

# Primer de C++

Linguagem de Programação Específica para IA - Lux.AI

INSTITUIÇÃO EXECUTORA



COORDENADORA



APOIO



# Tópicos

## → Conceitos Básicos

- ◆ Comentários, Include, Using namespace, Função main, Cout, Return

## → Variáveis

- ◆ Declaração de variáveis, Tipos de variáveis, Constantes

## → Input

- ◆ Cin

## → Tipos de Dados

- ◆ Principais tipos em C++, Float vs. double

## → Operadores

- ◆ Tipos de operadores, Operadores aritméticos, Operadores de atribuição, Operadores de comparação, Operadores lógicos

## → Condicionais

- ◆ Tipos de estruturas condicionais, If-else, Operador ternário, Switch, Break, Default

# Tópicos

→ Referências

→ Ponteiros

→ Arrays

- ◆ Acessando elementos de um array, Omitir o tamanho de um array

→ Strings

- ◆ Concatenação, Append, Length, Acessando elementos de uma string, String como input

→ Gerenciamento Dinâmico de Memória

- ◆ Malloc, Calloc, Realloc, Free, New, Delete

→ Loops

- ◆ While, Do-while, For, “For-each”

→ Estruturas

→ Funções

- ◆ Parâmetros e argumentos, Argumentos argc e argv, Valores de retorno, Referenciado parâmetros

# Tópicos

- Sobrecarga de Funções
- Recursão
- Programação Orientada a Objetos
- Classes e Objetos
  - ◆ Classes, Objetos, Métodos de classe, Construtores, Especificadores de acesso
- Encapsulamento
- Herança
  - ◆ Acesso em uma herança

# Conceitos Básicos

# Conceitos básicos

- Alguns elementos são inerentes à linguagem C++ e estão presentes até nos mais simples códigos, como ilustrado pelo código exemplo abaixo.
- Vamos analisar cada um dos elementos presentes neste código.

```
// Hello world C++  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    // print "Hello world!"  
    cout << "Hello world!";  
  
    return 0;  
}
```

Exemplo de código básico em C++

# Comentários

- Comentários são usados para adicionar informações sobre o código ou omitir um trecho do código na execução.
- Comentários podem ser feitos em uma ou várias linhas.
  - ◆ Para comentar uma linha usamos um par de barras (//), qualquer coisa entre as barras e o final da linha é considerado um comentário. Dessa forma, é possível comentar apenas parte de uma linha se necessário.
  - ◆ Para comentar várias linhas usamos uma barra e um asterisco (/\*). O bloco de comentário começa com “/\*” e termina em “\*/”, qualquer texto entre esses operadores é considerado parte do comentário.

# Comentários

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

int main() {
    // print "Hello world!"
    cout << "Hello";
    cout << " world"; // comentário
    cout << "!\n";

    return 0;
}
```

Exemplos de comentários de linha única

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

int main() {
    /* print "Hello world!"
    cout << "Hello";
    cout << " world"; */
    cout << "!\n";

    return 0;
}
```

Exemplo de comentário com várias linhas



# Comentários

```
// Hello world C++

#include <iostream>
using namespace std;

int main() {
    // print "Hello world!"
    cout << "Hello";
    cout << " world";
    cout << "!\n";

    return 0;
}
```

Código padrão em C++

```
● rsb7@ug4c30:~$ g++ -o test test.cpp
● rsb7@ug4c30:~$ ./test
  Hello world!
○ rsb7@ug4c30:~$ █
```

Saída do código no terminal

# Comentários

```
// Hello world C++

#include <iostream>
using namespace std;

int main() {
    // print "Hello world!"
    cout << "Hello";
    // cout << " world";
    cout << "!\n";

    return 0;
}
```

Código com uma linha comentada

```
● rsb7@ug4c30:~$ g++ -o test test.cpp
● rsb7@ug4c30:~$ ./test
Hello!
○ rsb7@ug4c30:~$ █
```

Perceba que a execução do programa ignorou o trecho de código comentado

# Include

- ➔ *Include* é a diretiva usada para importar bibliotecas em C++.
- ➔ Lembre-se sempre de incluir os arquivos header necessários para execução do programa. Por exemplo, “#include <iostream>” é necessário para usar “std::cin” e “std::cout”.

# Using namespace

- “using namespace” é usado para importar a entidade do *namespace* para o domínio atual do programa.
- Importar um *namespace* significa que não necessitamos especificar que um objeto ou variável pertence àquele *namespace*.
- Uma outra alternativa é utilizar o operador de escopo (::) sempre que um tipo ou variável forem declarados, por exemplo, “std::cout”.

# Using namespace

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

int main() {
    // print "Hello world!"
    string hello = "Hello world!";
    cout << hello;

    return 0;
}
```

Código utilizando “using namespace”

```
#include <iostream>

int main() {
    // print "Hello world!"
    std::string hello = "Hello world!";
    std::cout << hello;

    return 0;
}
```

Código sem “using namespace”, note o uso de “std::” na declaração do objeto *cout* e do tipo *string*, especificando que eles pertencem ao *namespace* “std” da biblioteca “iostream”

# Função main

- A função *main()* é o ponto inicial de todo programa em C++, não importa onde esteja localizada no código, a execução sempre iniciará por ela. Falaremos mais sobre funções depois.

# Cout

- *Cout* é um objeto da biblioteca *iostream* usado junto do operador “<<” para printar texto no terminal.
- O *cout* é usado em C++ para imprimir mensagens de texto no terminal, mas ele pode, da mesma forma, ser usado para imprimir variáveis. É possível ainda intercalar mensagens de texto e variáveis na declaração, separando-os com o operador “<<”.
- A função do *cout* é apenas printar texto, o que significa que ele não inicia uma nova linha automaticamente. Para isso, é necessário utilizar o caractere “\n” ou o manipulador “endl”.

# Cout

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

int main() {
    string nome = "Fulano";
    int idade = 25;

    cout << "Nome: " << nome << endl
         << "Idade: " << idade << endl;

    return 0;
}
```

*Cout* recebendo parâmetros variados

```
● rsb7@ug4c30:~$ g++ -o test test.cpp
● rsb7@ug4c30:~$ ./test
Nome: Fulano
Idade: 25
○ rsb7@ug4c30:~$ █
```

Saída do código



# Cout

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

int main() {
    string nome = "Fulano";
    int idade = 25;

    cout << "Nome: " << nome << "\n"
         << "Idade: " << idade << "\n";

    return 0;
}
```

Código semelhante ao exemplo anterior, mas fazendo uso de “\n” para iniciar uma nova linha

```
● rsb7@ug4c30:~$ g++ -o test test.cpp
● rsb7@ug4c30:~$ ./test
Nome: Fulano
Idade: 25
○ rsb7@ug4c30:~$
```

A saída do programa é essencialmente idêntica à do código usando *endl*

# Return

- “return” é usado para retornar o resultado de uma função e indica o encerramento dela.
- “return 0” é usado como retorno padrão da função *main*, significando que a execução da função foi bem-sucedida.

# Variáveis

# Declaração de variáveis

- Em C++, variáveis devem ser declaradas antes de serem utilizadas, isto é, o tipo da variável deve ser especificado antes de atribuí-la um valor.
- Declarações típicas de variáveis em C++ seguem os seguintes formatos:

```
// declaração e inicialização separadas  
int a;  
a = 10;  
  
// inicialização no momento da declaração  
int b = 10;
```

A inicialização da variável pode ser feita depois da sua declaração

# Declaração de variáveis

→ Há algumas convenções para declaração de variáveis:

- ◆ o nome deve conter apenas letras, dígitos e underline.
- ◆ o nome é sensível a letras maiúsculas. (ex: myvar ≠ myVar)
- ◆ o nome não deve conter nenhum espaço em branco ou caracteres especiais. (ex: #, \$, %, &, \*)
- ◆ o nome deve começar com uma letra ou underline.
- ◆ não se pode usar palavras-chave de C++ (ex: float, double, class) como nome.

# Tipos de variáveis

- Há três tipos de variáveis em C++, de acordo com o escopo delas:
- ◆ **Variáveis locais:** uma variável definida em um bloco ou função, o escopo dessas variáveis existe apenas no bloco em que são declaradas.
- ◆ **Variáveis estáticas:** também conhecidas como variáveis de classe, são criadas no início do programa e destruídas com seu encerramento.
- ◆ **Variáveis instanciadas:** são variáveis não estáticas declaradas em uma classe fora de qualquer bloco ou função.

# Tipos de variáveis

## Type of variables in C++

```
class GFG {  
    public :  
        static int a ;  
        int b ;  
    public :  
        func ()  
        {  
            int c ;  
        }  
};
```

The diagram illustrates the types of variables in the provided C++ code. It shows three variables: `static int a ;`, `int b ;`, and `int c ;`. Each variable is enclosed in a box, and an arrow points from the box to its corresponding label: `Static Variable` for `a`, `Instance Variable` for `b`, and `Local Variable` for `c`.

(fonte:

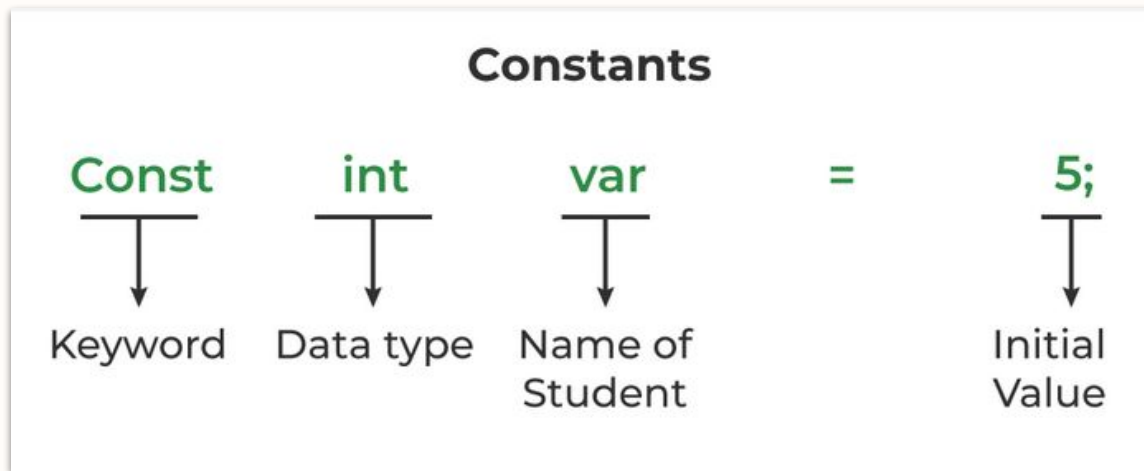
<https://www.geeksforgeeks.org/cpp-variables/?ref=lbp>)

# Constantes

- Constantes são variáveis que não podem ser alteradas no programa, servindo apenas para leitura.
- Para definir uma variável como constante, usamos a palavra-chave *const* na declaração da variável.
- Vale ressaltar que o valor de uma constante deve ser atribuído no momento de sua declaração.



# Constantes



Sintaxe de declaração de uma constante  
(fonte: <https://www.geeksforgeeks.org/constants-in-c/?ref=lbp>)

# Input

# Cin

- De maneira semelhante ao *cout*, o *cin* é usado junto ao operador “>>” para receber as entradas do usuário.
- O *cin* também permite a atribuição de múltiplas variáveis, desde que estejam separadas por um operador “>>”.

# Cin

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

int main() {

    int a;
    int b;

    cout << "Digite dois números: ";
    cin >> a >> b;
    cout << "Soma: " << a + b << endl;

    return 0;
}
```

*Cin* fazendo atribuição de mais de uma variável

# Tipos de Dados

# Principais tipos de C++

- Os principais tipos de dados em C++ são:
  - ◆ **int**: armazena números inteiros, sem casas decimais.
  - ◆ **float**: armazena pontos flutuantes, com casas decimais (4 bytes).
  - ◆ **double**: armazena pontos flutuantes, com casa decimais (8 bytes).
  - ◆ **char**: armazena um caracter único ou valor ASCII. Valores *char* são indicados por aspas simples.
  - ◆ **string**: armazena uma sequência de caracteres. Valores *string* são indicados por aspas duplas.
  - ◆ **bool**: armazena valores booleanos (*true* ou *false*).

## Float vs. double

- Enquanto *float* e *double* essencialmente definem o mesmo tipo de variável, eles se distinguem pela precisão dos valores que conseguem armazenar: a precisão do *float* é de seis ou sete casas decimais, enquanto a precisão do *double* é de cerca de 15 casas decimais.

# Operadores



# Tipos de operadores

→ C++ classifica operadores nos seguintes grupos:

- ◆ Operadores aritméticos
- ◆ Operadores de atribuição
- ◆ Operadores de comparação
- ◆ Operadores lógicos

# Operadores aritméticos

→ Operadores aritméticos são usados para executar operações matemáticas:

- ◆ **Adição (+):** soma dois valores.
- ◆ **Subtração (-):** subtrai um valor de outro.
- ◆ **Multiplicação (\*):** multiplica dois valores.
- ◆ **Divisão (/):** divide um valor por outro.
- ◆ **Módulo (%):** retorna o resto da divisão.
- ◆ **Incremento (++):** incrementa o valor de uma variável em 1.
- ◆ **Decremento (--):** decrementa o valor de uma variável em 1.

# Operadores aritméticos

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

int main() {
    int a = 3;

    a = a + 5; // 3 + 5 = 8
    a = a - 4; // 8 - 4 = 4
    a = a * 3; // 4 * 3 = 12
    a = a / 2; // 12 / 2 = 6
    a = a % 6; // 6 mod 6 = 0
    a++;      // 0 + 1 = 1
    a--;      // 1 - 1 = 0

    cout << "a = " << a << endl; // a = 0

    return 0;
}
```

Exemplo de código demonstrando o uso dos operadores aritméticos

# Operadores de atribuição

→ Operadores de atribuição são usados para atribuir valores às variáveis:

- ◆ **Atribuição (=):** atribui um valor a uma variável.
- ◆ **Atribuição de adição (+=):** atribui o resultado da soma à variável.
- ◆ **Atribuição de subtração (-=):** atribui o resultado da subtração à variável.
- ◆ **Atribuição de multiplicação (\*=):** atribui o resultado da multiplicação à variável.
- ◆ **Atribuição de divisão (/=):** atribui o resultado da divisão à variável.
- ◆ **Atribuição de módulo (%=):** atribui o resultado do módulo à variável.

# Operadores de atribuição

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

int main() {
    int a = 3;

    a += 5; // 3 + 5 = 8
    a -= 4; // 8 - 4 = 4
    a *= 3; // 4 * 3 = 12
    a /= 2; // 12 / 2 = 6
    a %= 6; // 6 % 6 = 0
    a++; // 0 + 1 = 1
    a--; // 1 - 1 = 0

    cout << a << endl; // a = 0

    return 0;
}
```

Note que a lógica do programa permanece a mesma, mas a declaração das operações é mais simplificada

# Operadores de comparação

→ Operadores de comparação são usados para comparar valores ou variáveis:

- ◆ Igual a (==)
- ◆ Diferente de (!=)
- ◆ Maior que (>)
- ◆ Menor que (<)
- ◆ Maior que ou igual a (>=)
- ◆ Menor que ou igual a (<=)

\* O resultado de uma comparação é um valor booleano: *true* (1) ou *false* (0).

# Operadores lógicos

- Operadores lógicos são usados para determinar a lógica entre valores ou variáveis:
- ◆ **And (&&):** retorna *true* se as duas expressões forem verdadeiras.
  - ◆ **Or (||):** retorna *true* se uma das expressões for verdadeira.
  - ◆ **Not (!):** inverte o resultado, retorna *false* se o resultado for *true*.

# Condicionais



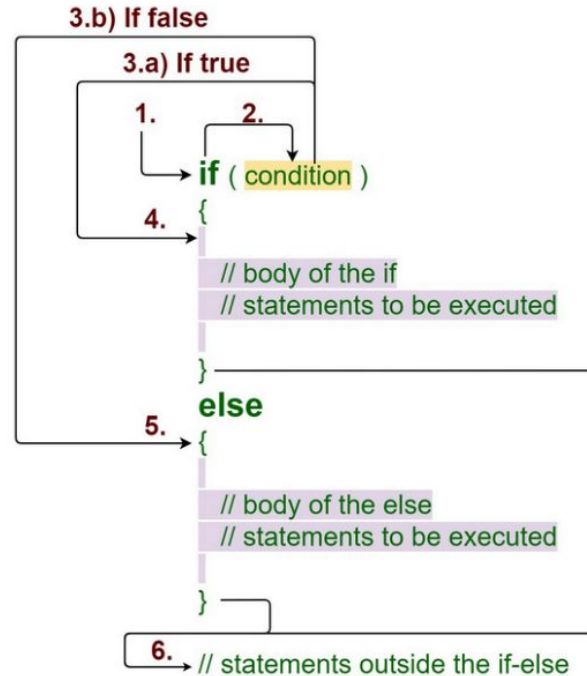
# Tipos de estruturas condicionais

→ C++ tem as seguintes estruturas condicionais:

- ◆ **if:** especifica um bloco de código a ser executado se uma condição específica for verdadeira.
- ◆ **else:** especifica um bloco de código a ser executado se a mesma condição for falsa.
- ◆ **else if:** especifica uma nova condição de teste se a primeira for falsa.
- ◆ **switch:** especifica muitos blocos alternativos de código a serem executados.

# If-else

## If - else statement



Esquema da sintaxe e funcionamento de uma estrutura condicional if-else

(fonte:

<https://www.geeksforgeeks.org/c-c-if-else-statement-with-examples/>)

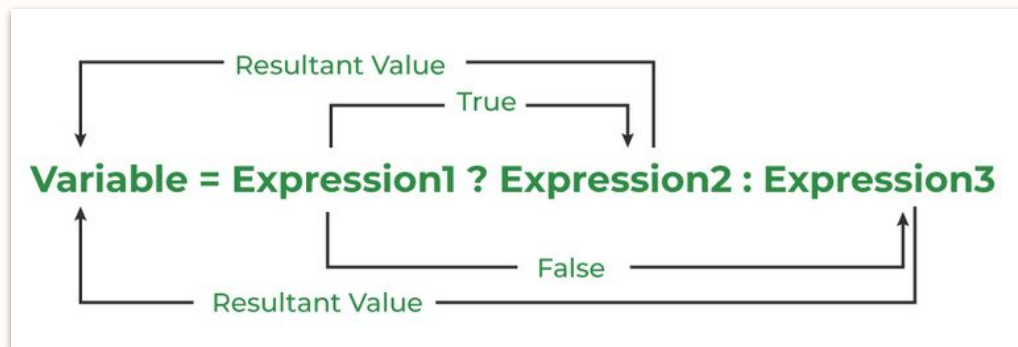
# If-else

```
int main() {  
  
    int n = 0;  
  
    // se n for um número Positivo, essa condição  
    // será verdadeira  
    if (n > 0) {  
        cout << "Positive" << endl;  
    }  
  
    // se n for um número Negativo, essa condição  
    // será verdadeira  
    else if (n < 0) {  
        cout << "Negative" << endl;  
    }  
  
    // se n não for Positivo nem Negativo, essa  
    // condição será verdadeira  
    else {  
        cout << "Zero" << endl;  
    }  
    return 0;  
}
```

Exemplo de código demonstrando a lógica condicional de funcionamento do if-else

# Operador ternário

- O operador ternário é uma versão abreviada do if-else, que consiste de três operandos e pode ser usado para substituir múltiplas linhas de código por uma única linha. Geralmente é usado para substituir instâncias mais simples de if-else.



Sintaxe de declaração do operador ternário

(fonte: <https://www.geeksforgeeks.org/conditional-or-ternary-operator-in-c/>)

# Operador ternário

- A seguir, está o nosso código condicional mostrado anteriormente, mas implementado com a estrutura condicional do operador ternário. Note quantas linhas de código essa implementação nos permitiu economizar para o programa.

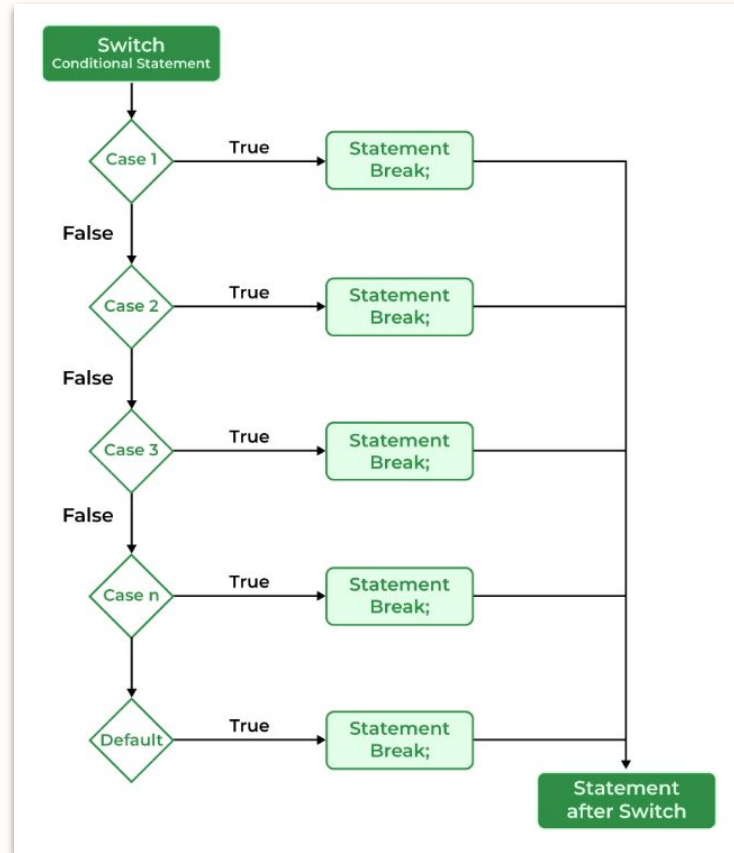
```
int main() {  
    int n = 0;  
    string result;  
  
    result = (n > 0)? "Positive" : (n < 0)? "Negative" : "Zero";  
    cout << result << endl;  
  
    return 0;  
}
```

Exemplo de código condicional implementando a estrutura do operador ternário

# Switch

- O *switch* é usado quando se precisa selecionar um entre vários blocos de código alternativos para ser executado. Ele funciona da seguinte forma:
  - ◆ A expressão do *switch* é avaliada uma vez;
  - ◆ O valor da expressão é comparado com os valores de cada *case*;
  - ◆ Se há uma correspondência, o bloco de código associado é executado.
- As palavras-chave *break* e *default* são opcionais, mas falaremos mais sobre elas.

# Switch



Esquema de fluxo de execução de um condicional *switch*  
(fonte:

<https://www.geeksforgeeks.org/switch-statement-in-cpp/>)

# Switch

```
char var = 'b';

switch (var) {
    case 'a':
        cout << "Caso 1 é correspondente." << endl;
        break;

    case 'b':
        cout << "Caso 2 é correspondente." << endl;
        break;

    case 'c':
        cout << "Caso 3 é correspondente." << endl;
        break;

    default:
        cout << "Caso padrão é correspondente." << endl;
        break;
}
```

```
● rsb7@pc076-ubu:~$ g++ -o test test.cpp
● rsb7@pc076-ubu:~$ ./test
  Caso 2 é correspondente.
○ rsb7@pc076-ubu:~$
```

Como var = 'b', o *switch* pulará o *case* 'a' e executará o bloco de código associado ao *case* 'b'

Exemplo de código demonstrando a estrutura de declaração do operador condicional *switch*



# Break

- Quando o programa alcança um *break*, ele abandona o bloco de código em execução.
- O *break* interromperá a execução de mais código e teste de casos dentro do bloco, pois quando uma correspondência é encontrada não há necessidade de mais testes.
- Isso economiza algum tempo de execução, uma vez que ele “ignora” os outros blocos de código do *switch*.

# Default

- O *default* especifica um bloco de código a ser executado se não houver nenhuma correspondência.
- No exemplo apresentado, o bloco *default* executará quando o valor de “var” não corresponder a nenhum dos outros valores especificados (a, b e c).

# Referências

# Referências

- Uma variável de referência “aponta” para outra variável declarada no programa e é criada com o operador “&”.

```
string comida = "Tapioca";  
string &prato = comida;  
  
cout << comida << endl  
    << prato << endl;
```

A variável “prato” é declarada fazendo referência à variável “comida”

```
● rsb7@ug4c29:~$ g++ -o test test.cpp  
● rsb7@ug4c29:~$ ./test  
Tapioca  
Tapioca  
○ rsb7@ug4c29:~$
```

As variáveis “comida” e “prato” armazenam o mesmo valor “Tapioca”

# Referências

- O operador de referência (&) também pode ser usado para acessar o endereço de memória de uma variável, que é onde a variável está armazenada no sistema. Para acessá-lo, use o operador antes do nome da variável e o resultado será seu endereço.

```
string comida = "Tapioca";  
string &prato = comida;  
  
cout << &comida << endl  
      << &prato << endl;
```

O operador "&" pode tanto ser usado para declarar uma variável de referência quanto para acessar o endereço de uma variável

```
• rsb7@ug4c29:~$ g++ -o test test.cpp  
• rsb7@ug4c29:~$ ./test  
0x7fffd16dac40  
0x7fffd16dac40  
○ rsb7@ug4c29:~$
```

Ambas as variáveis "comida" e "prato" apontam para o mesmo endereço em memória

**Ponteiros**

# Ponteiros

- Um ponteiro é uma variável que armazena um endereço de memória como valor.
- Um ponteiro aponta para um tipo de dado igual ao seu e é criado com o operador “\*”. O endereço da variável atribuída é armazenado no ponteiro.
- Há três formas de declarar um ponteiro:
  - ◆ `type* ptr`
  - ◆ `type *ptr`
  - ◆ `type * ptr`

# Ponteiros

- Um ponteiro também pode ser usado para acessar o valor armazenado pela variável em vez do seu endereço. Para isso, é usado o operador de desreferência (\*).
- \*\* Não confundir o operador de desreferência com o operador utilizado na declaração de ponteiros.
- Também é possível utilizar a referência feita pelo ponteiro à variável para alterá-la indiretamente. Essa é uma característica muito importante quando estamos trabalhando com funções em C++, que veremos mais adiante.



# Ponteiros

```
string comida = "Tapioca";

// declarando um ponteiro "ptr" que armazenará
// o endereço de memória da variável "comida"
string* ptr = &comida;

cout << comida << endl; // output: "Tapioca"
cout << &comida << endl; // output: 7ffe23b2d4c0

// "ptr" armazena o endereço de "comida"
cout << ptr << endl;    // output: 7ffe23b2d4c0

// podemos "resgatar" o valor da variável a
// partir do endereço armazenado no ponteiro,
// usando o operador "*"
cout << *ptr << endl;   // output: "Tapioca"
```

Usando o operador "\*" podemos obter o valor armazenado em uma variável partindo de um ponteiro com o endereço da variável armazenado

# Arrays

# Arrays

- Arrays são usados para armazenar múltiplos dados em uma única variável em vez de declarar variáveis para cada dado.
- Para declarar um array, deve-se especificar o seu tamanho, isto é, a quantidade de elementos armazenados, junto ao seu nome e tipo. No caso abaixo, temos dois arrays com 5 espaços de elementos.

```
// array de inteiros  
int arr_int[5];  
  
// array de caracteres  
char arr_char[5];
```

# Acessando elementos de um array

- Os elementos de um array podem ser acessados referenciando o índice da sua posição dentro dos colchetes.
- É importante destacar que índices de arrays começam no 0. Dessa forma, [0] é o primeiro elemento, [1] é o segundo e assim por diante.

# Acessando elementos de um array

```
// array de inteiros
int arr_int[5] = {3, 2, 1, 0, 0};

// acessando elementos do array
cout << "Primeiro elemento: " << arr_int[0] << endl;
cout << "Segundo elemento: " << arr_int[1] << endl;
cout << "Terceiro elemento: " << arr_int[2] << endl;
```

Exemplo de código acessando as posições de um array

```
• rsb7@ug4c30:~$ g++ -o test test.cpp
• rsb7@ug4c30:~$ ./test
Primeiro elemento: 3
Segundo elemento: 2
Terceiro elemento: 1
○ rsb7@ug4c30:~$
```

Saída do código

## Omitir o tamanho de um array

- Outra forma alternativa de declarar um array é omitir o seu tamanho e apresentar apenas os elementos armazenados nele. O compilador consegue determinar o tamanho do array fazendo a contagem do número de elementos declarados.
- Apesar de ser uma opção possível, esse tipo de declaração não é recomendada, pois pode gerar erros no programa.

# Omitir o tamanho de um array

```
// array de inteiros
int arr_int[] = {3, 2, 1, 0, 0};

// acessando elementos do array
cout << "Primeiro elemento: " << arr_int[0] << endl;
cout << "Quarto elemento: " << arr_int[3] << endl;
cout << "Quinto elemento: " << arr_int[4] << endl;
```

Declarando um array sem tamanho especificado

```
• rsb7@ug4c30:~$ g++ -o test test.cpp
• rsb7@ug4c30:~$ ./test
Primeiro elemento: 3
Quarto elemento: 0
Quinto elemento: 0
○ rsb7@ug4c30:~$ █
```

A atribuição das posições dos elementos  
no array permanece a mesma

# Strings



# Strings

- Uma variável *string* armazena uma sequência de caracteres delimitada por aspas duplas.
- Para trabalhar com strings, utilizamos a biblioteca `<string>` de C++.
- O tipo *string* em C++ deve ser acompanhado do prefixo “std::” que indica o *namespace* ao qual o tipo pertence. Mas esse prefixo pode ser omitido com o uso de “using namespace std”.

# Concatenação

- O operador “+” pode ser usado para concatenar duas strings, formando uma nova string.

```
string nome = "Allan";  
string sobrenome = "Turing";  
  
string nome_compl = nome + " " + sobrenome;  
cout << nome_compl << endl;
```

Usando o operador “+” para concatenar as strings nome e sobrenome, formando a string nome\_compl

```
rsb7@ug3c27:~$ g++ -o test test.cpp  
rsb7@ug3c27:~$ ./test  
Allan Turing  
rsb7@ug3c27:~$
```

String nome\_compl printada

# Append

- Outra forma de concatenar strings é utilizar a função *append*, um método interno à classe string.

```
string nome = "Allan";  
string sobrenome = " Turing";  
  
string nome_compl = nome.append(sobrenome);  
cout << nome_compl << endl;
```

Usando a função *append* para concatenar as strings nome e sobrenome, formando a string nome\_compl

```
rsb7@ug3c27:~$ g++ -o test test.cpp  
rsb7@ug3c27:~$ ./test  
Allan Turing  
rsb7@ug3c27:~$
```

String nome\_compl printada

# Length

→ *Length* é um método interno à classe string que retorna o tamanho da string chamando a função.

```
string msg = "vakdJndknBmojFbGkIasV";  
cout << "Tamanho da string msg: " << msg.length() << endl;
```

A string “msg” chama a função *length* que retornará seu tamanho

```
● rsb7@ug3c27:~$ g++ -o test test.cpp  
● rsb7@ug3c27:~$ ./test  
Tamanho da string msg: 21  
○ rsb7@ug3c27:~$
```

A função *length* retorna a extensão total da string

## Acessando elementos de uma string

- Sintaticamente, uma string nada mais é do que um array de caracteres. Por isso, é possível acessar elementos de uma string da mesma forma que acessamos os elementos de um array numérico.

# Acessando elementos de uma string

```
string msg = "vakdJndknBmojFbGkIasV";

cout << "Primeiro elemento da string: " << msg[0] << endl;
cout << "Terceiro elemento da string: " << msg[2] << endl;
cout << "Quinto elemento da string: " << msg[4] << endl;
cout << "Décimo elemento da string: " << msg[9] << endl;
```

Exemplo de código acessando os elementos da string

```
• rsb7@ug3c27:~$ g++ -o test test.cpp
• rsb7@ug3c27:~$ ./test
Primeiro elemento da string: v
Terceiro elemento da string: k
Quinto elemento da string: J
Décimo elemento da string: B
• rsb7@ug3c27:~$ █
```

Cada caracter é referenciado pelo índice de sua posição relativa na string

# String como input

- É possível utilizar *cin* para ler strings como entrada do programa. No entanto, ele encerra a leitura ao se deparar com um espaço em branco, limitando a leitura a apenas uma palavra.
- Por isso, quando trabalhamos com strings, geralmente utilizamos a função *getline()*, usando *cin* e a variável string como parâmetros, para ler uma string digitada pelo usuário.

# String como input

```
cout << "Digite uma mensagem: ";  
cin >> msg;  
cout << "Mensagem: " + msg << endl;
```

Leitura de string utilizando apenas *cin*

```
• rsb7@ug3c27:~$ g++ -o test test.cpp  
• rsb7@ug3c27:~$ ./test  
  Digite uma mensagem: hello world  
  Mensagem: hello  
○ rsb7@ug3c27:~$ █
```

A leitura da string é interrompida ao encontrar um espaço

```
cout << "Digite uma mensagem: ";  
getline(cin, msg);  
cout << "Mensagem: " + msg << endl;
```

Leitura de string utilizando a função *getline*

```
• rsb7@ug3c27:~$ g++ -o test test.cpp  
• rsb7@ug3c27:~$ ./test  
  Digite uma mensagem: hello world  
  Mensagem: hello world  
○ rsb7@ug3c27:~$ █
```

A leitura da string é feita até atingir o fim da linha



# **Gerenciamento Dinâmico de Memória**

# Gerenciamento dinâmico de memória

- Algumas vezes podem haver situações em que é necessário o redimensionamento de um array, como no caso de precisarmos trabalhar com mais elementos do que o tamanho do array consegue comportar. Para isso, utilizamos **alocação dinâmica de memória**.
- Alocução dinâmica de memória pode ser definida como um procedimento em que o tamanho de uma estrutura de dados é alterado durante a execução do programa.

# Gerenciamento dinâmico de memória

- C++ oferece duas formas principais de fazer alocação dinâmica de memória:
  - ◆ as funções *malloc*, *calloc*, *free* e *realloc*
  - ◆ os operadores *new* e *delete*
- Os operadores *new* e *delete* possuem essencialmente a mesma funcionalidade que as funções *malloc* e *free*, com a diferença de que, além das operações de gerenciamento de memória, eles também chamam o construtor e destrutor de uma classe, respectivamente.

# Malloc

- A função *malloc()* é usada para alocar dinamicamente um único bloco de memória com o tamanho especificado.
- Ela retorna um ponteiro *void* que pode ser convertido em um ponteiro de qualquer outro tipo.

```
// sintaxe do malloc
// ptr = (tipo*) malloc(tamanho)

ptr = (int*) malloc(100 * sizeof(int));
// como o tamanho de um int é 4 bytes, essa declaração
// alocará 400 bytes de memória e o ponteiro ptr contém
// o endereço do primeiro byte da memória alocada
```

# Calloc

- A função *calloc()* é usada para alocar dinamicamente o número especificado de blocos do tipo especificado.
- Ela inicializa cada bloco com valor padrão “0” e recebe dois parâmetros em comparação com o *malloc()*.

```
// sintaxe do malloc
// ptr = (tipo*) malloc(tamanho)

ptr = (float*) calloc(25, sizeof(float));
// essa declaração aloca um espaço de memória para 25
// elementos, cada um com o tamanho de um float
```

# Realloc

- A função *realloc()* é usada para mudar a alocação dinâmica de memória de uma memória já alocada, ou seja, se a memória for insuficiente, *realloc* pode ser usado para realocá-la.
- A realocação mantém os valores já presentes e inicializa novos blocos com valores de lixo.

```
// sintaxe do realloc
// ptr = realloc(ptr, novoTamanho)

int* ptr = (int*) malloc(5 * sizeof(int));
ptr = realloc(ptr, 10 * sizeof(int));
// o malloc aloca dinamicamente um bloco de memória de
// 20 bytes para ptr, que é alterado para 40 bytes pelo
// realloc
```

## *Free*

- A função *free()* é usada para desalocar a memória alocada dinamicamente.
- Quando alocamos memória usando as funções *malloc()* e *calloc()*, ela não se libera sozinha. Por isso, usamos *free()* para liberar a memória alocada e evitar o desperdício de memória.

# New

- O operador *new* denota um pedido de alocação de memória.
- Se houver memória suficiente disponível, o operador *a* inicializará e retornará o endereço da memória recém alocada e inicializada para o ponteiro.

```
// Sintaxe do operador new
// tipo ponteiro = new tipo

// Formas de inicializar um ponteiro com new
int* ptr1 = NULL;
ptr1 = new int;

int* ptr2 = new int;
```

O ponteiro inicializado pelo operador *new* pode ser de qualquer tipo integrado no sistema ou declarado pelo usuário, incluindo estruturas e classes



# New

- O operador *new* também pode ser usado para alocar um bloco de memória (um array) do tipo de dados referenciado pelo ponteiro com o tamanho especificado na chamada.

```
// Sintaxe do operador new  
// tipo ponteiro = new tipo[tamanho]  
  
// Alocando um array de inteiros usando  
// o operador new  
int* ptr = new int[10];
```

A instrução aloca memória dinamicamente para 10 inteiros e retorna um ponteiro apontando para o primeiro elemento do bloco

# Delete

- O operador *delete* é a ferramenta fornecida em C++ para desalocar a memória alocada dinamicamente pelo operador *new*.
- O operador pode ser usado para desalocar tanto elementos individuais quanto blocos de memória alocados.

```
// Sintaxe do operador delete
// delete ponteiro
delete ptr;

// Desalocando um bloco de memória alocado
// dinamicamente por um operador new
delete[] ptr;
```

O operador *delete[]* libera todo o bloco de memória alocado apontado pelo ponteiro indicado

# Loops

# Loops

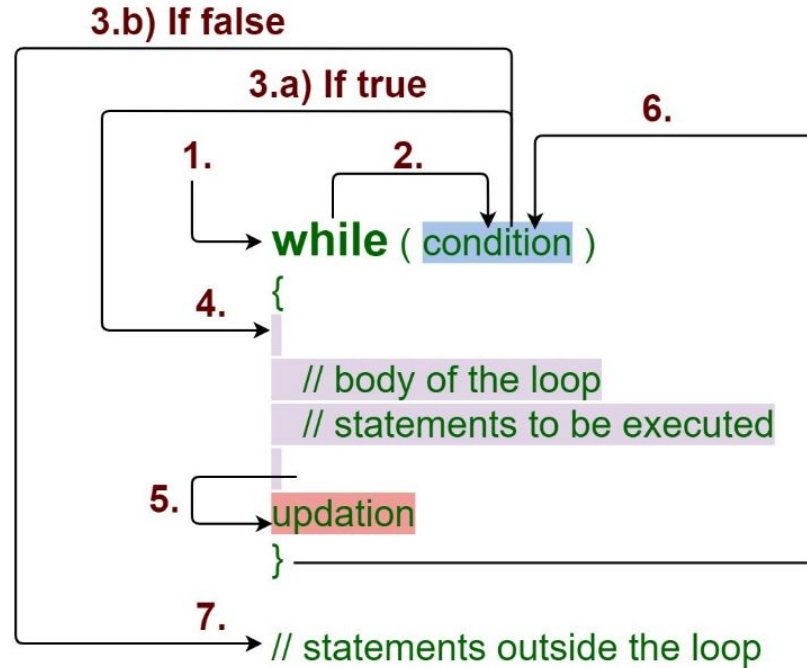
- Loops permitem a execução de um bloco de código inúmeras vezes, até que uma condição seja alcançada.
- Eles são úteis porque economizam tempo, reduzem erros e deixam o código mais legível.

# While

- O loop *while* repete a execução de um bloco de código enquanto a condição estabelecida for verdadeira.
- É preciso tomar cuidado e se certificar de que o bloco de código do *while* esteja executando de acordo, de modo a evitar que o programa fique preso em um loop infinito.

# While

## While Loop



Sintaxe da estrutura de um loop while  
(fonte: <https://www.geeksforgeeks.org/cpp-while-loop/>)

# While

```
int i = 6;

// verificar condição
while (i > 0) {

    // instrução para repetir
    cout << "Hey listen!" << endl;

    // atualizar condição
    i--;
}
```

Demonstração da estrutura e funcionamento de um loop *while*, o programa executará o bloco de código no *while* 6 vezes

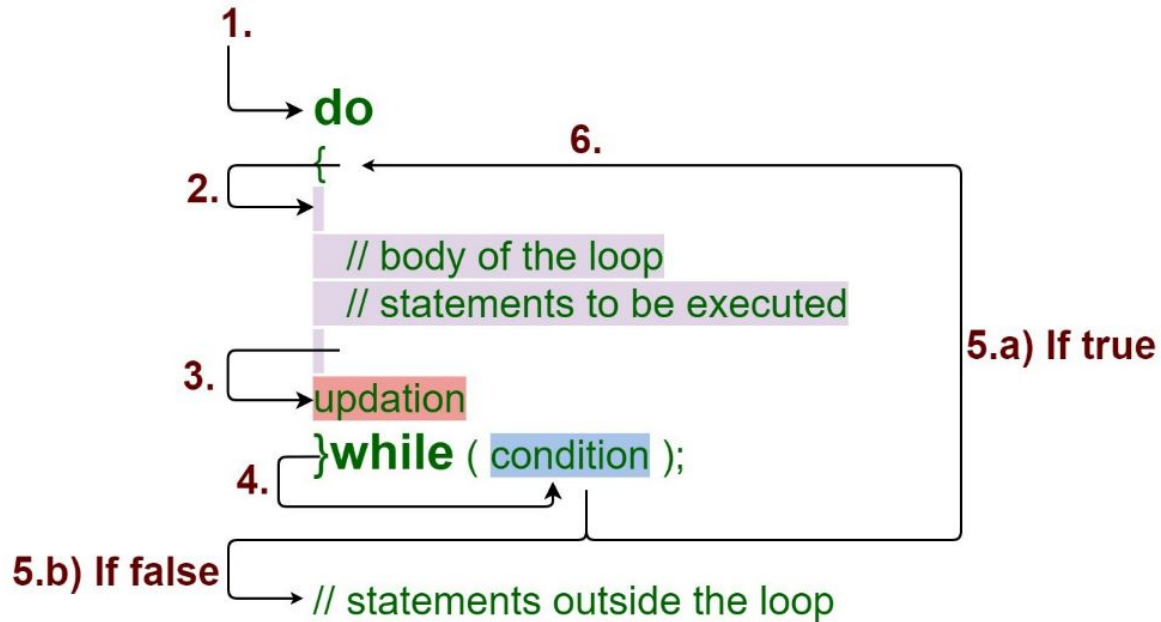
# Do-while

- O loop *do-while* é uma variante do *while*. Ele executa o bloco de código uma vez antes de verificar se a condição é verdadeira, então repete a execução do bloco enquanto a condição for verdadeira.
- Mesmo que a condição seja falsa, o *do-while* sempre a executará pelo menos uma vez.



# Do-while

## Do - While Loop



Sintaxe da estrutura de um loop do-while

(fonte: <https://www.geeksforgeeks.org/cpp-do-while-loop/?ref=lbp>)

# Do-while

```
int i = 6;

do {
    // instrução para repetir
    cout << "Hey listen!" << endl;

    // atualizar condição
    i--;
}
// verificar condição
while (i > 0);
```

Demonstração da estrutura e funcionamento de um loop *do-while*, o programa executará o bloco de código no *do* 6 vezes

# For

- O loop *for* deve ser usado em vez do *while* quando se sabe quantas vezes exatamente o bloco de código deve ser executado.
- A estrutura do loop *for* é definida por três campos de instruções:
  - ◆ A primeira instrução é executada uma vez antes da execução do bloco.
  - ◆ A segunda instrução define a condição de execução do loop.
  - ◆ A terceira instrução é executada toda vez que o bloco finaliza a execução.

# For

```
int i = 0;

// O loop inicia atribuindo valor 1 à variável i
// "i <= 6" é a condição de testagem do loop
// O loop executará se, e somente se, i for menor
// que 6
// "i++" incrementará o valor de i em 1 a cada
// iteração do loop

for (i = 1; i <= 6; i++) {
    // instrução para repetir
    cout << "Hey listen!" << endl;
}
```

Código explicando a sintaxe e funcionamento de um loop *for*

# “For-each”

- Há mais um tipo de loop em C++, o loop “*for-each*”, que foi projetado exclusivamente para rodar pelos elementos de um array (ou outros tipos de data sets).

```
int array[5] = {5, 4, 3, 2, 1};
string msg = "BANG";

// o loop percorrerá todos os elementos do
// array e encerrará uma vez que atinja o
// final dele
for (int i : array) {
    cout << i << "..." << endl;
}
for (char var : msg) {
    cout << var << " ";
}
cout << endl;
```

Exemplo de código usando um loop “for-each”

# Estruturas

# Estruturas

- Estruturas (ou *structs*) são uma forma de agrupar variáveis relacionadas em um único lugar. Cada variável em uma estrutura é chamada membro da estrutura.
- Para criar uma estrutura, usamos a palavra-chave *struct* e declaramos cada membro dentro de chaves, que definem o escopo da estrutura. Após a declaração, especificamos o nome da variável estrutura.
- O acesso aos membros da estrutura é feito utilizando ponto (.) como operador para referenciá-los.

# Estruturas

```
int main() {  
  
    // declarando uma variável estrutura chamada "carro"  
    struct {  
        string marca;  
        string modelo;  
        int ano;  
    } carro;  
  
    // atribuindo valores aos membros da estrutura  
    carro.marca = "Ford";  
    carro.modelo = "Mustang";  
    carro.ano = 1969;  
  
    // printando membros de "carro"  
    cout << "Marca: " << carro.marca << endl  
        << "Modelo: " << carro.modelo << endl  
        << "Ano: " << carro.ano << endl;  
  
    return 0;  
}
```

Exemplo de declaração de uma estrutura e atribuição de valores e consulta dos membros da estrutura



# Estruturas

- Nomeando uma estrutura, é possível utilizá-la como tipo de dados, o que permite criar variáveis com essa estrutura em qualquer lugar no programa.
- Para nomear uma estrutura, basta colocar seu nome logo depois da palavra-chave *struct*.

# Estruturas

```
// definindo uma estrutura chamada "carro"
struct carro {
    string marca;
    string modelo;
    int ano;
};

int main() {
    // declarando uma variável "carro1" do tipo struct "carro"
    carro carro1;
    carro1.marca = "Ford";
    carro1.modelo = "Mustang";
    carro1.ano = 1969;

    // declarando outra variável "carro2" do tipo struct "carro"
    carro carro2;
    carro2.marca = "BMW";
    carro2.modelo = "X5";
    carro2.ano = 1999;

    cout << carro1.marca + " " + carro1.modelo + " " << carro1.ano << endl
         << carro2.marca + " " + carro2.modelo + " " << carro2.ano << endl;

    return 0;
}
```

Exemplo demonstrando a utilização de uma estrutura como tipo de dados

# Funções

# Funções

- Uma função é um bloco de código que executa apenas quando é chamada.
- É possível passar dados, chamados parâmetros, para uma função.
- Toda função em C++ deve ter um tipo de dado atribuído a ela, é esse tipo que definirá o retorno da função, como veremos mais à frente.

# Funções

- Para criar uma função, defina o tipo atribuído a ela e especifique o nome da função, sucedendo-o com um par de parênteses e um par de chaves (os parâmetros de uma função são definidos dentro dos parênteses e seu bloco de execução é definido dentro das chaves).
- Uma função em C++ consiste de duas partes:
  - ◆ **Declaração:** o tipo de retorno, o nome da função e os parâmetros.
  - ◆ **Definição:** o corpo de execução da função (bloco de código).

# Funções

```
// a função abaixo recebe dois valores inteiros
// "val1" e "val2" como parâmetros e retorna
// o maior dentre eles
int max(int val1, int val2) {
    return (val1 > val2)? val1 : val2;
}

int main() {
    int num1 = 14;
    int num2 = 17;

    // chamando a função "max" para as variáveis
    // "num1" e "num2"
    int m = max(num1, num2);
    cout << "Máximo: " << m << endl;

    return 0;
}
```

A função “max” retorna o maior entre dois números enviados para a função como argumentos por meio dos parâmetros “val1” e “val2”

# Funções

- Uma função deve ser declarada no programa antes de ser chamada, mas não precisa necessariamente ser definida.
- É uma prática comum em C++ separar a declaração e a definição de uma função, declarando-a antes da *main()* e definindo-a após a *main()*.

# Funções

```
int max(int val1, int val2); // declaração

int main() {
    int num1 = 14;
    int num2 = 17;

    // chamando a função "max" para as variáveis
    // "num1" e "num2"
    int m = max(num1, num2);
    cout << "Máximo: " << m << endl;

    return 0;
}

int max(int val1, int val2) {
    return (val1 > val2)? val1 : val2; // definição
}
```

No código, a função “max” é definida após a *main*, mas é declarada antes, sem essa declaração prévia da função, sua chamada na main ocasionaria um erro no programa



# Parâmetros e argumentos

- Dados podem ser passados para funções por meio de parâmetros, que funcionam como variáveis dentro da função.
- No momento de definir os parâmetros de uma função, deve-se sempre especificar o tipo de dado correspondente a cada parâmetro.
- Quando um parâmetro é passado para uma função, ele é chamado de argumento.

# Parâmetros e argumentos

- É possível atribuir um valor padrão para o parâmetro de uma função, usando o sinal de igual (=). Assim, quando a função for chamada sem um parâmetro, ela usará o valor padrão atribuído.

# Parâmetros e argumentos

```
//a função abaixo recebe dois valores inteiros  
// "val1" e "val2" como parâmetros e retorna  
// o maior dentre eles  
int max(int val1, int val2 = 10) {  
    return (val1 > val2)? val1 : val2;  
}  
  
int main() {  
    int num1 = 14;  
    int num2 = 17;  
  
    // chamando a função "max" para as variáveis  
    // "num1" e "num2"  
    int m = max(num1, num2);  
    cout << "Máximo: " << m << endl;  
  
    // a ausência do segundo parâmetro induzirá  
    // a função a utilizar o argumento padrão  
    // definido em sua declaração  
    m = max(num1);  
    cout << "Máximo: " << m << endl;  
  
    return 0;  
}
```

Na primeira chamada da função, “val1” recebe o valor de “num1” e “val2” recebe o valor de “num2” na passagem de parâmetros. Dessa forma, “m” recebe o valor 17 após execução da função.

Na segunda chamada, apenas o valor de “val1” é passado nos parâmetros, então o programa utiliza o valor padrão definido para “val2” para executar a função adequadamente, o que resulta em “m” recebendo o valor 14.

## Argumentos `argc` e `argv`

- A função `main()` pode conter parâmetros formais, mas esses não podem ser escolhidos pelo programador. A declaração mais completa da função `main()` é: `main(int argc, char* argv[])`.
- Os parâmetros `argc` e `argv` dão acesso à linha de comando com a qual o programa foi chamado.
- O `argc` (argument count) é um inteiro e possui o número de argumentos com os quais a função `main()` foi chamada.

## Argumentos `argc` e `argv`

- O *argv* (argument values) é um vetor de strings, em que cada string corresponde a um dos parâmetros da linha de comando. O *argc* nos informa quantos elementos temos em *argv*.

# Argumentos argc e argv

```
// Linha de comando: ./test [dia] [mês] [ano]

// argc = 4
// argv = {"./test", "[dia]", "[mês]", "[ano]"}

// O código recebe uma data na linha de comando
// no formato acima e retorna uma string no
// formato "[dia] de [nomemes] de 20[ano]"
int main(int argc, char* argv[]) {
    int mes;
    string nomemes[] = {"Janeiro", "Fevereiro", "Março",
                        "Abril", "Maio", "Junho", "Julho", "Agosto", "Setembro",
                        "Outubro", "Novembro", "Dezembro"};

    if(argc == 4) {
        mes = atoi(argv[2]);
        if (mes<1 || mes>12)
            cout << "Erro!\nMes invalido!" << endl;
        else
            cout << argv[1] << " de " << nomemes[mes-1] << " de 20" << argv[3] << endl;
    }
    else
        cout << "Erro!\nUso: data dia mes ano, todos inteiros\n";
}
```

Exemplo de código demonstrando a lógica de funcionamento dos argumentos de linha de comando argc e argv

# Valores de retorno

- Quase toda função em C++ possui retorno, isto é, produz um resultado. O valor de retorno de uma função é definido pelo tipo atribuído à sua declaração. Por exemplo, uma função do tipo *int* só pode retornar valores numéricos inteiros e uma função do tipo *string* só pode retornar strings.
- Em C++, o tipo *void* indica que uma função não possui valor de retorno.

# Valores de retorno

```
// função com tipo de retorno string
string hello() {
    return "Hello wonderful world!\n";
}

// função void (sem retorno)
void bye() {
    cout << "Goodbye cruel world!" << endl;
}

int main() {

    // printando a mensagem retornada por "hello"
    cout << hello();

    // não precisamos chamar a instrução de print
    // para a função "bye" pois ela já está
    // incorporada dentro da função
    bye();

    return 0;
}
```

No exemplo, vemos três funções com tipos de retorno distintos: *hello* com tipo string, *bye* com tipo void (indicando que não possui retorno) e a *main* com tipo int



# Referenciando parâmetros

- Como já falado anteriormente, referenciar variáveis é um ponto principal quando trabalhamos com C++, sobretudo no âmbito das funções.
- Quando um parâmetro é passado para uma função, na verdade, a função “clona” a variável. Assim, se o argumento for alterado dentro da função, a alteração não será mantida fora do seu escopo.

# Referenciando parâmetros

- No entanto, quando passamos um parâmetro referenciado, isso permite que qualquer alteração feita dentro da função seja mantida fora do seu escopo.

# Referenciando parâmetros

```
// as duas funções alteram o valor de "var",  
// mas uma recebe o parâmetro referenciado  
// enquanto a outra não  
void nref(int var) {  
    var = 17;  
    cout << "var = " << var << endl;  
}  
void ref(int* var) {  
    *var = 13;  
    cout << "var = " << *var << endl;  
}  
  
int main() {  
    int var = 10;  
    cout << "var = " << var << endl;  
  
    // chamamos as funções "nref" e "ref" que  
    // alterarão o valor da variável e verificamos  
    // seu estado depois de cada chamada  
  
    nref(var);  
    cout << "var = " << var << endl;  
    ref(&var);  
    cout << "var = " << var << endl;  
  
    return 0;  
}
```

Quando o valor do argumento é alterado dentro da função sem que o parâmetro seja passado como referência, essa alteração não é mantida fora do escopo da função. Dessa forma, quando printamos o valor de "var" após a chamada de "nref", podemos constatar que a variável ainda tem valor 10, mesmo após ter sido modificada dentro da função.

# Sobrecarga de Funções

# Sobrecarga de Funções

- Em C++, é possível declarar inúmeras funções com parâmetros distintos que compartilhem o mesmo nome. Isso é chamado de sobrecarga de funções e possibilita a criação de múltiplas instâncias para uma mesma função, com cenários e parâmetros distintos.

# Sobrecarga de Funções

```
// a função recebe dois valores inteiros e
// retorna o maior entre eles
int max2int(int val1, int val2) {
    return (val1 > val2)? val1 : val2;
}

// a função recebe três valores inteiros e
// retorna o maior entre eles
int max3int(int val1, int val2, int val3) {
    return (val1 > val2)?
        ((val1 > val3)? val1 : val3) :
        ((val2 > val3)? val2 : val3);
}

// a função recebe dois valores racionais e
// retorna o maior entre eles
double max2double(double val1, double val2) {
    return (val1 > val2)? val1 : val2;
}

int main() {
    int int1 = 13, int2 = 15, int3 = 21;
    double doub1 = 3.45, doub2 = 3.4;

    cout << "Maximo: " << max2int(int1, int2) << endl;
    cout << "Maximo: " << max3int(int1, int2, int3) << endl;
    cout << "Maximo: " << max2double(doub1, doub2) << endl;

    return 0;
}
```

O exemplo apresenta três funções que adotam parâmetros distintos, mas que seguem o mesmo processo e têm a mesma funcionalidade, o que pode ocasionar um certo sucateamento do código conforme mais variações das funções são criadas

# Sobrecarga de Funções

```
// a função recebe dois valores inteiros e
// retorna o maior entre eles
int max(int val1, int val2) {
    return (val1 > val2)? val1 : val2;
}

// a função recebe três valores inteiros e
// retorna o maior entre eles
int max(int val1, int val2, int val3) {
    return (val1 > val2)?
        ((val1 > val3)? val1 : val3) :
        ((val2 > val3)? val2 : val3);
}

// a função recebe dois valores racionais e
// retorna o maior entre eles
double max(double val1, double val2) {
    return (val1 > val2)? val1 : val2;
}

int main() {
    int int1 = 13, int2 = 15, int3 = 21;
    double doub1 = 3.45, doub2 = 3.4;

    cout << "Maximo: " << max(int1, int2) << endl;
    cout << "Maximo: " << max(int1, int2, int3) << endl;
    cout << "Maximo: " << max(doub1, doub2) << endl;

    return 0;
}
```

Utilizando a sobrecarga de funções, podemos definir uma função *max* “única”, mas com cenários e aplicações variados, comportando quantidades e tipos de parâmetros diferentes

# Recursão



# Recursão

- Recursão é a técnica de fazer uma função chamar a si própria. Essa técnica permite quebrar problemas complexos em problemas mais simples de resolver.
- É muito importante estabelecer uma condição de parada clara para que a função não fique presa em uma recursão infinita.

# Recursão

```
// a função calcula o fatorial do número inteiro  
// passado como parâmetro  
int fact(int var) {  
    return (var == 0)? 1 : var * fact(var - 1);  
}  
  
int main() {  
    int num = 10;  
    cout << "Fatorial = " << fact(num) << endl;  
  
    return 0;  
}
```

A função *fact* chama a si mesma recursivamente, parando apenas quando o valor de “var” chega a zero, quebrando uma operação de cálculo fatorial em uma série de multiplicações recursivas

# **Programação Orientada a Objetos**

# Programação Orientada a Objetos

- C++ é uma linguagem de programação orientada a objetos, ou seja, é uma linguagem de programação que se baseia no conceito de objetos. Enquanto a programação procedural é voltada a escrever procedimentos e funções que executam operações nos dados, a programação orientada a objetos trabalha com a criação de objetos que contêm dados e funções.

# Programação Orientada a Objetos

- A programação orientada a objetos possui algumas vantagens sobre a programação procedural:
- ◆ É mais rápida e fácil de executar;
  - ◆ Oferece uma estrutura clara para o programa;
  - ◆ Ajuda a manter os códigos sem redundância e os tornam mais fáceis de manter, modificar e debugar.
  - ◆ Torna possível criar aplicações reusáveis com menos código e um menor tempo de desenvolvimento.

# Classes e Objetos

# Classes e Objetos

- Classes e objetos são os dois aspectos principais da programação orientada a objetos.
- Uma classe é um modelo de objeto e um objeto é uma instância de uma classe.
- Quando os objetos são criados, eles herdam todas as variáveis e funções da classe.

# Classes e Objetos

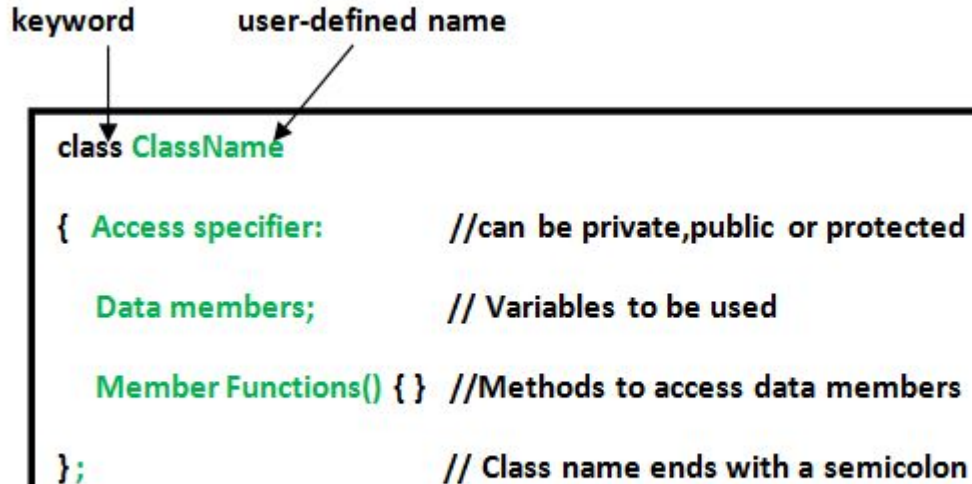
- C++ é uma linguagem orientada a objetos, portanto tudo nela está associado a classes e objetos, assim como seus atributos e métodos.
- Atributos e métodos são basicamente variáveis e funções que pertencem à classe e são geralmente referenciados como “membros da classe”.



# Classes

- Classes são tipos definidos pelo usuário que podem ser usados em programas e servem como um objeto construtor ou uma “receita” para criar objetos.
- Para criar uma classe, usa-se a palavra-chave *class*, como mostrado a seguir.

# Classes



The diagram illustrates the syntax for declaring a class. It features a code block with several lines of code. Above the code block, two labels with arrows point to specific parts of the code: 'keyword' points to the word 'class', and 'user-defined name' points to 'ClassName'. The code itself is as follows:

```
class ClassName  
  
{ Access specifier:           //can be private,public or protected  
  
    Data members;           // Variables to be used  
  
    Member Functions() {} //Methods to access data members  
  
};                           // Class name ends with a semicolon
```

Modelo de sintaxe para declaração de uma classe  
(fonte: <https://www.geeksforgeeks.org/c-classes-and-objects/?ref=lbp>)

# Objetos

- Objetos são criados a partir de classes. Para criar um objeto, especificamos o nome da classe à qual ele pertence antes do nome do objeto.
- Para acessar os atributos e métodos de um objeto, usamos um ponto final (.), de forma semelhante a estruturas.

# Objetos

```
class Aluno {  
    // especificador de acesso  
    public:  
        // membros da classe  
        string nome;  
        int idade;  
  
        // metodos da classe  
        void printDados() {  
            cout << "Nome: " << nome << endl  
                << "Idade: " << idade << endl;  
        }  
};  
  
int main() {  
    // declarando um objeto da classe Aluno  
    Aluno a1;  
  
    // acessando e atribuindo valores aos  
    // membros da classe  
    a1.nome = "João Augusto";  
    a1.idade = 22;  
  
    // acessando os metodos da classe  
    a1.printDados();  
  
    return 0;  
}
```

Cada objeto definido de uma classe “copia” a sua estrutura, criando instâncias individuais de cada atributo

# Métodos de classe

- Métodos são funções que pertencem a uma classe. Há duas formas de definir métodos: dentro e fora da classe.
- A definição interna à classe funciona de forma semelhante à definição de uma função comum, apenas dentro do escopo da classe.
- A definição externa à classe demanda que a função seja declarada dentro do escopo da classe, mas sua definição ocorre de maneira externa. A definição externa necessita que a classe à qual a função pertence seja especificada com o uso do operador de escopo (::).

# Métodos de classe

```
class Aluno {  
    // especificador de acesso  
    public:  
        // membros da classe  
        string nome;  
        int idade;  
  
        // métodos da classe  
        // printNome é definido fora da classe  
        void printNome();  
        // printIdade é definido dentro da classe  
        void printIdade() {  
            cout << "Idade: " << idade << endl;  
        }  
};  
  
// a definição é feita usando o operador de escopo  
// para referenciar a classe à qual o método pertence  
void Aluno::printNome() {  
    cout << "Nome: " << nome << endl;  
}
```

No exemplo, temos dois métodos na classe “Aluno”: *printIdade*, definido dentro da classe, e *printNome*, definido fora dela

# Construtores

- Um construtor é um método especial que é chamado automaticamente quando um objeto é criado.
- Para declarar um construtor, basta declarar um método sem tipo (portanto, sem retorno) e com o mesmo nome da classe a que ele pertence.
- Como qualquer função, construtores podem ser declarados com ou sem parâmetros e, como qualquer método, podem ser definidos dentro ou fora da classe.

# Construtores

```
class Aluno {  
    // especificador de acesso  
    public:  
        // membros da classe  
        string nome;  
        int idade;  
  
        // construtor da classe  
        Aluno() {  
            cout << "Novo registro de aluno!" << endl;  
        }  
};  
  
int main() {  
    // declarando um objeto da classe Aluno,  
    // isso chamará o construtor da classe  
    Aluno a1;  
  
    return 0;  
}
```

O construtor de uma classe é chamado automaticamente quando um objeto dessa classe é declarado



# Especificadores de Acesso

- Especificadores de acesso definem como os membros (métodos ou atributos) de uma classe podem ser acessados.
- Em C++, nós temos três especificadores de acesso, cada um representado por uma palavra-chave:
  - ◆ ***public:*** membros são acessíveis fora da classe.
  - ◆ ***private:*** membros não podem ser acessados ou visualizados fora da classe.
  - ◆ ***protected:*** membros não podem ser acessados fora da classe, mas podem ser acessados por classes herdeiras (falaremos mais sobre **Herança** depois).

# Especificadores de Acesso

- Por padrão, todos os membros de uma função são privados, a menos que seja especificado o contrário.

```
class Aluno {  
    // membros privados  
    private:  
        string nome;  
        int idade;  
  
    // membros públicos  
    public:  
        void setNome() {  
            getline(cin, nome);  
        }  
        void setIdade() {  
            cin >> idade;  
        }  
};
```

Os membros privados da classe só podem ser acessados por outros membros da mesma classe, a tentativa de acesso de um membro privado em outra região do código resultará em um erro.

Os membros públicos da classe são acessíveis em qualquer área do código.

# Encapsulamento

# Encapsulamento

- Encapsulamento é uma técnica usada para garantir que dados “sensíveis” fiquem ocultos ao usuário.
- Para isso, os atributos da classe são declarados como privados, impedindo que eles sejam acessados fora da classe.
- Se for necessário ler ou modificar o valor de um membro privado, são criados métodos públicos para obter (get) e atribuir (set) os valores.

# Encapsulamento

- Declarar os atributos de uma classe como privados é considerado uma boa prática.
- Encapsulamento assegura um melhor controle dos dados, pois permite alterar uma parte do código sem impactar outras.
- O encapsulamento de atributos também concede uma maior segurança de dados.

# Herança

# Herança

- Em C++, uma classe pode ser capaz de herdar atributos e métodos de outra classe.
- A herança é útil pois permite a reutilização de atributos e métodos de uma classe já existente na criação de outra classe.
- Para definir uma herança, usa-se o símbolo “:”.

# Herança

- A herança é separada em duas categorias:
  - ◆ **classe derivada (filha):** classe que herda de outra classe.
  - ◆ **classe base (mãe):** classe da qual é herdado.
- Uma classe derivada também pode ser base para outras classes.
- Uma classe pode ser derivada de mais de uma classe, desde que todas as classes base estejam listadas em sua declaração.



# Herança

- Uma classe derivada tem acesso livre a quaisquer métodos e atributos públicos (*public*) e protegidos (*protected*) de sua classe base, mas não pode acessar aqueles privados (*private*).

# Herança

```
class Pessoa {  
    string nome;  
    int idade;  
  
    public:  
        void setNome();  
        void setIdade();  
        void printNome();  
        void printIdade();  
};
```

```
class Aluno : public Pessoa {  
    string curso;  
    double nota;  
  
    public:  
        void setCurso();  
        void setNota();  
        void printCurso();  
        void printNota();  
};
```

A classe *Aluno* é uma classe derivada de *Pessoa*,  
dessa forma, qualquer objeto pertencente à  
classe *Aluno* possuirá também os atributos e  
métodos da classe *Pessoa*

# Herança

```
class Pai {  
    public:  
        string sobrenome_p;  
};  
  
class Mae {  
    public:  
        string sobrenome_m;  
};  
  
class Filho : public Pai, public Mae {};
```

A classe *Filho* é derivada das classes *Pai* e *Mae* e, portanto, herda os atributos e métodos de ambas

# Fim

INSTITUIÇÃO EXECUTORA



COORDENADORA



APOIO

