Em C++ padrão, o tamanho do vetor deve ser uma constante. Esta definição de vetor não funcionará em todos os compiladores.
<code>int vec[5] {0,1,4,9,16};</code>	Um vetor de cinco inteiros, inicializado com "brace initialization"
<code>int vec[] {0,1,4,9,16};</code>	O tamanho do vetor pode ser omitido neste caso, pois ele é definido pelo número de valores iniciais.
<code>int squares[5] = {0,1,4};</code>	Se você fornecer menos valores iniciais que o tamanho, os valores restantes serão definidos como 0. Esse array contém 0, 1, 4, 0, 0.

## Tamanho de um vetor

Se estivermos no mesmo escopo em que um array foi definido, podemos determinar seu tamanho usando o operador `sizeof`.

Exemplo:

# Tamanho de um vetor

Se estivermos no mesmo escopo em que um array foi definido, podemos determinar seu tamanho usando o operador `sizeof`.

## Exemplo:

```
1 #include <iostream> // prog16.cpp
2
3 int main() {
4     int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
5
6     std::cout << "Tamanho do vetor: ";
7     std::cout << sizeof(array) / sizeof(array[0]) << "\n";
8
9     return 0;
10 }
```

- **Obs.:** Só funcionará se `sizeof` estiver na mesma função na qual o vetor estiver declarado.

# Vetores e laços for-each

- O C++11 introduziu o **loop for-each**, que fornece um método mais simples para iterar sobre os elementos de um vetor.

```
1 #include <iostream> // prog31.cpp
2
3 int main() {
4     int fibonacci[] = {0,1,1,2,3,5,8,13};
5
6     for (int valor : fibonacci) {
7         std::cout << valor << std::endl;
8     }
9
10    return 0;
11 }
```



# Vetores e laços for-each

- O C++11 introduziu o **loop for-each**, que fornece um método mais simples para iterar sobre os elementos de um vetor.

```
1 #include <iostream> // prog31.cpp
2
3 int main() {
4     int fibonacci[] = {0,1,1,2,3,5,8,13};
5
6     for (int valor : fibonacci) {
7         std::cout << valor << std::endl;
8     }
9
10    return 0;
11 }
```

- **Atenção:** o vetor e o loop devem estar no mesmo escopo.

## Palavra-chave auto

- A partir do C++11, quando uma variável é declarada seguida de inicialização, podemos usar a palavra-chave `auto` no lugar do tipo da variável.

## Palavra-chave auto

- A partir do C++11, quando uma variável é declarada seguida de inicialização, podemos usar a palavra-chave `auto` no lugar do tipo da variável.
  - Com isso, o tipo da variável será inferido da expressão de inicialização.

## Palavra-chave auto

- A partir do C++11, quando uma variável é declarada seguida de inicialização, podemos usar a palavra-chave **auto** no lugar do tipo da variável.
  - Com isso, o tipo da variável será inferido da expressão de inicialização.
- Exemplo (equivalente ao do slide anterior):

```
1 #include <iostream> // prog32.cpp
2
3 int main() {
4     int fibonacci[] = {0,1,1,2,3,5,8,13};
5
6     for (auto valor : fibonacci) {
7         std::cout << valor << std::endl;
8     }
9
10    return 0;
11 }
```

## Palavra-chave auto

- A partir do C++11, quando uma variável é declarada seguida de inicialização, podemos usar a palavra-chave **auto** no lugar do tipo da variável.
  - Com isso, o tipo da variável será inferido da expressão de inicialização.
- Exemplo (equivalente ao do slide anterior):

```
1 #include <iostream> // prog32.cpp
2
3 int main() {
4     int fibonacci[] = {0,1,1,2,3,5,8,13};
5
6     for (auto valor : fibonacci) {
7         std::cout << valor << std::endl;
8     }
9
10    return 0;
11 }
```

- Qual vantagem de se usar a palavra-chave auto?

# Arrays de caracteres – Strings do C



# Arrays de caracteres — Strings do C

- Na linguagem C, uma **string** é uma sequência de caracteres.

# Arrays de caracteres — Strings do C

- Na linguagem C, uma **string** é uma sequência de caracteres.
- C++ suporta dois tipos diferentes de string:



# Arrays de caracteres — Strings do C

- Na linguagem C, uma **string** é uma sequência de caracteres.
- C++ suporta dois tipos diferentes de string:
  - (1) strings como array de caracteres terminados em `'\0'` (**C-style strings**).

Exemplo:

```
1 char palavra[20] = "abracadabra";  
2 char minhaString[] = "string";
```

# Arrays de caracteres — Strings do C

- Na linguagem C, uma **string** é uma sequência de caracteres.
- C++ suporta dois tipos diferentes de string:
  - (1) strings como array de caracteres terminados em `'\0'` (**C-style strings**).

Exemplo:

```
1 char palavra[20] = "abracadabra";  
2 char minhaString[] = "string";
```

- (2) **std::string** (como parte da biblioteca padrão). Exemplo:

```
1 std::string palavra;  
2 palavra = "abracadabra"; // Atribuicao funciona aqui
```

## C-style Strings — Características

- **Inicialização de strings:** C-style strings seguem as mesmas regras que os arrays: você pode inicializar a string no momento da criação, mas você não pode atribuir valores a ela usando o operador de atribuição depois disso.

## C-style Strings — Características

- **Inicialização de strings:** C-style strings seguem as mesmas regras que os arrays: você pode inicializar a string no momento da criação, mas você não pode atribuir valores a ela usando o operador de atribuição depois disso.

```
1 #include <iostream> // prog18.cpp
2
3 int main() {
4     char str1[20] = "Oi gente"; // Ok!
5     char str2[20];
6
7     str2 = str1; // ERRADO!
8     str2 = "SOL"; // ERRADO!
9     str2[0] = 'S';
10    str2[1] = 'O';
11    str2[2] = 'L';
12    str2[3] = '\0';
13
14    return 0;
15 }
```

## C-style Strings — Lendo e Imprimindo

- **Imprimindo com `std::cout`:** `std::cout` imprime caracteres até encontrar o terminador nulo `'\0'`.

## C-style Strings — Lendo e Imprimindo

- **Imprimindo com `std::cout`:** `std::cout` imprime caracteres até encontrar o terminador nulo `'\0'`.
- **Lendo strings:** uma forma segura de ler strings do teclado é usar o comando

```
std::cin.get(char* s, streamsize n),
```

onde `s` é um array de caracteres e `n-1` é o número máximo de caracteres que será lido e armazenado em `s`.

## C-style Strings — Lendo e Imprimindo

- **Imprimindo com `std::cout`:** `std::cout` imprime caracteres até encontrar o terminador nulo `'\0'`.
- **Lendo strings:** uma forma segura de ler strings do teclado é usar o comando  
`std::cin.get(char* s, streamsize n),`  
onde `s` é um array de caracteres e `n-1` é o número máximo de caracteres que será lido e armazenado em `s`.
- Exemplo:

## C-style Strings — Lendo e Imprimindo

- **Imprimindo com `std::cout`:** `std::cout` imprime caracteres até encontrar o terminador nulo `'\0'`.
- **Lendo strings:** uma forma segura de ler strings do teclado é usar o comando

`std::cin.get(char* s, streamsize n),`

onde `s` é um array de caracteres e `n-1` é o número máximo de caracteres que será lido e armazenado em `s`.

- Exemplo:

```
1 #include <iostream> // prog19.cpp
2
3 int main() {
4     char name[10]; // declara um array de tamanho 10
5     std::cout << "Digite seu nome: ";
6     std::cin.get(name, 10);
7     std::cout << "Voce digitou: " << name << "\n";
8
9     return 0;
10 }
```



## Cuidado com `cin.get()`

- A função `std::cin.get(char* s, streamsize n)` lê no máximo  $n - 1$  caracteres do buffer de entrada. Se sobram caracteres, eles ficarão no buffer. Mesmo se não sobrar caracteres, o caractere `'\n'` não é lido e ficará no buffer.
  - Ou seja, essa função sempre deixa um caractere no buffer, o que pode impactar a próxima leitura, caso ela seja uma leitura de string.
  - Para limpar o buffer de leitura, usa-se o seguinte comando:  
`cin.ignore(numeric_limits<streamsize>::max(), '\n');`

# C-style Strings — Lendo e Imprimindo

```
1 #include <iostream> // prog65.cpp
2 #include <cstring>
3 #include <limits>
4 using namespace std;
5
6 const int MAX{ 10 };
7
8 int main() {
9     char vec[MAX];
10
11     do {
12         cout << "Digite uma string: " << endl;
13         cin.get(vec, MAX);
14         cin.ignore(numeric_limits<streamsize>::max(), '\n');
15         cout << "String digitada foi: \"" << vec << "\"" << endl;
16     } while(strcmp(vec, "0") != 0);
17
18     return 0;
19 }
```

# Manipulando C-style Strings (Biblioteca `cstring`)

Duas funções da biblioteca `cstring`:

- `char* strcpy(char *destino, const char *origem);`  
Copia a string `origem` para a string `destino`, incluindo o caractere nulo de terminação (e parando nesse ponto).  
Para evitar overflows, o tamanho do vetor `destino` deve ser longo o suficiente para conter a string `origem`.

# Manipulando C-style Strings (Biblioteca `cstring`)

Duas funções da biblioteca `cstring`:

- `char* strcpy(char *destino, const char *origem);`  
Copia a string `origem` para a string `destino`, incluindo o caractere nulo de terminação (e parando nesse ponto).  
Para evitar overflows, o tamanho do vetor `destino` deve ser longo o suficiente para conter a string `origem`.
- `size_t strlen(const char *str);`  
Retorna o número de bytes da string `str`, excluindo o caractere nulo de terminação. O valor `size_t` é qualquer inteiro sem sinal (`unsigned int`).

# Manipulando C-style Strings (Biblioteca `cstring`)

Duas funções da biblioteca `cstring`:

- `char* strcpy(char *destino, const char *origem);`  
Copia a string `origem` para a string `destino`, incluindo o caractere nulo de terminação (e parando nesse ponto).  
Para evitar overflows, o tamanho do vetor `destino` deve ser longo o suficiente para conter a string `origem`.
- `size_t strlen(const char *str);`  
Retorna o número de bytes da string `str`, excluindo o caractere nulo de terminação. O valor `size_t` é qualquer inteiro sem sinal (`unsigned int`).
- Para outras funções, consultar:  
<http://www.cplusplus.com/reference/cstring/>

## Exemplo com strlen e strcpy

```
1 #include <iostream> // prog20.cpp
2 #include <cstring>
3
4 int main () {
5     char str1[] = "oi gente";
6     char str2[40];
7     char str3[40];
8
9     strcpy(str2, str1);
10
11     strcpy(str3, "Copia bem sucedida");
12
13     std::cout << "str1: " << str1 << "\nstr2: " << str2;
14     std::cout << "\nstr3: " << str3 << "\n";
15     std::cout << "Tamanho de str1: " << strlen(str1);
16
17     return 0;
18 }
```

## Exemplo com strlen e strcpy

```
1 #include <iostream> // prog20.cpp
2 #include <cstring>
3
4 int main () {
5     char str1[] = "oi gente";
6     char str2[40];
7     char str3[40];
8
9     strcpy(str2, str1);
10
11    strcpy(str3, "Copia bem sucedida");
12
13    std::cout << "str1: " << str1 << "\nstr2: " << str2;
14    std::cout << "\nstr3: " << str3 << "\n";
15    std::cout << "Tamanho de str1: " << strlen(str1);
16
17    return 0;
18 }
```

- Para outras funções da biblioteca **cstring**, consultar:  
<http://www.cplusplus.com/reference/cstring/>

# Biblioteca ctype

<ctype>: útil ao se trabalhar com caracteres.


int isalnum(int c)	Verifica se c é um dígito decimal ou uma letra maiúscula ou minúscula.
int isalpha(int c)	Verifica se c é uma letra alfabética.
int isblank(int c)	Verifica se c é um caractere em branco (espaço ou tab)
int isdigit(int c)	Verifica se c é um caractere de dígito decimal.
int islower(int c)	Verifica se c é uma letra minúscula.
int isupper(int c)	Verifica se o parâmetro c é uma letra alfabética maiúscula.
int isspace(int c)	Verifica se c é um caractere de espaço em branco, ou seja, ' ' (space), '\t' (horizontal tab), '\n' (newline), '\v' (vertical tab), '\f' (feed), '\r' (carriage return)

<https://cplusplus.com/reference/ctype/>



## Exemplo — cctype

```
1 #include <iostream> // prog347.cpp
2 #include <cctype>
3 using namespace std;
4
5 int main() {
6     char ch;
7     cout << "Digite um caractere: ";
8
9     while(cin >> ch) {
10         cout << "O caractere '" << ch << "':";
11         if(isalnum(ch)) cout << "\neh um caractere
12         alfanumerico";
13         else cout << "\nnao eh um caractere alfanumerico";
14         if(isdigit(ch)) cout << "\neh um digito decimal";
15         else cout << "\nnao eh um digito decimal";
16         if(isspace(ch)) cout << "\neh um espaco";
17         else cout << "\nnao eh um espaco";
18         if(islower(ch)) cout << "\neh minusculo";
19         else cout << "\nnao eh minusculo";
20         if(isupper(ch)) cout << "\neh maiusculo";
21         else cout << "\nnao eh maiusculo";
22         cout << "\nDigite um caractere: ";
23     }
```



`std::string`



## O tipo de dado `std::string`

- Como strings são comumente usadas em programas, o C++ oferece um tipo de dados de string como parte da biblioteca padrão, o `std::string`.

## O tipo de dado `std::string`

- Como strings são comumente usadas em programas, o C++ oferece um tipo de dados de string como parte da biblioteca padrão, o `std::string`.
- Para usar esse tipo, primeiro precisamos incluir o cabeçalho `<string>`. Feito isso, podemos definir variáveis do tipo `std::string`.

## O tipo de dado `std::string`

- Como strings são comumente usadas em programas, o C++ oferece um tipo de dados de string como parte da biblioteca padrão, o `std::string`.
- Para usar esse tipo, primeiro precisamos incluir o cabeçalho `<string>`. Feito isso, podemos definir variáveis do tipo `std::string`.
- Exemplo:

# O tipo de dado `std::string`

- Como strings são comumente usadas em programas, o C++ oferece um tipo de dados de string como parte da biblioteca padrão, o `std::string`.
- Para usar esse tipo, primeiro precisamos incluir o cabeçalho `<string>`. Feito isso, podemos definir variáveis do tipo `std::string`.
- Exemplo:

```
1 #include <string> // prog21.cpp
2 #include <iostream>
3
4 int main() {
5     std::string myName {"Alex"};
6     std::cout << "Meu nome eh " << myName;
7
8     return 0;
9 }
```

# Inicializando uma std::string

Abaixo são mostradas diferentes formas de inicializar uma string:

```
1 #include <string> // prog345.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     string str1;           // no parameter
7     string str2 ("abracadabra"); // c-style string
8     string str3 ( str2 );  // std::string parameter
9     string str4 {'s', 'o', 'l'}; // char list parameter
10    string str5 ( 7, 'b' ); // b repeated 7 times
11    string str6 ( str2, 4 ); // "cadabra"
12    string str7 ( str2, 4, 3 ); // "cad"
13
14    cout << "str1 = " << str1 << "\nstr2 = "
15         << str2 << "\nstr3 = " << str3 << "\nstr4 = "
16         << str4 << "\nstr5 = " << str5 << "\nstr6 = "
17         << str6 << "\nstr7 = " << str7 << endl;
18 }
```

# Lendo uma `std::string` do teclado

```
1 #include <string> // prog22.cpp
2 #include <iostream>
3
4 int main() {
5     std::cout << "Digite seu nome completo: ";
6     std::string nome;
7     std::cin >> nome; /* isto nao funcionara como esperado
8                        pois std::cin quebra as palavras
9                        por espacos em branco */
10
11     std::cout << "Digite sua idade: ";
12     std::string idade;
13     std::cin >> idade;
14
15     std::cout << "Seu nome eh " << nome <<
16               " e sua idade eh " << idade << std::endl;
17
18     return 0;
19 }
```

- `std::cin` não é recomendado para ler strings com espaços em branco.



## std::getline()

- A biblioteca `<string>` define uma função global que lê uma linha de texto e guarda numa `std::string`.
- **`istream& getline(istream& is, string& str, char delim);`**
  - `is` = qualquer input stream (`ifstream`, `cin`, etc.)
  - `str` = uma `std::string` que será preenchida com texto
  - `delim` = caractere delimitador de leitura (default é `'\n'`)
  - retorna a `istream is` passada no primeiro argumento
- O texto da stream de entrada será lido até a primeira ocorrência de `delim` (o padrão é `'\n'`) e colocado em `str`. O delimitador será removido do final de `str` e a stream de entrada estará apontando para o primeiro caractere após `delim`.

## Lendo uma `std::string` do teclado (2)

- A fim de ler uma string completa (incluindo os espaços em branco), é melhor usar a função `std::getline()`, que recebe dois parâmetros: o primeiro é `std::cin` e o segundo é uma variável do tipo `std::string`.

## Lendo uma `std::string` do teclado (2)

- A fim de ler uma string completa (incluindo os espaços em branco), é melhor usar a função `std::getline()`, que recebe dois parâmetros: o primeiro é `std::cin` e o segundo é uma variável do tipo `std::string`.

```
1 #include <string> // prog23.cpp
2 #include <iostream>
3
4 int main() {
5     std::cout << "Digite seu nome completo: ";
6     std::string nome;
7     std::getline(std::cin, nome); /* le caracteres e os
8                                   armazena na variavel
9                                   nome ate que seja
10                                  digitado o caractere '\n' */
11
12     std::cout << "Seu nome eh " << nome << std::endl;
13
14     return 0;
15 }
```

# Lendo várias strings do teclado

```
1 #include <string> // prog23b.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     string str;
7
8     cout << ">>>: ";
9     while(getline(std::cin, str)) {
10         cout << "digitado: " << str << "\n";
11         cout << "CTRL+D ou CTRL+Z para encerrar\n";
12         cout << ">>>: ";
13     }
14
15     return 0;
16 }
```

## Misturando std::cin e std::getline

```
1 #include <string> // prog24.cpp
2 #include <iostream>
3
4 int main() {
5     std::cout << "Escolha 1 ou 2: ";
6     int escolha { 0 };
7     std::cin >> escolha;
8
9     std::cout << "Digite seu nome: ";
10    std::string nome;
11    std::getline(std::cin, nome); // Problema: nao sera lido
12
13    std::cout << "Ola, " << nome;
14    std::cout << ", voce escolheu " << escolha << '\n';
15
16    return 0;
17 }
```

## Misturando `std::cin` e `std::getline`

```
1 #include <string> // prog24.cpp
2 #include <iostream>
3
4 int main() {
5     std::cout << "Escolha 1 ou 2: ";
6     int escolha { 0 };
7     std::cin >> escolha;
8
9     std::cout << "Digite seu nome: ";
10    std::string nome;
11    std::getline(std::cin, nome); // Problema: nao sera lido
12
13    std::cout << "Ola, " << nome;
14    std::cout << ", voce escolheu " << escolha << '\n';
15
16    return 0;
17 }
```

- Após ler um valor com `std::cin`, o caractere de nova linha `'\n'` permanece no fluxo de entrada.

# Misturando `std::cin` e `std::getline`

```
1 #include <string> // prog24.cpp
2 #include <iostream>
3
4 int main() {
5     std::cout << "Escolha 1 ou 2: ";
6     int escolha { 0 };
7     std::cin >> escolha;
8
9     std::cout << "Digite seu nome: ";
10    std::string nome;
11    std::getline(std::cin, nome); // Problema: nao sera lido
12
13    std::cout << "Ola, " << nome;
14    std::cout << ", voce escolheu " << escolha << '\n';
15
16    return 0;
17 }
```

- Após ler um valor com `std::cin`, o caractere de nova linha `'\n'` permanece no fluxo de entrada.
- Assim, ao chamar a função `std::getline` na sequência, ela vê que `'\n'` está no fluxo, e deduz que inserimos uma string vazia!

# Misturando std::cin e std::getline

```
1 #include <string> // prog25.cpp
2 #include <limits>
3 #include <iostream>
4
5 int main() {
6     std::cout << "Escolha 1 ou 2: ";
7     int escolha { 0 };
8     std::cin >> escolha;
9
10    // Ignora o primeiro caractere armazenado no buffer.
11    std::cin.ignore();
12
13    std::cout << "Digite seu nome: ";
14    std::string nome;
15    std::getline(std::cin, nome); // OK agora!
16
17    std::cout << "Ola, " << nome;
18    std::cout << ", voce escolheu " << escolha << '\n';
19
20    return 0;
21 }
```



# Operações com strings



# Atribuição de strings — operador =

```
1 #include <string> // prog26a.cpp
2 #include <iostream>
3
4 int main() {
5     std::string palavra1("fortaleza");
6     std::string palavra2("caucaia");
7
8     palavra1 = palavra2; // atribuicao
9
10    std::cout << palavra1 << std::endl; // caucaia
11    std::cout << palavra2 << std::endl; // caucaia
12 }
```

# Concatenando strings — operadores + e +=

```
1 #include <string> // prog26.cpp
2 #include <iostream>
3
4 int main() {
5     std::string a("45");
6     std::string b("11");
7
8     std::cout << a + b << "\n"; // a, b serao concatenadas
9     a += " volts";
10    std::cout << a << "\n";
11
12    return 0;
13 }
```

## Accesando elementos em uma `std::string`

```
1 #include <iostream> // prog27.cpp
2 #include <string>
3 using namespace std;
4
5 int main () {
6     string str("abcdefg");
7     cout << str[5] << endl; // Usando o operador [ ]
8     str[5] = 'X';
9     cout << str << endl;
```

## Accesando elementos em uma `std::string`

```
1 #include <iostream> // prog27.cpp
2 #include <string>
3 using namespace std;
4
5 int main () {
6     string str("abcdefg");
7     cout << str[5] << endl; // Usando o operador [ ]
8     str[5] = 'X';
9     cout << str << endl;
```

## Accesando elementos em uma `std::string`

```
1 #include <iostream> // prog27.cpp
2 #include <string>
3 using namespace std;
4
5 int main () {
6     string str("abcdefg");
7     cout << str[5] << endl; // Usando o operador [ ]
8     str[5] = 'X';
9     cout << str << endl;
10
11     string str2("mnopqrstuv");
12     cout << str2.at(5) << endl; // usando o metodo at()
13     str2.at(5) = 'X';
14     cout << str2 << endl;
15
16     cout << str2.substr(2, 3) << endl;
17
18     return 0;
19 }
```

- A função-membro `at()` é mais lento que o operador `[]`, pois usa exceções para verificar se o índice passado como parâmetro é válido.

## std::string — Funções-membro

- string `substr (size_t pos = 0, size_t len = npos)`

Retorna uma substring da string. A substring retornada contém os caracteres que começam na posição `pos` e abrange `len` caracteres (ou até o final da string, o que ocorrer primeiro).

- `const char* c_str()`  
`char* data()`

Essas duas funções-membro retornam um ponteiro para um array que contém uma sequência de caracteres terminada em nulo (uma string C) representando o valor atual do objeto string.

Esse array contém a mesma sequência de caracteres que compõem o valor do objeto string mais um caractere nulo de terminação adicional (`\0`) no final.

## std::string — Funções-membro

size_t <code>size()</code>	retorna o número de bytes da string
size_t <code>length()</code>	retorna o número de bytes da string
void <code>clear()</code>	deixa a string vazia
bool <code>empty()</code>	testa se a string está vazia
char& <code>at (size_t pos)</code>	retorna uma referência para o caractere na posição <code>pos</code> especificada
char& <code>back()</code>	retorna uma referência para o último caractere da string
char& <code>front()</code>	retorna uma referência para o primeiro caractere da string
void <code>pop_back()</code>	Apaga o último caractere da string.
string& <code>append (string str)</code> string& <code>append (char* s)</code>	Estende a string anexando caracteres adicionais no final de seu valor atual.
<code>==, !=, &lt;, &lt;=, &gt;, &gt;=</code>	comparam duas strings lexicograficamente



## std::string — Mais informações

Mais detalhes sobre as funções da classe string:

<https://cplusplus.com/reference/string/string/>

## Convertendo strings para números



# Conversões `std::string` $\longleftrightarrow$ tipos nativos

- É possível usar funções pré-definidas no cabeçalho `<string>` para converter string para tipos nativos e vice-versa.
- As funções abaixo estão na biblioteca `<string>`.

Função	Significado
<code>stoi</code>	converte string para int
<code>stol</code>	converte string para long int
<code>stoul</code>	converte string para unsigned int
<code>stoll</code>	converte string para long long
<code>stoull</code>	converte string para unsigned long long
<code>stof</code>	converte string para float
<code>stod</code>	converte string para double
<code>stold</code>	converte string para long double

<code>to_string</code>	converte um valor numérico para string
------------------------	--

# Exemplo: conversão de string para numérico

```
1 #include <string> // prog120.cpp
2 #include <iomanip>
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     int v1 {stoi("-12")};
8     long unsigned int v2 {stoul("13")};
9     float v3 {stof("13.456")};
10    double v4 {stod("201.65432")};
11
12    cout << setw(15) << "int = " << v1 << '\n';
13    cout << setw(15) << "unsigned int = " << v2 << '\n';
14    cout << setw(15) << "float = " << v3 << '\n';
15    cout << setw(15) << "double = " << v4 << '\n';
16 }
```

# Exemplo: conversão de numérico para string

```
1 #include <string> // prog121.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     int v1 {-12};
7     unsigned int v2 {13};
8     float v3 {13.456};
9     double v4 {201.65432};
10
11     cout << to_string(v1) << '\n';
12     cout << to_string(v2) << '\n';
13     cout << to_string(v3) << '\n';
14     cout << to_string(v4) << '\n';
15 }
```

# String Streams



# String streams

- Vimos dois tipos de streams (`ostream` e `istream`), cujos representantes `std::cin` e `std::cout` possibilitam nossa interação com o terminal.

# String streams

- Vimos dois tipos de streams (`ostream` e `istream`), cujos representantes `std::cin` e `std::cout` possibilitam nossa interação com o terminal.
- A biblioteca padrão do C++ também disponibiliza streams que permitem enviar e receber dados de uma `std::string`.



# String streams

- Vimos dois tipos de streams (`ostream` e `istream`), cujos representantes `std::cin` e `std::cout` possibilitam nossa interação com o terminal.
- A biblioteca padrão do C++ também disponibiliza streams que permitem enviar e receber dados de uma `std::string`.
- Essas streams estão definidas no cabeçalho `<sstream>`:
  - `std::istringstream`
  - `std::ostringstream`
  - `std::stringstream`

# String streams

- Vimos dois tipos de streams (`ostream` e `istream`), cujos representantes `std::cin` e `std::cout` possibilitam nossa interação com o terminal.
- A biblioteca padrão do C++ também disponibiliza streams que permitem enviar e receber dados de uma `std::string`.
- Essas streams estão definidas no cabeçalho `<sstream>`:
  - `std::istringstream`
  - `std::ostringstream`
  - `std::stringstream`
- Qual a utilidade disso?
  - Bufferizar dados de saída para exibição posterior
  - Quebrar uma `std::string` em pedaços (*string tokenization*)
  - Converter tipos nativos para `std::string` e vice-versa

# Input String Streams

- Para extrair dados de uma `std::string`, declare uma variável do tipo `std::istream`

# Input String Streams

- Para extrair dados de uma `std::string`, declare uma variável do tipo `std::istream`

- O conteúdo da variável pode ser definido usando a função-membro:  
`str(std::string str)`

Mas também pode ser definido no momento da declaração da variável.

# Input String Streams

- Para extrair dados de uma `std::string`, declare uma variável do tipo `std::istream`

- O conteúdo da variável pode ser definido usando a função-membro:

```
str(std::string str)
```

Mas também pode ser definido no momento da declaração da variável.

## Exemplo:

```
std::istream inputStream;  
inputStream.str("123 45.6 Luciana");
```

# Input String Streams

- Para extrair dados de uma `std::string`, declare uma variável do tipo `std::istream`

- O conteúdo da variável pode ser definido usando a função-membro:

```
str(std::string str)
```

Mas também pode ser definido no momento da declaração da variável.

## Exemplo:

```
std::istream inputStream;  
inputStream.str("123 45.6 Luciana");  
std::istream inputstr("123 45.6 Luciana");
```

# Input String Streams

- Para extrair dados de uma `std::string`, declare uma variável do tipo `std::istream`
- O conteúdo da variável pode ser definido usando a função-membro:  
`str(std::string str)`

Mas também pode ser definido no momento da declaração da variável.

## Exemplo:

```
std::istream inputStream;  
inputStream.str("123 45.6 Luciana");  
std::istream inputstr("123 45.6 Luciana");
```

- Para “extrair” os valores, usamos o operador de extração `>>`, de forma similar ao que fazemos com o objeto `std::cin`.

## Exemplo 1: std::istringstream

```
1 #include <string> // prog122.cpp
2 #include <sstream>
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     istringstream iss;
8     iss.str("123.45.67 Ana");
9
10    int a;
11    float b;
12    string c, d;
13
14    iss >> a >> b >> c >> d;
15
16    cout << a << " " << b << " " << c << " " << d << endl;
17 }
```



## std::istream — Observações

- Ao tentar extrair um valor de uma `std::istream` e não existir mais valor a ser extraído, ela entra em um estado inválido (seu `eofbit` é setado para `true`).

## std::istream — Observações

- Ao tentar extrair um valor de uma `std::istream` e não existir mais valor a ser extraído, ela entra em um estado inválido (seu `eofbit` é setado para `true`).
- Por ser uma `stream`, um objeto do tipo `std::istream` também tem os mesmos bits de sinalização de erro que o objeto `std::cin`
- Os bits de erro de um objeto `std::istream` podem ser restaurados à condição inicial usando a função-membro `clear()`.

## std::istream — Observações

- Ao tentar extrair um valor de uma `std::istream` e não existir mais valor a ser extraído, ela entra em um estado inválido (seu `eofbit` é setado para `true`).
- Por ser uma `stream`, um objeto do tipo `std::istream` também tem os mesmos bits de sinalização de erro que o objeto `std::cin`
- Os bits de erro de um objeto `std::istream` podem ser restaurados à condição inicial usando a função-membro `clear()`.
- Para limpar o buffer, podemos usar qualquer uma das funções-membros:
  - `str(std::string)`
  - `ignore(int n, int delim)`

# Exercício

- 1) Faça um programa que lê do terminal uma string contendo uma quantidade indeterminada de números reais (separados por espaço) e imprime a soma de todos os números lidos.
  - Dica: use `istringstream` em algum momento.
  - **Exemplo de Entrada:**  
12 34 56.7 81.3 13
  - **Exemplo de Saída:**  
197
- 2) Depois, crie um outro programa que faz o que foi pedido acima cinco vezes.

# Exercício — Solução 1

```
1 #include <string> // prog123.cpp
2 #include <sstream>
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     double number{}, total{};
8     string line;
9     getline(cin, line);
10    istringstream iss(line);
11
12    while(iss >> number) {
13        total += number;
14    }
15    cout << "total = " << total << endl;
16 }
```

## Exercício — Solução 2.1

```
1 #include <string> // prog124.cpp
2 #include <sstream>
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     double number{}, total{};
8     string line;
9     istringstream iss;
10
11     for(int i{1}; i <= 5; ++i) {
12         getline(cin, line);
13         iss.str(line);
14         while(iss >> number) {
15             total += number;
16         }
17         cout << "total = " << total << endl;
18         iss.clear();
19         iss.str("");
20         total = 0;
21     }
22 }
```

## Exercício — Solução 2.2

```
1 #include <string> // prog125.cpp
2 #include <sstream>
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     for(int i{1}; i <= 5; ++i) {
8         double number{}, total{};
9         string line;
10
11         getline(cin, line);
12         istringstream iss(line);
13
14         while(iss >> number) {
15             total += number;
16         }
17         cout << "total = " << total << endl;
18     }
19 }
```

# Exercício

- 1) Faça um programa que lê do terminal três strings (uma em cada linha), tal que cada string contém o nome, idade e cidade de nascimento de uma pessoa, separados por ponto-e-vírgula (;) e imprime na tela os dados das pessoas, só que agora separados por vírgula.

- **Exemplo de Entrada:**

atilio;21;quixada  
ana maria;22;banabuiu  
tiago;23;pedra branca

- **Exemplo de Saída:**

atilio,21,quixada  
ana maria,22,banabuiu  
tiago,23,pedra branca



# Output String Streams

- Para montar uma string stream, declare uma variável do tipo `std::ostringstream`
  - `std::ostringstream oss;`
  - `std::ostringstream oss("123 45.67 ana");`

# Output String Streams

- Para montar uma string stream, declare uma variável do tipo `std::ostringstream`
  - `std::ostringstream oss;`
  - `std::ostringstream oss("123 45.67 ana");`
- Para inserir os valores, usamos o operador de inserção `<<`, de forma similar ao que fazemos com o objeto `std::cout`

# Output String Streams

- Para montar uma string stream, declare uma variável do tipo `std::ostringstream`
  - `std::ostringstream oss;`
  - `std::ostringstream oss("123 45.67 ana");`
- Para inserir os valores, usamos o operador de inserção `<<`, de forma similar ao que fazemos com o objeto `std::cout`
  - Após inserir os valores, o conteúdo da `ostringstream` pode ser obtido como uma string usando a função-membro `str()`.

## std::ostringstream — Exemplo 1

```
1 #include <string> // prog126.cpp
2 #include <sstream>
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     ostringstream oss;
8
9     // converte varios dados para string
10
11     oss << 123u << " " << 1.234f << " "
12         << '*' << " " << "asa" << " "
13         << -44 << " " << 12.543;
14
15     // imprime a string na tela
16
17     cout << oss.str() << endl;
18 }
```

## std::ostringstream — Exemplo 2

```
1 #include <iostream> // prog126a.cpp
2 #include <sstream>
3 using namespace std;
4
5 int main () {
6     ostringstream far;
7     ostringstream bar (ostringstream::ate); // at end
8     ostringstream car ("Test string",
9                        ostringstream::ate); // at end
10
11     far.str("Test string");
12     bar.str("Test string");
13
14     far << 101;
15     bar << 101;
16     car << 101;
17
18     cout << far.str() << '\n'; // 101t string
19     cout << bar.str() << '\n'; // Test string101
20     cout << car.str() << '\n'; // Test string101
21 }
```

## std::ostringstream — Exemplo 3

**OBS.:** Os manipuladores do cabeçalho `<iomanip>` podem ser usados com `std::ostringstream` a fim de formatar os dados inseridos.

## std::ostringstream — Exemplo 3

**OBS.:** Os manipuladores do cabeçalho `<iomanip>` podem ser usados com `std::ostringstream` a fim de formatar os dados inseridos.

```
1 #include <string> // prog127.cpp
2 #include <iomanip>
3 #include <sstream>
4 #include <iostream>
5 using namespace std;
6
7 int main() {
8     ostringstream oss;
9     oss << fixed << setprecision(2);
10    oss << setw(6) << 12.3456 << '\n';
11    oss << setw(6) << .1321 << '\n';
12    oss << setw(6) << 875.98 << '\n';
13    oss << setw(6) << 24.87654;
14
15    cout << oss.str() << endl;
16 }
```

# std::stringstream

- O cabeçalho `<sstream>` ainda tem um terceiro tipo de stream chamado `std::stringstream`, que pode fazer tanto o papel de uma `istringstream` quanto de uma `ostringstream`.
- **Atenção:** Cuidado! Recomendo não misturar as duas utilidades numa mesma operação.



# Cuidado ao reusar uma stringstream!

- Cuidado ao reutilizar o mesmo objeto stringstream para múltiplas conversões. Pode ser estranho.
  - **Opção 1:** Após usar a stringstream, certifique-se de resetar os bits de erro e reinicie a stringstream com uma string vazia.
    - Para isso, use as funções-membros `clear()` e `str(string&)`
  - **Opção 2:** Ou simplesmente crie uma nova stringstream para cada uso.

## Opção 1: reusando uma stringstream

```
1 #include <iostream> // sstream_test5.cpp
2 #include <string>
3 #include <sstream>
4 using namespace std;
5
6 int main() {
7     stringstream ss;
8     string line;
9     double value;
10
11     for(int i{1}; i <= 3; ++i) {
12         cout << "Digite numeros separados por espaco: ";
13         getline(cin, line);
14         ss.str(line);
15         cout << "Numeros duplicados: ";
16         while(ss >> value) {
17             cout << value * 2 << " ";
18         }
19         cout << endl;
20         ss.clear();
21         ss.str("");
22     }
23 }
```

## Opção 2: usando uma nova stringstream

```
1 #include <iostream> // sstream_test6.cpp
2 #include <string>
3 #include <sstream>
4 using namespace std;
5
6 int main() {
7     string line;
8     double value;
9
10    for(int i {0}; i < 3; ++i) {
11        cout << "Digite numeros separados por espaco: ";
12        getline(cin, line);
13        stringstream ss (line);
14        cout << "Numeros duplicados: ";
15        while(ss >> value) {
16            cout << value * 2 << " ";
17        }
18        cout << endl;
19    }
20 }
```

# Estruturas (structs)



# Estruturas: struct

- Uma **estrutura** é um **tipo de dado composto**.
- Uma **estrutura** pode ser vista como um conjunto de variáveis sob o mesmo nome, e cada uma delas pode ter qualquer tipo válido.

# Estruturas: struct

- Uma **estrutura** é um **tipo de dado composto**.
- Uma **estrutura** pode ser vista como um conjunto de variáveis sob o mesmo nome, e cada uma delas pode ter qualquer tipo válido.
- Definindo uma estrutura:

```
1 struct Empregado {  
2     short id;  
3     int idade;  
4     double salario;  
5 };
```

## Instanciando e inicializando um struct

```
1 #include <string> // prog08.cpp
2 #include <iostream>
3 using namespace std;
4
5 struct Empregado {
6     short id;
7     string nome;
8     int idade;
9     double salario;
10 };
11
12 int main () {
13     Empregado carlos;
14     carlos.id = 10;
15     carlos.nome = "Carlos Gomes";
16     carlos.idade = 23;
17     carlos.salario = 985.98;
18
19     cout << "ID: " << carlos.id
20         << "\nNome: " << carlos.nome
21         << "\nIdade: " << carlos.idade
22         << "\nSalario: " << carlos.salario << endl;
23 }
```

# Instanciando e inicializando um struct

```
1 #include <string> // prog08a.cpp
2 #include <iostream>
3 using namespace std;
4
5 struct Pessoa {
6     string nome;
7     int idade;
8     float peso;
9 };
10
11 void imprime_pessoa(Pessoa pessoa) {
12     cout << "Nome: " << pessoa.nome
13         << "\nIdade: " << pessoa.idade
14         << "\nPeso: " << pessoa.peso << endl;
15 }
16
17 int main () {
18     Pessoa p {"Paula", 23, 57.65}; // a partir do C++11
19     imprime_pessoa(p);
20 }
```



# Exercícios

- Crie uma estrutura para representar as coordenadas de um ponto no plano (posições X e Y). Em seguida, declare e leia do teclado um ponto e exiba a distância dele até a origem das coordenadas, isto é, a posição (0,0).

**Dica:** A distância entre dois pontos  $P(x_p, y_p)$  e  $Q(x_q, y_q)$ , é dada pela fórmula  $dist(P, Q) = \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}$ .

- Crie uma estrutura chamada **Retangulo**. Essa estrutura deverá conter o ponto superior esquerdo e o ponto inferior direito do retângulo. Cada ponto é definido por uma estrutura **Ponto**, a qual contém as posições X e Y. Faça um programa que declare e leia uma estrutura Retangulo e exiba a área, o comprimento da diagonal e o perímetro desse retângulo.
- Usando a estrutura Retângulo do exercício anterior, faça um programa que declare e leia uma estrutura Retângulo e um Ponto, e informe se esse ponto está ou não dentro do retângulo.

# Exercícios

- Crie uma estrutura representando um aluno de uma disciplina. Essa estrutura deve conter o número de matrícula do aluno, seu nome e as notas de três provas. Agora, escreva um programa que leia os dados de cinco alunos e os armazena nessa estrutura. Em seguida, exiba o nome e as notas do aluno que possui a maior média geral dentre os cinco.

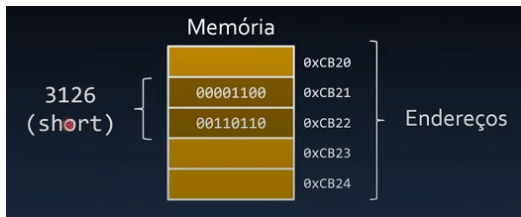
- Faça um programa para ler os dados de 100 empregados e depois imprimir esses dados na tela. Os dados a serem lidos são:
  - nome completo
  - endereço
  - CPF
  - salário
- Os dados devem ser armazenados na memória, pois caso seja preciso mudar algum dado de algum usuário, seja possível acessar a variável em que este dado está armazenado para mudá-lo.

# Ponteiros



# Variáveis

- Ao armazenar dados, um programa gerencia:
  - Onde a informação está armazenada
  - Que tipo de informação é armazenada
  - O valor que é mantido lá

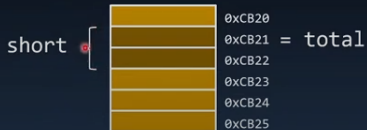


# Variáveis

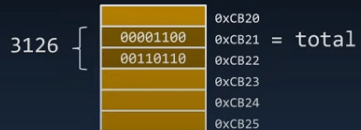
- A declaração de uma variável em um programa realiza os passos necessários para o armazenamento de dados

```
short total;    // declaração de variável  
total = 3126;  // atribuição de valor
```

## Declaração



## Atribuição

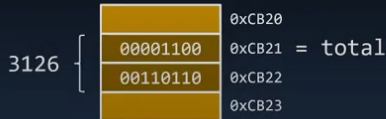


# Endereços de Variáveis

- Todo nome de variável está associado a um endereço de memória.
  - O **operador de endereço &** obtém a sua localização

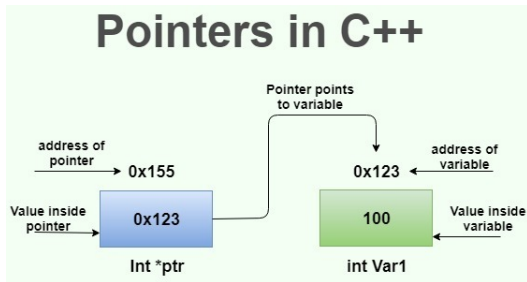
```
short total;           // declaração
total = 3126;          // atribuição

cout << total;         // valor
cout << &total;       // localização
```



# Ponteiros

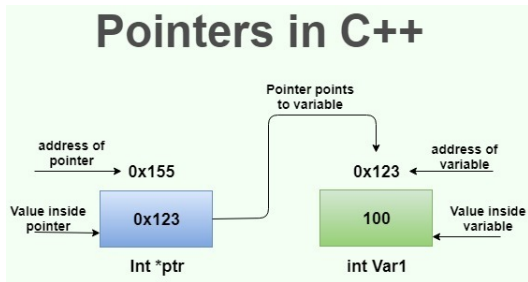
- **Ponteiro:** é um tipo de variável usada para guardar um endereço de memória.





# Ponteiros

- **Ponteiro:** é um tipo de variável usada para guardar um endereço de memória.



- Cada variável tem um endereço, inclusive o ponteiro.

## Declarando ponteiros

- Em C++, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

## Declarando ponteiros

- Em C++, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

- É o operador asterisco (\*) que informa ao compilador que a variável `nome_do_ponteiro` não vai guardar um valor, mas um endereço de memória para o tipo especificado.

# Declarando ponteiros

- Em C++, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

- É o operador asterisco (\*) que informa ao compilador que a variável `nome_do_ponteiro` não vai guardar um valor, mas um endereço de memória para o tipo especificado.

- Exemplo:

```
1 int *p_int; // p_int eh um ponteiro para int
2 double *p_d; // p_d eh um ponteiro para double
```

# Declarando ponteiros

- Em C++, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

- É o operador asterisco (\*) que informa ao compilador que a variável `nome_do_ponteiro` não vai guardar um valor, mas um endereço de memória para o tipo especificado.

- Exemplo:

```
1 int *p_int; // p_int eh um ponteiro para int
2 double *p_d; // p_d eh um ponteiro para double
```

- Quando declaramos um ponteiro, informamos ao compilador para que tipo de variável poderemos apontá-lo.

## Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o **operador de endereçamento**, `&`, na frente do nome da variável.

# Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o **operador de endereçamento**, **&**, na frente do nome da variável.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int x = 5;
6     cout << x << '\n'; // imprime 5
7
8     // imprime o endereço de memória da variavel x
9     std::cout << &x << endl;
10 }
```

# Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o **operador de endereçamento**, **&**, na frente do nome da variável.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int x = 5;
6     cout << x << '\n'; // imprime 5
7
8     // imprime o endereço de memoria da variavel x
9     std::cout << &x << endl;
10 }
```

- Ao se trabalhar com ponteiros, **duas tarefas básicas** serão sempre executadas:



## Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o **operador de endereçamento**, **&**, na frente do nome da variável.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int x = 5;
6     cout << x << '\n'; // imprime 5
7
8     // imprime o endereço de memoria da variavel x
9     std::cout << &x << endl;
10 }
```

- Ao se trabalhar com ponteiros, **duas tarefas básicas** serão sempre executadas:
  - Acessar o endereço de memória de uma variável, usando o operador **&**

# Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o **operador de endereçamento**, **&**, na frente do nome da variável.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int x = 5;
6     cout << x << '\n'; // imprime 5
7
8     // imprime o endereço de memoria da variavel x
9     std::cout << &x << endl;
10 }
```

- Ao se trabalhar com ponteiros, **duas tarefas básicas** serão sempre executadas:
  - Acessar o endereço de memória de uma variável, usando o operador **&**
  - Acessar o conteúdo de um endereço de memória, usando o operador **\***

## Manipulando ponteiros — Exemplo

```
1 #include <iostream> // prog35.cpp
2 using namespace std;
3
4 int main() {
5     // Declara uma variavel int contendo o valor 10
6     int count = 10;
```

# Manipulando ponteiros — Exemplo

```
1 #include <iostream> // prog35.cpp
2 using namespace std;
3
4 int main() {
5     // Declara uma variavel int contendo o valor 10
6     int count = 10;
7
8     // Declara um ponteiro para int e atribui ao ponteiro
9     // o endereco da variavel int
10    int *p;
11    p = &count;
```

## Manipulando ponteiros — Exemplo

```
1 #include <iostream> // prog35.cpp
2 using namespace std;
3
4 int main() {
5     // Declara uma variavel int contendo o valor 10
6     int count = 10;
7
8     // Declara um ponteiro para int e atribui ao ponteiro
9     // o endereco da variavel int
10    int *p;
11    p = &count;
12    cout << "Conteudo de p: " << p << "\n";
13    //Imprime 10
14    cout << "Conteudo apontado por p: " << *p << "\n";
```

## Manipulando ponteiros — Exemplo

```
1 #include <iostream> // prog35.cpp
2 using namespace std;
3
4 int main() {
5     // Declara uma variavel int contendo o valor 10
6     int count = 10;
7
8     // Declara um ponteiro para int e atribui ao ponteiro
9     // o endereco da variavel int
10    int *p;
11    p = &count;
12    cout << "Conteudo de p: " << p << "\n";
13    //Imprime 10
14    cout << "Conteudo apontado por p: " << *p << "\n";
15
16    // Atribui um novo valor a posicao de memoria apontada por p
17    *p = 12;
```

# Manipulando ponteiros — Exemplo

```
1 #include <iostream> // prog35.cpp
2 using namespace std;
3
4 int main() {
5     // Declara uma variavel int contendo o valor 10
6     int count = 10;
7
8     // Declara um ponteiro para int e atribui ao ponteiro
9     // o endereco da variavel int
10    int *p;
11    p = &count;
12    cout << "Conteudo de p: " << p << "\n";
13    //Imprime 10
14    cout << "Conteudo apontado por p: " << *p << "\n";
15
16    // Atribui um novo valor a posicao de memoria apontada por p
17    *p = 12;
18
19    // As duas linhas abaixo imprimem o numero 12 na tela
20    cout << "Conteudo apontado por p: " << *p << "\n";
21    cout << "Conteudo de count: " << count << "\n";
22
23    return 0;
24 }
```

## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.



## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <iostream> // prog37.cpp
2
3 int main() {
4     float f = 45.78;
5     int *ptr = &f; // Erro de compilacao
```

## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <iostream> // prog37.cpp
2
3 int main() {
4     float f = 45.78;
5     int *ptr = &f; // Erro de compilacao
6     int i = 54;
7     float *f2 = &i; // Erro de compilacao
```

## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <iostream> // prog37.cpp
2
3 int main() {
4     float f = 45.78;
5     int *ptr = &f; // Erro de compilacao
6     int i = 54;
7     float *f2 = &i; // Erro de compilacao
8
9     float *fptr = &f; // OK
```

## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <iostream> // prog37.cpp
2
3 int main() {
4     float f = 45.78;
5     int *ptr = &f; // Erro de compilacao
6     int i = 54;
7     float *f2 = &i; // Erro de compilacao
8
9     float *fptr = &f; // OK
10    int *iptr = &i; // OK
```

## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <iostream> // prog37.cpp
2
3 int main() {
4     float f = 45.78;
5     int *ptr = &f; // Erro de compilacao
6     int i = 54;
7     float *f2 = &i; // Erro de compilacao
8
9     float *fptr = &f; // OK
10    int *iptr = &i; // OK
11    fptr = iptr; // Erro de compilacao
```

## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <iostream> // prog37.cpp
2
3 int main() {
4     float f = 45.78;
5     int *ptr = &f; // Erro de compilacao
6     int i = 54;
7     float *f2 = &i; // Erro de compilacao
8
9     float *fptr = &f; // OK
10    int *iptr = &i; // OK
11    fptr = iptr; // Erro de compilacao
12    float *f3 = fptr; // OK
```

## Atribuição entre ponteiros

- Em geral, no C++, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <iostream> // prog37.cpp
2
3 int main() {
4     float f = 45.78;
5     int *ptr = &f; // Erro de compilacao
6     int i = 54;
7     float *f2 = &i; // Erro de compilacao
8
9     float *fptr = &f; // OK
10    int *iptr = &i; // OK
11    fptr = iptr; // Erro de compilacao
12    float *f3 = fptr; // OK
13
14    // imprime 45.78 e 54
15    std::cout << *fptr << ", " << *iptr << std::endl;
16
17    return 0;
18 }
```

# Qual o tamanho de um ponteiro?

- O tamanho de um ponteiro depende da arquitetura para a qual o programa é compilado.
  - Arquitetura de 32 bits — ponteiro ocupa 32 bits (4 bytes).
  - Arquitetura de 64 bits — ponteiro ocupa 64 bits (8 bytes).Independentemente do que está sendo apontado.



# Ponteiro nulo

- Além de endereços de memória, todo ponteiro pode armazenar o valor nulo.

# Ponteiro nulo

- Além de endereços de memória, todo ponteiro pode armazenar o valor nulo.
- **Valor nulo** é um valor especial que significa que o ponteiro não está apontando para nada. Um ponteiro que contém um valor nulo é chamado de **ponteiro nulo**.

# Ponteiro nulo

- Além de endereços de memória, todo ponteiro pode armazenar o valor nulo.
- **Valor nulo** é um valor especial que significa que o ponteiro não está apontando para nada. Um ponteiro que contém um valor nulo é chamado de **ponteiro nulo**.
- Em C++ moderno, a forma de tornar um ponteiro nulo consiste em atribuir-lhe a palavra-chave `nullptr`.

## Ponteiro nulo

- **Boa prática de programação 1:** Se, no momento da declaração de um ponteiro, nenhum endereço de memória válido for atribuído ao ponteiro, então inicialize o ponteiro com um valor nulo.

## Ponteiro nulo

- **Boa prática de programação 1:** Se, no momento da declaração de um ponteiro, nenhum endereço de memória válido for atribuído ao ponteiro, então inicialize o ponteiro com um valor nulo.
- **Erro Comum em C++:** Tentar desreferenciar um ponteiro não inicializado ou um ponteiro nulo é ilegal e potencialmente fará com que seu programa seja encerrado.

```
1 int *ptr = nullptr;  
2 *ptr = 2023; // erro: falha de segmentacao
```

## Ponteiro nulo

- **Boa prática de programação 1:** Se, no momento da declaração de um ponteiro, nenhum endereço de memória válido for atribuído ao ponteiro, então inicialize o ponteiro com um valor nulo.
- **Erro Comum em C++:** Tentar desreferenciar um ponteiro não inicializado ou um ponteiro nulo é ilegal e potencialmente fará com que seu programa seja encerrado.

```
1 int *ptr = nullptr;  
2 *ptr = 2023; // erro: falha de segmentacao
```

- **Boa prática de programação 2:** Antes de usar um ponteiro, certifique-se de que ele não é um ponteiro nulo. **Exemplo:**

```
1 if (ptr != nullptr) {  
2     *ptr = 2023; // ptr NAO eh nulo  
3 } else {  
4     cout << "ponteiro nulo" << endl;  
5 }
```

# Exercício

- Implemente uma função que recebe dois valores do tipo `int` como argumento e troca os valores dos inteiros. Sua função deve obedecer o protótipo:  
`void troca(int *p1, int *p2);`

## Ponteiros e arrays





# Ponteiros e arrays

- Ponteiros e arrays estão intrinsecamente relacionados.

# Ponteiros e arrays

- Ponteiros e arrays estão intrinsecamente relacionados.
- Quando um array é usado em uma expressão, ele **decai** (é implicitamente convertido) em um ponteiro que aponta para o primeiro elemento do vetor. **Exemplo:**

## Ponteiros e arrays — Exemplo

```
1 #include <iostream> // prog38.cpp
2 using namespace std;
3
4 int main() {
5     int array[5] = { 9,7,5,3,1 };
6
7     // valor do primeiro elemento de 'array'
8     cout << "array[0]: " << array[0] << '\n'; // 9
9
10    // imprime o endereco do primeiro elemento de 'array'
11    cout << "Endereco de array[0]: " << &array[0] << '\n'; //
    hexa
12
13    // imprime o valor do ponteiro para o qual 'array' decai
14    cout << "array decai para um ponteiro com endereco: ";
15    cout << array << '\n'; // hexa
16
17    for(int i = 0; i < 5; ++i)
18        cout << i << ": " << array[i] << endl;
19
20    return 0;
21 }
```

# Ponteiros e arrays

- Como um array, usado em uma expressão, decai para um ponteiro que aponta para o primeiro elemento do array, podemos derreferenciar o array a fim de obter o valor do primeiro elemento:

# Ponteiros e arrays

- Como um array, usado em uma expressão, decai para um ponteiro que aponta para o primeiro elemento do array, podemos derreferenciar o array a fim de obter o valor do primeiro elemento:

```
1 #include <iostream> // prog39.cpp
2
3 int main() {
4     int array[5] = { 9,7,5,3,1 };
```

# Ponteiros e arrays

- Como um array, usado em uma expressão, decai para um ponteiro que aponta para o primeiro elemento do array, podemos derreferenciar o array a fim de obter o valor do primeiro elemento:

```
1 #include <iostream> // prog39.cpp
2
3 int main() {
4     int array[5] = { 9,7,5,3,1 };
```

# Ponteiros e arrays

- Como um array, usado em uma expressão, decai para um ponteiro que aponta para o primeiro elemento do array, podemos dereferenciar o array a fim de obter o valor do primeiro elemento:

```
1 #include <iostream> // prog39.cpp
2
3 int main() {
4     int array[5] = { 9,7,5,3,1 };
5
6     // dereferenciar um array retorna o primeiro elemento
7     std::cout << *array << "\n"; // imprime 9!
```

# Ponteiros e arrays

- Como um array, usado em uma expressão, decai para um ponteiro que aponta para o primeiro elemento do array, podemos dereferenciar o array a fim de obter o valor do primeiro elemento:

```
1 #include <iostream> // prog39.cpp
2
3 int main() {
4     int array[5] = { 9,7,5,3,1 };
5
6     // dereferenciar um array retorna o primeiro elemento
7     std::cout << *array << "\n"; // imprime 9!
8
9     // Dada essa propriedade dos arrays, podemos declarar
10    // um ponteiro do tipo int e fazer ele apontar para array
11    int *ptr = array;
12    std::cout << *ptr << "\n"; // imprime 9
13
14    return 0;
15 }
```



# Passando arrays como argumentos para funções

- Em C++, arrays são sempre passados por referência.

# Passando arrays como argumentos para funções

- Em C++, arrays são sempre passados por referência.
- Ao passar um array como um argumento para uma função, ele decai em um ponteiro e o ponteiro é passado para a função:

# Passando arrays como argumentos para funções

```
1 #include <iostream> // prog42.cpp
2 using namespace std;
3
4 void printSize(int array[]) {
5     // array eh tratado como ponteiro aqui
6     // o tamanho do ponteiro sera impresso
7     std::cout << sizeof(array) << '\n';
8 }
9
10 int main() {
11     int array[] = { 1,1,2,3,5,8,13,21 };
12
13     // imprime sizeof(int) * array size que eh igual a 32
14     std::cout << sizeof(array) << '\n';
15
16     printSize(array); // array decai para um ponteiro aqui
17
18     return 0;
19 }
```

# Passando arrays como argumentos para funções

```
1 #include <iostream> // prog43.cpp
2
3 // ptr contém uma copia do endereço passado como parametro
4 void modifica_array(int ptr[]) {
5     *ptr = 5; // o que faz essa linha?
6 }
7
8 int main() {
9     int vec[] = { 1,4,2,3,7,8,13,21 };
10    modifica_array(vec);
11    int size = sizeof(vec) / sizeof(vec[0]);
12    for(int i = 0; i < size; i++)
13        std::cout << "vec[" << i << "] = " << vec[i] << '\n';
14    return 0;
15 }
```

- Quando `modifica_array()` é chamado, `vec` decai em um ponteiro e o valor desse ponteiro é copiado no parâmetro `ptr`.

# Passando arrays como argumentos para funções

```
1 #include <iostream> // prog43.cpp
2
3 // ptr contém uma copia do endereço passado como parametro
4 void modifica_array(int ptr[]) {
5     *ptr = 5; // o que faz essa linha?
6 }
7
8 int main() {
9     int vec[] = { 1,4,2,3,7,8,13,21 };
10    modifica_array(vec);
11    int size = sizeof(vec) / sizeof(vec[0]);
12    for(int i = 0; i < size; i++)
13        std::cout << "vec[" << i << "] = " << vec[i] << '\n';
14    return 0;
15 }
```

- Quando `modifica_array()` é chamado, `vec` decai em um ponteiro e o valor desse ponteiro é copiado no parâmetro `ptr`.
- Embora o valor em `ptr` seja uma cópia do endereço de `vec`, `ptr` ainda aponta para o vetor real. Assim, quando `ptr` é desreferenciado, o vetor é desreferenciado.

## Aritmética de ponteiros

- Apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: **adição** e **subtração**.

## Aritmética de ponteiros

- Apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: **adição** e **subtração**.
- Se `ptr` apontar para um inteiro, `ptr+1` é o endereço do próximo inteiro na memória após o `ptr`.  
E `ptr-1` é o endereço do inteiro antes de `ptr`.

# Aritmética de ponteiros

- Apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: **adição** e **subtração**.
- Se `ptr` apontar para um inteiro, `ptr+1` é o endereço do próximo inteiro na memória após o `ptr`.

E `ptr-1` é o endereço do inteiro antes de `ptr`.

```
1 #include <iostream> // prog44.cpp
2
3 int main() {
4     int array[5] = { 0,1,2,3,4 };
5     int *ptr = array; // ptr aponta para o primeiro
6                       // elemento do array
7
8     std::cout << ptr << ": " << *ptr << '\n';
9     std::cout << ptr+1 << ": " << *(ptr+1) << '\n';
10    std::cout << ptr+2 << ": " << *(ptr+2) << '\n';
11    std::cout << ptr+3-2 << ": " << *(ptr+3-2) << '\n';
12
13    return 0;
14 }
```



# Aritmética de ponteiros

- Subtrair dois ponteiros dá a distância entre eles. Os ponteiros subtraídos devem ser do mesmo tipo.

# Aritmética de ponteiros

- Subtrair dois ponteiros dá a distância entre eles. Os ponteiros subtraídos devem ser do mesmo tipo.
- O valor resultante da subtração de dois ponteiros é do tipo `ptrdiff_t`, que é um inteiro com sinal.

# Aritmética de ponteiros

- Subtrair dois ponteiros dá a distância entre eles. Os ponteiros subtraídos devem ser do mesmo tipo.
- O valor resultante da subtração de dois ponteiros é do tipo `ptrdiff_t`, que é um inteiro com sinal.

```
1 #include <iostream> // prog45.cpp
2
3 int main() {
4     int array[5] = { 9, 1, 2, 3, 4 };
5
6     int *ptr = array;
7
8     ptrdiff_t valor = (ptr + 5) - ptr; // valor == 5
9
10    std::cout << valor << std::endl;
11
12    return 0;
13 }
```

# Ponteiros e indexação de arrays

- Quando o compilador vê o operador de indexação [ ], ele o traduz em uma adição de ponteiro seguida de derreferência.
  - ou seja, `array[n]` é o mesmo que `*(array+n)`, onde `n` é um inteiro.

# Ponteiros e indexação de arrays

- Quando o compilador vê o operador de indexação [ ], ele o traduz em uma adição de ponteiro seguida de derreferência.
  - ou seja, `array[n]` é o mesmo que `*(array+n)`, onde `n` é um inteiro.

```
1 #include <iostream> // prog47.cpp
2
3 int main() {
4     int array [5] = { 9,7,5,3,1 };
5
6     // imprime o endereco do elemento array[1]
7     std::cout << &array[1] << '\n';
8     // imprime o endereco do ponteiro (array+1)
9     std::cout << array+1 << '\n';
10
11     std::cout << array[1] << '\n'; // imprime 7
12     std::cout << *(array+1) << '\n'; // imprime 7
13
14     return 0;
15 }
```

# Ponteiros e operadores relacionais

- Os operadores relacionais ( $>$ ,  $<$ ,  $>=$ ,  $<=$ ,  $==$ ,  $!=$ ) podem ser usados para comparar dois ponteiros.

# Ponteiros e operadores relacionais

- Os operadores relacionais ( $>$ ,  $<$ ,  $>=$ ,  $<=$ ,  $==$ ,  $!=$ ) podem ser usados para comparar dois ponteiros.
- **Exemplo:** (percorrendo um array)

# Ponteiros e operadores relacionais

- Os operadores relacionais ( $>$ ,  $<$ ,  $>=$ ,  $<=$ ,  $=$ ,  $!=$ ) podem ser usados para comparar dois ponteiros.
- **Exemplo:** (percorrendo um array)

```
1 #include <iostream> // prog46.cpp
2
3 int main() {
4     int array[5] = { 0,1,2,3,4 };
5     int *b = array, *e = array + 5;
6
7     while (b < e) { // imprime os elementos do array
8         std::cout << *b << " ";
9         b++;
10    }
11    std::cout << std::endl;
12    return 0;
13 }
```



# Ponteiros e structs



# Ponteiros e estruturas

- Ao criarmos uma variável de um tipo `struct`, esta é armazenada na memória como qualquer outra variável, e portanto possui um endereço.
- É possível então criar um ponteiro para uma variável de um tipo `struct`.

## Ponteiros e estruturas

- Para acessarmos os campos de uma variável struct via um ponteiro, podemos utilizar o operador ( \* ) juntamente com o operador ( . ) como de costume:

```
1 Ponto p1, *p3;  
2 p3 = &p1;  
3 (*p3).x = 1.5;  
4 (*p3).y = 1.5;
```

## Ponteiros e estruturas

- Para acessarmos os campos de uma variável struct via um ponteiro, podemos utilizar o operador ( `*` ) juntamente com o operador ( `.` ) como de costume:

```
1 Ponto p1, *p3;  
2 p3 = &p1;  
3 (*p3).x = 1.5;  
4 (*p3).y = 1.5;
```

- Em C também podemos usar o operador ( `->` ) para acessar campos de uma estrutura via um ponteiro.

```
1 Ponto p1, *p3;  
2 p3 = &p1;  
3 p3->x = 1.5;  
4 p3->y = 1.5;
```

# Ponteiros e estruturas

- Para acessarmos os campos de uma variável struct via um ponteiro, podemos utilizar o operador ( `*` ) juntamente com o operador ( `.` ) como de costume:

```
1 Ponto p1, *p3;  
2 p3 = &p1;  
3 (*p3).x = 1.5;  
4 (*p3).y = 1.5;
```

- Em C também podemos usar o operador ( `->` ) para acessar campos de uma estrutura via um ponteiro.

```
1 Ponto p1, *p3;  
2 p3 = &p1;  
3 p3->x = 1.5;  
4 p3->y = 1.5;
```

- Para acessar campos de estruturas via ponteiros use um dos dois:
  - `ponteiroEstrutura->campo`
  - `(*ponteiroEstrutura).campo`

# O que será impresso pelo programa abaixo?

```
1 #include <iostream> // prog98.cpp
2
3 struct Ponto {
4     double x;
5     double y;
6 };
7
8 int main() {
9     Ponto p1, p2, *p3, *p4;
10    p3 = &p1;
11    p4 = &p2;
12
13    p1.x = 1; p1.y = 2;
14    p2.x = 3; p2.y = 4;
15
16    (*p3).x = 2.5;
17    (*p3).y = 2.5;
18
19    p4->x = 4.5;
20    p4->y = 4.5;
21
22    std::cout << "p1 = (" << p1.x << "," << p1.y << ")\n";
23    std::cout << "p1 = (" << p2.x << "," << p2.y << ")\n";
24 }
```

# Exercício

Um ponto no plano cartesiano é definido pela sua coordenada  $x$  e sua coordenada  $y$ . Seja **Ponto** um struct com dois campos  $x$  e  $y$ , do tipo **float**.

- Implemente uma função que recebe dois valores do tipo **Ponto** como argumento e troca os valores dos pontos. Sua função deve obedecer o protótipo:  
`void troca(Ponto *p1, Ponto *p2);`
- Implemente uma função que recebe um valor do tipo **Ponto** como argumento e dobra os valores das suas coordenadas.

## Exemplo 1 — Ponteiros e Estruturas

```
1 #include <iostream> // prog99.c
2
3 struct Ponto {
4     float x;
5     float y;
6 };
7
8 void troca(Ponto *p1, Ponto *p2) {
9     Ponto aux;
10    aux = *p1;
11    *p1 = *p2;
12    *p2 = aux;
13 }
14
15 int main() {
16     Ponto a = {2, 3}, b = {4.5, 4.5};
17     troca(&a, &b);
18     // 0 que sera impresso?
19     std::cout << "a = (" << a.x << ", " << a.y << ")\n";
20     // 0 que sera impresso?
21     std::cout << "b = (" << b.x << ", " << b.y << ")\n";
22 }
```



## Exemplo 2 — Ponteiros e Estruturas

```
1 #include <iostream> // prog101.c
2
3 struct Ponto {
4     float x;
5     float y;
6 };
7
8 void dobraCoordenada(Ponto *p) {
9     (*p).x = 2 * (*p).x;
10    (*p).y = 2 * (*p).y;
11 }
12
13 int main() {
14     Ponto a = {2, 3};
15     dobraCoordenada(&a);
16     std::cout << a.x << ", " << a.y; // 0 que sera impresso?
17 }
```

## Exemplo 3 — Ponteiros e Estruturas

Equivalente ao exemplo anterior:

```
1 #include <iostream> // prog100.c
2
3 struct Ponto {
4     float x;
5     float y;
6 };
7
8 void dobraCoordenada(Ponto *p) {
9     p->x = 2 * p->x;
10    p->y = 2 * p->y;
11 }
12
13 int main() {
14     Ponto a = {2, 3};
15     dobraCoordenada(&a);
16
17     // 0 que sera impresso?
18     std::cout << "a = " << "(" << a.x << ", " << a.y << ")";
19 }
```

# Alocação Dinâmica de Memória



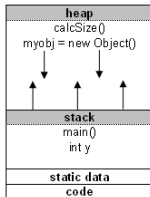
# Organização da memória: Static, Stack e Heap

Os programas escritos em C++, enxergam e dividem a memória em três diferentes regiões:

# Organização da memória: Static, Stack e Heap

Os programas escritos em C++, enxergam e dividem a memória em três diferentes regiões:

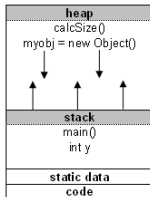
**Static:** persiste durante toda a vida do programa e é geralmente usada para armazenar o código fonte, variáveis globais e variáveis `static`.



# Organização da memória: Static, Stack e Heap

Os programas escritos em C++, enxergam e dividem a memória em três diferentes regiões:

**Static:** persiste durante toda a vida do programa e é geralmente usada para armazenar o código fonte, variáveis globais e variáveis `static`.

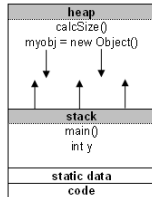


**Stack (Pilha):** usada para armazenar variáveis locais. É gerenciada automaticamente pela CPU. Quando uma função termina a execução, todas as variáveis associadas a essa função na pilha são excluídas e a memória que elas usam é liberada.

# Organização da memória: Static, Stack e Heap

Os programas escritos em C++, enxergam e dividem a memória em três diferentes regiões:

**Static:** persiste durante toda a vida do programa e é geralmente usada para armazenar o código fonte, variáveis globais e variáveis `static`.



**Stack (Pilha):** usada para armazenar variáveis locais. É gerenciada automaticamente pela CPU. Quando uma função termina a execução, todas as variáveis associadas a essa função na pilha são excluídas e a memória que elas usam é liberada.

**Heap:** memória livre disponível que não é gerenciada automaticamente pela CPU — o programador deve alocar e desalocar explicitamente, usando funções como `new` e `delete`.

## Alocação dinâmica de variáveis

- Para alocar dinamicamente uma variável, usamos o operador `new` e para desalocar, usamos o operador `delete`.



# Alocação dinâmica de variáveis

- Para alocar dinamicamente uma variável, usamos o operador `new` e para desalocar, usamos o operador `delete`.
  - Como `new` retorna o endereço de memória da região alocada, devemos atribuir esse endereço a um ponteiro.

# Alocação dinâmica de variáveis

- Para alocar dinamicamente uma variável, usamos o operador **new** e para desalocar, usamos o operador **delete**.
  - Como **new** retorna o endereço de memória da região alocada, devemos atribuir esse endereço a um ponteiro.
- Sintaxe (alocação de memória):

**<tipo\_de\_dado> \*variavel = new <tipo\_de\_dado>;**

# Alocação dinâmica de variáveis

- Para alocar dinamicamente uma variável, usamos o operador **new** e para desalocar, usamos o operador **delete**.
  - Como **new** retorna o endereço de memória da região alocada, devemos atribuir esse endereço a um ponteiro.
- Sintaxe (alocação de memória):  
**<tipo\_de\_dado> \*variavel = new <tipo\_de\_dado>;**
- Sintaxe (liberação de memória):  
**delete variavel;**

# Alocação dinâmica de variáveis

- Para alocar dinamicamente uma variável, usamos o operador **new** e para desalocar, usamos o operador **delete**.
  - Como **new** retorna o endereço de memória da região alocada, devemos atribuir esse endereço a um ponteiro.

- Sintaxe (alocação de memória):

```
<tipo_de_dado> *variavel = new <tipo_de_dado>;
```

- Sintaxe (liberação de memória):

```
delete variavel;
```

- Exemplo:

```
int *ptr = new int;
```

```
*ptr = 5;
```

```
delete ptr;
```

```
ptr = nullptr; // evita 'dangling pointer'
```

# Alocação dinâmica de arrays

- Para alocar dinamicamente um array, usamos o operador `new []` e, para desalocar, usamos o operador `delete []`.

# Alocação dinâmica de arrays

- Para alocar dinamicamente um array, usamos o operador `new []` e, para desalocar, usamos o operador `delete []`.
- Sintaxe (alocação de memória):  
`<tipo_de_dado> *ptr = new <tipo_de_dado> [] ;`

# Alocação dinâmica de arrays

- Para alocar dinamicamente um array, usamos o operador `new []` e, para desalocar, usamos o operador `delete []`.
- Sintaxe (alocação de memória):  
`<tipo_de_dado> *ptr = new <tipo_de_dado> [];`
- Sintaxe (liberação de memória):  
`delete [] ptr;`

# Alocação dinâmica de arrays

- Para alocar dinamicamente um array, usamos o operador `new []` e, para desalocar, usamos o operador `delete []`.
- Sintaxe (alocação de memória):  
`<tipo_de_dado> *ptr = new <tipo_de_dado> [];`
- Sintaxe (liberação de memória):  
`delete[] ptr;`
- Exemplo:  
`int *ptr = new int[15];  
ptr[0] = 5;  
delete[] ptr;  
ptr = nullptr; // evita 'dangling pointer'`



# Atividade

Escreva um programa que aloca dinamicamente um array de inteiros de tamanho  $n$ . O valor  $n$  é entrado pelo usuário no início do programa.

- (a) Escreva uma função que recebe como parâmetro o array alocado dinamicamente e preenche esse array com valores inteiros digitados pelo usuário. Sua função deve obedecer o protótipo:  
`void preencheArray(int *A, int n);`
- (b) Escreva uma função que recebe como parâmetro o array alocado dinamicamente e imprime na tela os seus elementos. Sua função deve obedecer o protótipo:  
`void imprimeArray(int *A, int n);`
- (c) Antes do programa acabar, libere a memória alocada dinamicamente.

# Vazamento de memória

- **Vazamentos de memória** acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.

# Vazamento de memória

- **Vazamentos de memória** acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.
  - Quando isso acontece, seu programa não pode excluir a memória alocada dinamicamente porque não sabe mais onde ela está.

# Vazamento de memória

- **Vazamentos de memória** acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.
  - Quando isso acontece, seu programa não pode excluir a memória alocada dinamicamente porque não sabe mais onde ela está.
  - O sistema operacional também não pode usar essa memória, pois considera que ela ainda está sendo usada pelo seu programa.

# Vazamento de memória

- **Vazamentos de memória** acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.
  - Quando isso acontece, seu programa não pode excluir a memória alocada dinamicamente porque não sabe mais onde ela está.
  - O sistema operacional também não pode usar essa memória, pois considera que ela ainda está sendo usada pelo seu programa.
- **Exemplo: que problema pode haver com esse trecho de código?**

```
1 void funcao() {  
2     int *ptr = new int[10];  
3 }
```

## Vazamento de memória

- Há outras formas de vazamento de memória. Por exemplo, um vazamento de memória pode ocorrer se um ponteiro que contém o endereço da memória alocada dinamicamente receber outro valor:

```
1 int v = 5;  
2 int *ptr = new int; // aloca memoria  
3 ptr = &v; // endereco antigo perdido: vazamento de memoria
```

## Vazamento de memória

- Há outras formas de vazamento de memória. Por exemplo, um vazamento de memória pode ocorrer se um ponteiro que contém o endereço da memória alocada dinamicamente receber outro valor:

```
1 int v = 5;  
2 int *ptr = new int; // aloca memoria  
3 ptr = &v; // endereco antigo perdido: vazamento de memoria
```

- Também é possível obter um vazamento de memória via alocação dupla:

```
1 int *ptr = new int;  
2 ptr = new int;
```

# Vazamento de memória

- Há outras formas de vazamento de memória. Por exemplo, um vazamento de memória pode ocorrer se um ponteiro que contém o endereço da memória alocada dinamicamente receber outro valor:

```
1 int v = 5;  
2 int *ptr = new int; // aloca memoria  
3 ptr = &v; // endereco antigo perdido: vazamento de memoria
```

- Também é possível obter um vazamento de memória via alocação dupla:

```
1 int *ptr = new int;  
2 ptr = new int;
```

- Uma forma de evitar esse tipo de vazamento consiste em liberar a memória antes de atribuir novo valor ao ponteiro:

```
1 int *ptr = new int;  
2 delete ptr;  
3 ptr = new int;
```



# Observações adicionais sobre arrays dinâmicos

- Na expressão `int *ptr = new int[10]`, o ponteiro `ptr` aponta para o primeiro elemento do array.

# Observações adicionais sobre arrays dinâmicos

- Na expressão `int *ptr = new int[10]`, o ponteiro `ptr` aponta para o primeiro elemento do array.
  - Deste modo, `ptr` desconhece o tamanho do array e o operador `sizeof` não retorna o tamanho total da memória alocada para `ptr`.

# Observações adicionais sobre arrays dinâmicos

- Na expressão `int *ptr = new int[10]`, o ponteiro `ptr` aponta para o primeiro elemento do array.
  - Deste modo, `ptr` desconhece o tamanho do array e o operador `sizeof` não retorna o tamanho total da memória alocada para `ptr`.
  - Pelo mesmo motivo, não é possível usar um loop `for-each` para processar elementos de um array dinâmico.

# Ponteiros para ponteiros



# Ponteiros para ponteiros

- Um **ponteiro para ponteiro** é um ponteiro que guarda o endereço de outro ponteiro.

# Ponteiros para ponteiros

- Um **ponteiro para ponteiro** é um ponteiro que guarda o endereço de outro ponteiro.
- Em C++, a declaração de um ponteiro para ponteiro criado pelo programador segue esta forma geral:

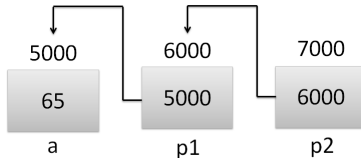
```
tipo_do_ponteiro **nome_do_ponteiro;
```

# Ponteiros para ponteiros

- Um **ponteiro para ponteiro** é um ponteiro que guarda o endereço de outro ponteiro.
- Em C++, a declaração de um ponteiro para ponteiro criado pelo programador segue esta forma geral:

`tipo_do_ponteiro **nome_do_ponteiro;`

```
1 int a = 65;  
2 int *p1 = &a;  
3 int **p2 = &p1;
```



# Ponteiros para ponteiros

- Um ponteiro para um ponteiro pode ser desreferenciado a fim de recuperar o valor apontado. Como esse valor é, ele próprio, um ponteiro, é possível desreferenciá-lo novamente para chegar ao valor subjacente:

```
1 int value = 5;
2
3 int *ptr = &value;
4 std::cout << *ptr; // imprime 5
5
6 int **ptrptr = &ptr;
7 std::cout << **ptrptr; // imprime 5
```



# Arrays de ponteiros

- Um uso comum de ponteiros para ponteiros consiste em alocar dinamicamente um array de ponteiros:

```
1 // aloca um array de ponteiros para inteiros de tamanho 10
2 int **array = new int*[10];
```

# Arrays de ponteiros

- Um uso comum de ponteiros para ponteiros consiste em alocar dinamicamente um array de ponteiros:

```
1 // aloca um array de ponteiros para inteiros de tamanho 10
2 int **array = new int*[10];
```

- Ponteiros para ponteiros também são usados para facilitar a alocação dinâmica de arrays multidimensionais.

# Arrays de ponteiros

- Um uso comum de ponteiros para ponteiros consiste em alocar dinamicamente um array de ponteiros:

```
1 // aloca um array de ponteiros para inteiros de tamanho 10
2 int **array = new int*[10];
```

- Ponteiros para ponteiros também são usados para facilitar a alocação dinâmica de arrays multidimensionais.
  - Primeiro, alocamos um array de ponteiros (como acima).

# Arrays de ponteiros

- Um uso comum de ponteiros para ponteiros consiste em alocar dinamicamente um array de ponteiros:

```
1 // aloca um array de ponteiros para inteiros de tamanho 10
2 int **array = new int*[10];
```

- Ponteiros para ponteiros também são usados para facilitar a alocação dinâmica de arrays multidimensionais.
  - Primeiro, alocamos um array de ponteiros (como acima).
  - Depois, percorremos o array de ponteiros e alocamos um array dinâmico para cada elemento.

# Arrays de ponteiros

- Um uso comum de ponteiros para ponteiros consiste em alocar dinamicamente um array de ponteiros:

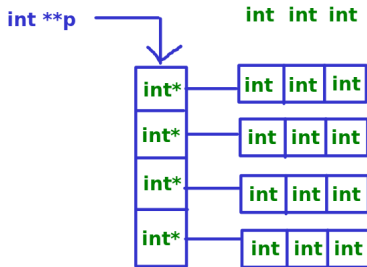
```
1 // aloca um array de ponteiros para inteiros de tamanho 10
2 int **array = new int*[10];
```

- Ponteiros para ponteiros também são usados para facilitar a alocação dinâmica de arrays multidimensionais.
  - Primeiro, alocamos um array de ponteiros (como acima).
  - Depois, percorremos o array de ponteiros e alocamos um array dinâmico para cada elemento.

```
1 // aloca um array de ponteiros para inteiros de tamanho 3
2 // essas sao as linhas (3 linhas)
3 int **array = new int*[3];
4
5 // para cada elemento de array, aloca um array de tamanho 4
6 // estas sao as colunas
7 for (int count = 0; count < 3; ++count)
8     array[count] = new int[4];
```

# Alocação dinâmica de arrays bidimensionais

- **Exercício:** Escreva um programa para criar uma matriz de inteiros dinamicamente usando ponteiro para ponteiro.
- Basicamente, para alocar uma matriz utiliza-se um ponteiro com dois níveis.



- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

# Arrays bidimensionais – Exemplo 1

```
1 #include <iostream> //prog53.cpp
2
3 int main() {
4     int **array = new int*[3];
5     for (int count = 0; count < 3; ++count) {
6         // aloca colunas e inicializa com zeros
7         array[count] = new int[4]{};
8     }
```

# Arrays bidimensionais – Exemplo 1

```
1 #include <iostream> //prog53.cpp
2
3 int main() {
4     int **array = new int*[3];
5     for (int count = 0; count < 3; ++count) {
6         // aloca colunas e inicializa com zeros
7         array[count] = new int[4]{};
8     }
9
10    for (int i = 0; i < 3; i++) { // imprime matriz
11        for (int j = 0; j < 4; j++)
12            std::cout << array[i][j] << " ";
13        std::cout << '\n';
14    }
```



# Arrays bidimensionais – Exemplo 1

```
1 #include <iostream> //prog53.cpp
2
3 int main() {
4     int **array = new int*[3];
5     for (int count = 0; count < 3; ++count) {
6         // aloca colunas e inicializa com zeros
7         array[count] = new int[4]{};
8     }
9
10    for (int i = 0; i < 3; i++) { // imprime matriz
11        for (int j = 0; j < 4; j++)
12            std::cout << array[i][j] << " ";
13        std::cout << '\n';
14    }
15
16    for (int i = 0; i < 3; i++) // liberando a matriz
17        delete[] array[i];
18    delete[] array;
19
20    return 0;
21 }
```

## Arrays bidimensionais – Exemplo 2

```
1  #include <iostream> //prog54.cpp
2
3  void imprime_matriz(int **M, int lin, int col) {
4      for (int i = 0; i < lin; i++) { // imprime matriz
5          for (int j = 0; j < col; j++)
6              std::cout << M[i][j] << " ";
7          std::cout << '\n';
8      }
9  }
10
11 int main() {
12     int **array = new int*[3];
13     for (int lin = 0; lin < 3; ++lin) {
14         // aloca colunas e inicializa com zeros
15         array[lin] = new int[4]{0};
16     }
17
18     imprime_matriz(array, 3, 4);
19
20     for (int i = 0; i < 3; i++) // libera matriz
21         delete[] array[i];
22     delete[] array;
23 }
```

## Aplicações de ponteiros



# Aplicações de ponteiros

- **Passar argumentos por referência.** A passagem por referência serve a dois propósitos:

# Aplicações de ponteiros

- **Passar argumentos por referência.** A passagem por referência serve a dois propósitos:
  - (i) Modificar o valor de uma variável externa dentro de uma função.  
**Exemplo:** trocar os valores de duas variáveis.

# Aplicações de ponteiros

- **Passar argumentos por referência.** A passagem por referência serve a dois propósitos:
  - (i) Modificar o valor de uma variável externa dentro de uma função.  
**Exemplo:** trocar os valores de duas variáveis.
  - (ii) **Eficiência:** podemos passar um dado de tamanho grande (por exemplo, um array) para uma função de forma que não envolva a cópia dos dados.

# Aplicações de ponteiros

- **Passar argumentos por referência.** A passagem por referência serve a dois propósitos:
  - (i) Modificar o valor de uma variável externa dentro de uma função.  
**Exemplo:** trocar os valores de duas variáveis.
  - (ii) **Eficiência:** podemos passar um dado de tamanho grande (por exemplo, um array) para uma função de forma que não envolva a cópia dos dados.
- **Acessar elementos de um array.** O compilador usa internamente ponteiros para acessar os elementos de um array.

# Aplicações de ponteiros

- **Passar argumentos por referência.** A passagem por referência serve a dois propósitos:
  - (i) Modificar o valor de uma variável externa dentro de uma função.  
**Exemplo:** trocar os valores de duas variáveis.
  - (ii) **Eficiência:** podemos passar um dado de tamanho grande (por exemplo, um array) para uma função de forma que não envolva a cópia dos dados.
- **Acessar elementos de um array.** O compilador usa internamente ponteiros para acessar os elementos de um array.
- Permitir que uma função “retorne” vários valores.



# Aplicações de ponteiros

- Programar no nível do sistema, onde os endereços de memória são úteis.

# Aplicações de ponteiros

- **Programar no nível do sistema**, onde os endereços de memória são úteis.
- **Alocação dinâmica de memória**: podemos usar ponteiros para alocar memória dinamicamente. A vantagem da memória alocada dinamicamente é que ela não é excluída até que seja excluída explicitamente.

# Aplicações de ponteiros

- **Programar no nível do sistema**, onde os endereços de memória são úteis.
- **Alocação dinâmica de memória**: podemos usar ponteiros para alocar memória dinamicamente. A vantagem da memória alocada dinamicamente é que ela não é excluída até que seja excluída explicitamente.
- **Implementar diversas estruturas de dados**. Eles serão usados na implementação eficiente de diversas estruturas de dados que veremos durante o curso.

## Referências em C++



# Referências

- Frequentemente precisamos referenciar um objeto
  - sem fazer uma cópia do objeto
- Há dois modos de fazermos isso:
  - **Indiretamente**, por meio de um ponteiro
    - dá o endereço (em memória) do objeto
    - Requer o uso de trabalho extra: derreferenciação
  - **Diretamente**, por meio de uma **referência**
    - age como um **alias**(apelido) para o objeto
    - O usuário interage com a referência como se ela fosse o próprio objeto.

# Referências – Exemplo 1

```
1 #include <iostream> // Referencia01.cpp
2 using namespace std;
3
4 int main() {
5     int x = 45;
6
7     int& ref = x; // criação de uma referência
8
9     cout << ref << endl; // posso usar ref no lugar de x
10
11    ref = 67; // muda o valor de x para 67
12
13    cout << x << endl; // x mudou de valor ----> 67
14 }
```

## Referências – Exemplo 2

```
1 #include <iostream> // Referencia02.cpp
2 using namespace std;
3
4 void troca(int& x, int& y) {
5     int aux = x;
6     x = y;
7     y = aux;
8 }
9
10 int main() {
11     int a = 45;
12     int b = 67;
13
14     troca(a,b);
15
16     cout << "a: " << a << endl; // imprime 67
17     cout << "b: " << b << endl; // imprime 45
18 }
```

## Referências – Exemplo 3

```
1 #include <iostream> // Referencia03.cpp
2 using namespace std;
3
4 struct Ponto {
5     double x = 0, y = 0;
6 };
7
8 void lerPonto(Ponto& p) {
9     cin >> p.x;
10    cin >> p.y;
11 }
12
13 int main() {
14     Ponto ponto;
15     lerPonto(ponto);
16     cout << ponto.x << ", " << ponto.y << endl;
17 }
```



# O que é uma referência em C++?

- Uma variável que guarda um endereço
- Porém com uma interface mais amigável que um ponteiro
  - Uma referência para um objeto oculta a indireção do programador.
- Referências devem ser tipadas
  - checadas pelo compilador
  - assim como ponteiros, elas só podem referenciar o tipo para o qual elas podem apontar.
- Referências devem **obrigatoriamente** referenciar alguma coisa.
  - devem ser inicializadas

# Referências vs Ponteiros

- Depois que uma referência é criada, ela não pode referenciar outro objeto. Já com ponteiros isso é possível.
- Referências não podem ser `null`, enquanto ponteiros podem. Toda referência deve referenciar algum objeto.
  - Por esse motivo, não podemos ter um array de referências por exemplo, já que referências devem ser inicializadas no momento em que são declaradas.
- Não é possível referenciar diretamente um objeto do tipo referência depois que ele é definido. Qualquer ocorrência do seu nome refere-se diretamente ao objeto que ele referencia.

# Quando usar referência?

- **Motivo 1:** evitar fazer uma cópia de objetos ou structs muito grandes ao passá-los como argumentos para funções.
- **Motivo 2:** quando você quiser modificar o valor do parâmetro de entrada de uma função e não houver a necessidade do uso de ponteiros para fazer isso.

# Quando não usar referência?

Funções não devem retornar uma referência para variáveis locais.

```
1 #include <iostream> // Referencia04.cpp
2 using namespace std;
3
4 // Código inválido
5 int& getLocalVariable() {
6     int x { 45 };
7     return x;
8 }
9
10 int main() {
11     cout << getLocalVariable() << endl;
12 }
```

# Quando não usar referência?

Funções não devem retornar uma referência para variáveis locais.

```
1 #include <iostream> // Referencia04.cpp
2 using namespace std;
3
4 // Código inválido
5 int& getLocalVariable() {
6     int x { 45 };
7     return x;
8 }
9
10 int main() {
11     cout << getLocalVariable() << endl;
12 }
```

- Como  $x$  é local, ela é destruída logo depois da função terminar. Logo, ela não existe mais quando a função main executar.

## Referências e a palavra-chave const



## Referências para valores constantes

- É possível declarar uma referência para um valor constante. Isso é feito declarando a referência com a palavra-chave `const`

```
const int apples = 5;  
const int& ref = apples;
```

# Referências para valores constantes

- É possível declarar uma referência para um valor constante. Isso é feito declarando a referência com a palavra-chave `const`

```
const int apples = 5;  
const int& ref = apples;
```

- **Atenção:** Referências para valores não-constantes só podem ser iniciadas com dados(variáveis/valores) não-constantes.

```
const int apples = 5;  
int& ref = apples; // erro
```



# Iniciando referências para valores constantes

- Referências para valores constantes podem ser iniciadas com:
  - variáveis não-constantes
  - variáveis constantes
  - valores temporários (literais, constantes, objetos anônimos)

```
int x = 5;  
const int& ref1 = x; // okay, x é um valor não-const  
  
const int y = 7;  
const int& ref2 = y; // okay, y é um valor const  
  
const int& ref3 = 6; // okay, 6 é uma constante literal
```

# Iniciando referências para valores constantes

- Referências para valores constantes podem ser iniciadas com:
  - variáveis não-constantes
  - variáveis constantes
  - valores temporários (literais, constantes, objetos anônimos)

```
int x = 5;  
const int& ref1 = x; // okay, x é um valor não-const  
  
const int y = 7;  
const int& ref2 = y; // okay, y é um valor const  
  
const int& ref3 = 6; // okay, 6 é uma constante literal
```

- Quando acessado a partir de uma referência para valor constante, um valor é considerado `const` mesmo se a variável original não for `const`.

## Referências para valores temporários

- Referências para valores temporários estendem o tempo de vida do valor referenciado.
  - Geralmente, os valores temporários são destruídos ao final da expressão em que eles são criados.
  - Exemplo:  
`cout << 2+3; //2+3 é avaliado para 5, que é destruído ao final da declaração`

# Referências para valores temporários

- Referências para valores temporários estendem o tempo de vida do valor referenciado.
  - Geralmente, os valores temporários são destruídos ao final da expressão em que eles são criados.
  - **Exemplo:**  
`cout << 2+3; //2+3 é avaliado para 5, que é destruído ao final da declaração`
- Contudo, quando uma referência para valor constante é iniciada com um valor temporário, o tempo de vida do valor temporário é estendido para o tempo de vida da referência. **Exemplo:**

```
int func() {  
    const int& ref = 2+3; //normalmente o resultado de 2+3 é  
    //destruído ao final desta expressão  
    cout << ref; //porém, como ele foi atribuído a uma  
    //referência, o seu tempo de vida é estendido  
    //até aqui, quando a referência morre  
}
```

# Exemplo

```
1 #include <iostream> // Referencia05.cpp
2 using namespace std;
3
4 int soma1 (int& x, int& y) { // non-const
5     return x + y;
6 }
7
8 int soma2 (const int& x, const int& y) {
9     return x + y;
10 }
11
12 int main() {
13     int a = 5;
14     int b = 6;
15
16     cout << soma1(a, b) << endl;
17     cout << soma2(a, b) << endl;
18     cout << soma1(3, 4) << endl; // erro
19     cout << soma2(3, 4) << endl;
20 }
```

# Exercícios



# Exercícios

- (1) Escreva uma função `troca` que receba como entrada duas variáveis inteiras e troque os seus valores. Escreva também uma função `main` que use a função `troca`.

# Exercícios

- (1) Escreva uma função `troca` que receba como entrada duas variáveis inteiras e troque os seus valores. Escreva também uma função `main` que use a função `troca`.
- (2) Escreva uma função `mm` que receba um vetor inteiro  $A$  com  $n$  elementos e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nestas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função `main` que use a função `mm`.



# Exercícios

- (1) Escreva uma função `troca` que receba como entrada duas variáveis inteiras e troque os seus valores. Escreva também uma função `main` que use a função `troca`.
- (2) Escreva uma função `mm` que receba um vetor inteiro  $A$  com  $n$  elementos e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nestas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função `main` que use a função `mm`.
- (3) Escreva um programa que leia um inteiro  $n$  seguido de  $n$  números inteiros e imprima esses  $n$  números em ordem invertida (primeiro o último, depois o penúltimo, etc.) O seu programa não deve impor quaisquer restrições ao valor de  $n$ .

- (4) Faça uma função **MAX** que recebe como entrada um inteiro  $n$ , uma matriz inteira  $A_{n \times n}$  e devolve três inteiros:  $k$ ,  $l$  e  $c$ , tal que
- $k$  é o maior elemento de  $A$  e é igual a  $A[l][c]$ .

Se o elemento máximo ocorrer mais de uma vez, indique em  $l$  e  $c$  qualquer uma das possíveis posições. Use ponteiros para os argumentos.

FIM

