

# Apostila



# Progressivo

[www.cmmprogressivo.net](http://www.cmmprogressivo.net)

# Sobre o e-book C++ Progressivo

Antes de mais nada, duas palavrinhas: parabéns e obrigado.

Primeiro, parabéns por querer estudar, aprender, por ir atrás de informação. Isso é cada vez mais raro hoje em dia.

Segundo, obrigado por ter adquirido esse material em [www.cmmprogressivo.net](http://www.cmmprogressivo.net). Mantemos neste endereço um *website* totalmente voltado para o ensino da linguagem de programação C++.

O objetivo dele é ser bem completo, ensinando praticamente tudo sobre C++, desde o básico, supondo o leitor um total leigo em que se refere a absolutamente tudo sobre computação e programação.

Ele é gratuito, não precisa pagar absolutamente nada. Aliás, precisa nem se cadastrar, muito menos deixar e-mail ou nada disso. É simplesmente acessar e estudar.

E você, ao adquirir esse e-book, está incentivando que continuemos esse trabalho.

Confiram nossos outros trabalhos:

[www.programacaoprogessiva.net](http://www.programacaoprogessiva.net)

[www.pythonprogressivo.net](http://www.pythonprogressivo.net)

[www.javaprogressivo.net](http://www.javaprogressivo.net)

[www.cprogressivo.net](http://www.cprogressivo.net)

[www.htmlprogressivo.net](http://www.htmlprogressivo.net)

[www.javascriptprogressivo.net](http://www.javascriptprogressivo.net)

[www.phpprogressivo.net](http://www.phpprogressivo.net)

Certamente, seu incentivo \$\$ vai nos motivar a fazer cada vez mais artigos, tutoriais e novos sites!

Este e-book é composto por todo o material do site. Assim, você pode ler no computador, no tablet, no celular, em casa, no trabalho, com ou sem internet, se tornando algo bem mais cômodo.

Além disso, este e-book contém mais coisas, mais textos e principalmente mais exercícios resolvidos, de modo a te oferecer algo mais, de qualidade, por ter pago pelo material.

Aliás, isso não é pagamento, é investimento. Tenho certeza que, no futuro, você vai ganhar 10x mais, por hora trabalhada, graças ao conhecimento que adquiriu aqui.

# Proprietário da Apostila

Esse e-book pertence a:

Nome: Joao Pedro Sales Deus

Código: 1DFC9E0DC297469199B11099B42C3902

Email do comprador: jdedeus@sga.pucminas.br

Banco Pagador: MERCADO PAGO IP LTDA.

Pedimos, encarecidamente, que não distribua ou comercialize seu material. Além de conter suas informações, prejudica muito nosso projeto.

Se desejar indicar o C++ Progressivo para um amigo, nosso site possui todo o material, de forma gratuita, sem precisar de cadastro e o acesso dessas pessoas também ajuda a mantermos o site no ar e criamos cada vez mais projetos:

[www.cmmprogressivo.net](http://www.cmmprogressivo.net)

# Sumário

## Índice

|   |    |
|---|----|
| Sobre o e-book C++ Progressivo.....   | 2  |
| Proprietário da Apostila.....   | 3  |
| Sumário.....  | 4  |
| Básico da linguagem C++.....  | 11 |
| Computação e Programação de Computadores.....                                       | 12 |
| Computação e Computadores.....  | 12 |
| Programação de computadores.....  | 12 |
| Hardware e Software.....  | 13 |
| Linguagem de Programação.....   | 14 |
| Programa de Computador.....   | 15 |
| Como ser um bom programador C++.....  | 16 |
| Fontes de estudo.....   | 17 |
| Linguagem C++: O que é? Para que serve? Como funciona? Onde é usada ?.....          | 18 |
| O que é C++ ?.....  | 18 |
| C e C++.....  | 19 |
| Programas desenvolvidos em C++.....   | 20 |
| Onde C++ não é (tão) usado.....   | 21 |
| Referências e fontes de estudo.....   | 22 |
| C ou C++? Qual a diferença? Qual é melhor? Precisa saber C para aprender C++?.....  | 23 |
| Para aprender C++ precisa saber C?.....   | 23 |
| É necessário saber C para aprender C++ ?.....                                       | 23 |
| Qual a diferença entre C e C++ ?.....   | 23 |
| C++ é melhor que a linguagem C ?.....   | 24 |
| Onde se usa C e onde se usa C++ ?.....  | 24 |
| Qual o melhor, Java ou C++?.....  | 25 |
| Os comandos de C rodam em C++?.....   | 26 |
| Como está o mercado de C++ no Brasil ?.....   | 26 |
| Como começar a programar em C++.....  | 28 |
| O que precisa pra programar em C++ ?.....   | 28 |
| IDE: CodeBlocks.....  | 29 |
| Primeiro Programa: "Olá, mundo" ("Hello, world!").....                              | 31 |
| Criando um Projeto no Code::Blocks.....   | 31 |
| Programando o primeiro programa em C++.....   | 34 |
| Compilando e Executando um código C++.....  | 35 |
| Entendendo o código C++.....  | 36 |
| Pré-processar, Compilar, Linkar, Carregar e Rodar um programa C++.....              | 37 |
| Saída simples em C++: cout, <<, endl, \n e Outros caracteres especiais.....         | 38 |
| Saída simples (imprimindo coisas na tela): cout e <<.....                           | 38 |
| Quebra de linha: endl e \n.....   | 40 |
| Caracteres especiais em C++.....  | 41 |
| Fontes de estudo e consulta.....  | 41 |
| Exercícios de saída simples em C++: cout, <<, endl e \n.....                        | 42 |
| Exercícios de C++.....  | 42 |
| Tipos de Dados, Variáveis e Atribuição em C++: int, char, float, double e bool..... | 44 |
| Armazenando Informações.....  | 44 |

|  |    |
|--|----|
| O tipo de dado inteiro: int.....   | 44 |
| Tipo de dado caractere: char.....  | 46 |
| Tipo de dado flutuante: float e double.....  | 47 |
| O tipo de dado Booleano: bool.....   | 48 |
| Exercícios:.....   | 50 |
| Nomes de variáveis.....  | 50 |
| Palavras-chave reservadas.....   | 51 |
| Resposta do exercício.....   | 51 |
| A Função sizeof() em C++ e outros tipos de dados (short, long e unsigned).....                                       | 52 |
| Tamanho de dados: a função sizeof().....   | 52 |
| Tipo de dado inteiro: short, long e unsigned.....  | 53 |
| Precisão de float e double.....  | 55 |
| Referências.....   | 55 |
| Matemática em C++: Operadores de soma (+), subtração (-), multiplicação (*), divisão (/) e resto da divisão (%)..... | 56 |
| Soma em C++: operador +.....   | 56 |
| Subtração em C++: operador -.....  | 57 |
| Multiplicação em C++: operador *.....  | 58 |
| Divisão em C++: operador /.....  | 59 |
| Resto da divisão em C++: operador %.....   | 61 |
| Exercícios de Matemática em C++.....   | 61 |
| Fontes de estudo.....  | 62 |
| Precedência de Operadores e Agrupamento de Expressões com Parêntesis.....  | 63 |
| Ordem dos operadores matemáticos.....  | 63 |
| Precedência de Operadores Matemáticos no C++.....  | 64 |
| Parêntesis () - Organização e Precedência.....   | 65 |
| Exercícios de C++.....   | 66 |
| Respostas:.....  | 66 |
| Referências de estudo.....   | 67 |
| Recebendo Dados do Teclado - O Objeto cin do C++.....  | 68 |
| Trocando Informações.....  | 68 |
| Como Receber Informações do Usuário: cin >>.....   | 68 |
| Recebendo Números do Usuário com cin >>.....   | 69 |
| Importante: vírgula é ponto.....   | 71 |
| Recebendo textos (strings) do usuário.....   | 72 |
| Recebendo dados Booleanos em C++.....  | 73 |
| Exercícios usando cin>>.....   | 73 |
| Resposta dos exercícios.....   | 74 |
| Referência de estudo.....  | 76 |
| Como comentar códigos em C++: // e /* */.....  | 77 |
| Comentários: O que são? Para que servem? Por que usar?.....  | 77 |
| Como comentar uma linha em C++: //.....  | 78 |
| Como comentar várias linhas: /* */.....  | 79 |
| Exercícios Básicos de C++.....   | 81 |
| Questões de C++.....   | 81 |
| Exercícios de Porcentagem em C++.....  | 81 |
| Questões de Média em C++.....  | 83 |
| Soluções.....  | 83 |
| Média Aritmética Simples em C++.....   | 90 |
| Exercícios de Média aritmética Simples.....  | 90 |

|  |     |
|--|-----|
| Média Ponderada em C++.....  | 92  |
| Exercícios de Média em C++.....  | 93  |
| Calculadora Simples em C++: Como Programar.....                              | 94  |
| Como fazer uma calculadora em C++.....                                       | 94  |
| Teste Condicionai em C++: IF e ELSE.....                                     | 96  |
| Operadores Relacionais (de comparação) em C++: > , < , >= , <=, == e !=..... | 97  |
| Como comparar coisas em C++.....   | 97  |
| Operador de igualdade em C++: ==.....  | 98  |
| Operador de diferença em C++: !=.....  | 99  |
| Operador de maior em C++: >.....   | 99  |
| Operador de maior igual em C++: >=.....                                      | 100 |
| Operador Menor e Menor igual em C++: < e <=.....                             | 100 |
| Cuidado com comparações.....   | 100 |
| Exercício de C++.....  | 101 |
| O teste condicional IF.....  | 102 |
| Tomando decisões em C++.....   | 102 |
| O comando IF no C++.....   | 102 |
| Como usar o IF em C++.....   | 103 |
| Exemplo de uso do IF.....  | 104 |
| Exemplo de uso do IF.....  | 104 |
| Usando o IF em C++.....  | 105 |
| Exercício.....   | 106 |
| A instrução IF e ELSE.....   | 107 |
| A Instrução IF e ELSE do C++.....  | 107 |
| Exemplo de IF e ELSE.....  | 108 |
| Como usar IF e ELSE.....   | 109 |
| Usando IF e ELSE em C++.....   | 109 |
| Par ou Ímpar: Como descobrir (e outros múltiplos).....                       | 111 |
| Par ou Ímpar em C++.....   | 111 |
| Múltiplo de 3.....   | 112 |
| Números múltiplos.....   | 113 |
| Operador Condicional Ternário ?:.....  | 114 |
| Operador Condicional ?:.....   | 114 |
| Como usar o Operador Ternário ?:.....  | 114 |
| Operador Ternário ?: como utilizar em C++.....                               | 115 |
| Exemplo de uso de ?:.....  | 115 |
| Usando o operador ternário ?: em C++.....                                    | 116 |
| IF e ELSE aninhados.....   | 117 |
| IF e ELSE dentro de IF e ELSE.....   | 117 |
| Exemplo de IF e ELSE aninhados.....  | 118 |
| IF e ELSE aninhados em C++.....  | 119 |
| O comando IF/ELSE IF.....  | 121 |
| Operadores Lógicos em C++: AND (&&), OR (  ) e NOT (!).....                  | 124 |
| Operador Lógico AND: &&.....   | 124 |
| Operador Lógico OR:   .....  | 125 |
| Operador Lógico NOT: !.....  | 127 |
| Exercício de Operadores Lógicos.....   | 128 |
| Ano bissexto em C++.....   | 129 |
| Anos bissextos.....  | 129 |
| Ano bissexto em C++.....   | 130 |

|   |     |
|---|-----|
| Algoritmo do ano bissexto em C++.....                     | 132 |
| O comando SWITCH em C++.....                              | 133 |
| Tomando caminhos diferentes - Ramificações.....           | 133 |
| Comandos SWITCH e CASE em C++.....                        | 133 |
| Criando um menu com o comando SWITCH.....                 | 134 |
| O comando BREAK no Switch.....                            | 136 |
| Exemplo de uso do SWITCH CASE.....                        | 137 |
| Cases acumulados.....                                     | 139 |
| Exercício de SWITCH CASE massa.....                       | 141 |
| Exercícios de Testes Condicionais em C++.....             | 142 |
| Problemas de testes condicionais em C++.....              | 142 |
| Soluções dos exercícios de Testes condicionais.....       | 144 |
| Como comparar dois números em C++.....                    | 144 |
| Como trocar o valor de dois números em C++.....           | 145 |
| 3 números em ordem crescente.....                         | 147 |
| 3 números em ordem decrescente.....                       | 148 |
| Quantos dias tem no mês.....                              | 149 |
| Tipos de triângulo em C++.....                            | 152 |
| Equilátero, Isósceles ou Escaleno ?.....                  | 154 |
| Teste hacker C++.....                                     | 155 |
| Fórmulas de Bháskara em C++.....                          | 155 |
| Desafio de C++.....                                       | 157 |
| Laços e Loopings.....                                     | 159 |
| Operadores de Atribuição, de incremento e decremento..... | 160 |
| Operadores de Atribuição.....                             | 160 |
| Operadores de Incremento e Decremento.....                | 161 |
| O laço WHILE em C++.....                                  | 164 |
| Laço WHILE: Estrutura de Repetição.....                   | 164 |
| Exemplo de uso de WHILE - Estrutura de Repetição.....     | 165 |
| Como usar laço WHILE em C++.....                          | 166 |
| WHILE em C++: Validando entradas.....                     | 167 |
| Exemplo de uso de WHILE em C++.....                       | 168 |
| DO ... WHILE looping em C++.....                          | 169 |
| O Looping DO ... WHILE.....                               | 169 |
| Como usar DO WHILE em C++.....                            | 170 |
| Exemplo de uso de DO WHILE.....                           | 171 |
| Estrutura de Repetição FOR - Laço controlado.....         | 173 |
| Estrutura de repetição FOR em C++.....                    | 173 |
| Exemplo de uso do laço FOR.....                           | 174 |
| Como usar a estrutura de repetição FOR.....               | 174 |
| Quando usar o laço FOR.....                               | 176 |
| Estrutura de Repetição FOR.....                           | 176 |
| Como fazer tabuada com os laços.....                      | 178 |
| Tabuada em C++ com FOR.....                               | 178 |
| Tabuada em C++ com WHILE e DO WHILE.....                  | 178 |
| Somatório e Fatorial com laços.....                       | 180 |
| Somatório usando laços em C++.....                        | 180 |
| Fatorial usando loopings em C++.....                      | 182 |
| Exponenciação usando laços em C++.....                    | 184 |
| Exponenciação na Matemática.....                          | 184 |

|   |     |
|---|-----|
| Exponenciação com laços no C++.....                                       | 184 |
| Laços aninhados em C++ - Laço dentro de laço.....                         | 187 |
| Estruturas de Repetição aninhadas em C++.....                             | 187 |
| Usando laços aninhados em C++.....  | 187 |
| Como usar laços aninhados em C++.....                                     | 188 |
| Exemplo de uso de laços aninhados.....                                    | 190 |
| Laço dentro de laço de laço dentro de laço.....                           | 192 |
| Mega-Sena com C++.....  | 194 |
| A loteria da Mega-Sena.....   | 194 |
| Quantos palpites são possíveis na Mega Sena.....                          | 195 |
| Exibindo todos os palpites da Mega-Sena.....                              | 196 |
| As instruções BREAK e CONTINUE do C++.....                                | 197 |
| Instrução BREAK em C++.....   | 197 |
| O comando CONTINUE em C++.....  | 199 |
| Números primos em C++ - Como descobrir.....                               | 201 |
| Números Primos na Matemática.....   | 201 |
| Como Descobrir se um número é primo.....                                  | 202 |
| Otimizando a busca por primos.....  | 203 |
| Achando primos num intervalo.....   | 204 |
| Exercícios de Laços e Loopings.....                                       | 206 |
| Exercícios de WHILE, DO WHILE e FOR em C++.....                           | 206 |
| Soluções.....   | 212 |
| Funções.....  | 219 |
| Função em C++ - O que é, Como funciona e Como criar e usar ?.....         | 220 |
| Função em C++: O que é e Para que serve.....                              | 220 |
| Como criar uma função em C++.....   | 221 |
| Como chamar e usar uma função.....  | 222 |
| Exemplo de uso de funções em C++.....                                     | 223 |
| Quando usar uma função.....   | 224 |
| Como Receber Informações de uma Função - O comando RETURN.....            | 225 |
| Trocando dados com funções em C++.....                                    | 225 |
| O comando RETURN em Funções.....  | 226 |
| Exemplo de uso de RETURN em Funções.....                                  | 228 |
| Boas práticas de funções.....   | 229 |
| Enviando dados para Funções em C++ - Parâmetros e Argumentos.....         | 231 |
| Enviando Dados para Funções.....  | 231 |
| Exemplo de parâmetros e argumentos em C++.....                            | 232 |
| Parâmetros e Argumentos em C++.....                                       | 233 |
| Protótipo de Função - Como programar uma calculadora completa em C++..... | 235 |
| Programando uma Calculadora em C++.....                                   | 235 |
| Protótipos de funções em C++.....   | 238 |
| Variável Local, Global, Constante Global e Variável estática.....         | 242 |
| Variável Local.....   | 242 |
| Variável Global.....  | 244 |
| Constantes Globais.....   | 245 |
| Variável estática local.....  | 247 |
| Argumentos Padrão e Omissão de Argumentos.....                            | 249 |
| Argumento Padrão em C++.....  | 249 |
| Omissão de argumentos.....  | 250 |
| Regras no uso de argumentos padrão.....                                   | 251 |



|   |     |
|---|-----|
| Parâmetros e Variáveis de Referência.....                       | 254 |
| Passagem por Valor.....   | 254 |
| Parâmetro de Referência: &.....                                 | 255 |
| Mais sobre parâmetros de referência em C++.....                 | 257 |
| Exercício de passagem de valor e referência.....                | 258 |
| Sobrecarga de Funções: Parâmetros e Argumentos diferentes.....  | 259 |
| Tamanhos de Parâmetros e Argumentos em Funções no C++.....      | 259 |
| Sobrecarga de Funções em C++.....                               | 260 |
| Função Recursiva em C++.....                                    | 263 |
| Função Recursiva.....   | 263 |
| Como usar recursão com funções.....                             | 264 |
| Somatório com recursão.....                                     | 265 |
| Fatorial com recursão.....                                      | 266 |
| Sequência de Fibonacci com recursividade.....                   | 267 |
| Recursão x Iteração.....  | 269 |
| Exercício de C++.....   | 270 |
| MDC em C++ - Como calcular o Máximo Divisor Comum.....          | 271 |
| O que é MDC ?.....  | 271 |
| Como calcular o MDC.....  | 271 |
| Como calcular o MDC com C++ e recursão.....                     | 272 |
| A função exit() do C++.....                                     | 274 |
| A função exit().....  | 274 |
| Argumentos da função exit().....                                | 275 |
| Exercícios de Funções.....                                      | 277 |
| Soluções.....   | 279 |
| Par ou Ímpar em C++: Como programar.....                        | 279 |
| Par ou Ímpar em C++: Código comentado do Jogo.....              | 279 |
| Código do jogo Par ou Ímpar em C++.....                         | 280 |
| Game em C++: Adivinhe o número sorteado.....                    | 283 |
| Código comentado do jogo em C++.....                            | 283 |
| Código do Jogo em C++.....                                      | 285 |
| Arrays / Vetores.....   | 288 |
| Vetor/Array em C++: O que são? Para que servem?.....            | 289 |
| O que é um Array (ou vetor) ?.....                              | 289 |
| Para que serve um vetor (ou array) ?.....                       | 290 |
| Arrays em C++ : Como Declarar, Inicializar, Acessar e Usar..... | 291 |
| Como declarar um Array em C++.....                              | 291 |
| Como inicializar um Array.....                                  | 291 |
| Como acessar os elementos de um Array.....                      | 292 |
| Como usar Arrays em C++.....                                    | 293 |
| Probabilidade e Estatística em C++: Jogando Dados.....          | 297 |
| Lançar Dados em C++.....  | 297 |
| Como achar o Maior e o Menor elemento de um Array.....          | 301 |
| Fazendo buscas em Arrays.....                                   | 301 |
| Arrays em Funções.....  | 304 |
| Arrays como argumentos para Funções.....                        | 304 |
| Retornar Array: Passagem por referência.....                    | 306 |
| Como copiar Arrays.....   | 307 |
| Ordenar elementos de um Array em C++.....                       | 310 |
| Ordenar elementos (sorting).....                                | 310 |

|   |     |
|---|-----|
| Como ordenar um Array em C++.....               | 310 |
| Exercícios de Arrays.....                       | 313 |
| Matriz em C++ : Array de Arrays.....            | 315 |
| Array de Arrays - O que é? Para que serve?..... | 315 |
| Como declarar uma Matriz em C++.....            | 316 |
| Como inicializar uma Matriz em C++.....         | 317 |
| Matrizes em Funções.....                        | 319 |
| Como Passar Matriz para Função.....             | 319 |
| Matriz em C++: Passagem por referência.....     | 320 |
| Exercício de Matriz em C++.....                 | 322 |
| Exercícios de Arrays em C++.....                | 323 |
| Questões de Arrays em C++.....                  | 323 |
| Jogo da Velha em C++.....                       | 325 |
| Lógica do Jogo da Velha em C++.....             | 325 |
| Código do jogo da Velha em C++.....             | 328 |
| Desafio em C++.....                             | 332 |

# Básico da linguagem C++

Seja bem-vindo a seção básica, de introdução ao estudo da linguagem C++.

Vamos te guiar do mais absoluto requisito básico, que é entender o que é um computador, um programa, até instalar o necessário para começar a programar, também te mostrar como criar programinhas pequenos, simples e bem úteis, como calcular médias, áreas e perímetros de figuras geométricas, fazer conversões de temperatura, resolver equações do segundo grau e muito mais.

Qualquer dúvida, não pense 2x em entrar em contato conosco para sanar quaisquer dúvidas.

Sempre tente fazer os exercícios, se esforce o máximo possível, raciocine, pegue um papel, esboce...só depois vá olhar a solução, esse é um passo essencial para se tornar um exímio programador, ok?

Vamos lá!

# Computação e Programação de Computadores

Antes de entrarmos em detalhe sobre o C++, precisamos entender de fato o que é a computação e especificamente, a programação de computadores.

## Computação e Computadores

Já parou pra pensar o que é computação?

Ou no que é um computador?

Essas palavras derivam da palavra **computar**, que basicamente significa contar, calcular.

E é isso que um computador é, uma máquina de calcular. Mas uma incrível máquina, capaz de milhões de cálculos em frações de segundos.

Vale aqui ressaltar que a Teoria da Computação é um ramo da **Matemática**. Ou seja, computação é algo puramente matemático, vamos trabalhar com raciocínio, criatividade, concentração...vamos buscar soluções para problemas, criando *lógicas* (algoritmos, caminhos para resolver problemas).

Como iremos ver ao longo de nosso **curso de C++**, o computador é uma máquina perfeita para fazer armazenamento e processamento de dados, os dois pilares da computação.

## Programação de computadores

O martelo é a ferramenta do pedreiro.

A serra, do marceneiro.

O bisturi, do médico.

Já o computador é a ferramenta do...programador? Sim, mas também da secretária, dos funcionários de uma empresa, das lojas dos comerciantes, dos operadores da bolsa de valores...é tanta utilidade que o computador tem que é difícil de dizer de quem ele é ferramenta principal.

E isso se deve a um motivo: a **programação**.

Os computadores são programáveis, você pode adaptar ele para uma infinidade de propósitos diferente, através da programação de computadores.

Sendo mais específico, o computador é uma máquina que obedece comandos.

E aqui que vem o segredo: esses comandos, ou instruções, são dados por um programador.

Assim, se você souber a programar, vai ter o poder de fazer o computador obedecer seus comandos, para fazer, literalmente, qualquer coisa, tudo depende da sua imaginação (e conhecimentos de programação).

## Hardware e Software

Todo computador possui duas categorias: o hardware e o software.

O hardware se refere aos componentes reais do computador, físicos, que você pode ver e tocar.

Embora falamos de 'computador' como um dispositivo só, ele é, na verdade, um conjunto de hardwares, onde os principais dispositivos são:

- CPU - Unidade de processamento central, o 'coração' do computador, é lá onde as instruções vão 'acontecer'. Basicamente, a CPU recebe algumas informações de entrada, faz algum tipo de processamento, e retorna uma saída.
- Memória - Dispositivo que guarda, armazena informações. Já notou que você baixa algum arquivo, desliga o computador e quando liga novamente esses arquivos ainda estão lá? Pois é, pra isso acontecer eles precisam ser 'guardados' em algum lugar, no caso, na memória.
- Dispositivos de entrada - São hardwares que irão mandar informações pra CPU, como o teclado (cada tecla que você aperta, um comando/informação é enviado), mouse (posição dele, o ato de clicar etc), joysticks etc
- Dispositivos de saída - São os hardwares responsáveis por emitir informação, do computador para o mundo externo. O exemplo mais clássico é o monitor, que exibe informações para você. Impressora também é um dispositivo de saída (sai informação na folha de papel)

Já o software nada mais é que o programa rodando em seu computador. Ele é abstrato, você não pode 'ver' ele, apenas seu resultado. Ele existe na forma de bits de informação, em sua máquina e você não vê esses bits.

O principal software de um computador é, sem dúvidas, o sistema operacional (como Windows e Linux), que é o programa responsável por lidar com o hardware e as informações recebidas do usuário.

## **Linguagem de Programação**

Você já sabe uma linguagem, o Português.  
O que é possível fazer com ele?

Você pode escrever livros, textos para sites, usam a língua portuguesa para fazer letreiros, outdoors, revistas, jornais, vamos a língua sendo usada na TV, no celular, usamos para nos comunicar, conversar...enfim, uma infinidade de coisas.

Linguagem de programação é a mesma coisa, uma língua. Mas, é a língua que usamos para nos comunicarmos com um computador.

Não podemos fazer: "Ei computador, faz aí um jogo que é assim e assim"  
Obviamente ele não vai entender, mas se você usar uma linguagem de programação (como o C++, uma das melhores, mais importante e poderosa), ele vai entender.

Com a linguagem de programação, podemos criar instruções específicas para o computador obedecer, podemos criar imagens, vídeos, artes gráficas, menus, botões, programar sistemas de carros, aviões, jogos, sistema para eletrônicos, como geladeiras, microondas, etc etc etc...as possibilidades são infinitas também.

```

17 string sInput;
18 int iLength, iN;
19 double dblTemp;
20 bool again = true;
21
22 while (again) {
23     iN = -1;
24     again = false;
25     getline(cin, sInput);
26     system("cls");
27     stringstream(sInput) >> dblTemp;
28     iLength = sInput.length();
29     if (iLength < 4) {
30         again = true;
31         continue;
32     } else if (sInput[iLength - 3] != '.') {
33         again = true;
34         continue;
35     } while (++iN < iLength) {
36         if (isdigit(sInput[iN])) {
37             continue;
38         } else if (iN == (iLength - 3)) {

```

Exemplo de um trecho de código da linguagem de programação C++

## Programa de Computador

Um programa de computador, ou *software*, nada mais é que um conjunto de instruções que seu computador deve seguir para realizar alguma tarefa.

Vamos supor que você queira criar um programa que exibe o preço de um produto de uma loja, mas com um desconto. Seu computador deve seguir os seguintes passos, na ordem:

1. Perguntar o preço do produto
2. Aplicar o desconto (cálculo matemático)
3. Exibir a informação do produto com desconto

Nosso código C++, aplicando 10% de desconto, ficaria:

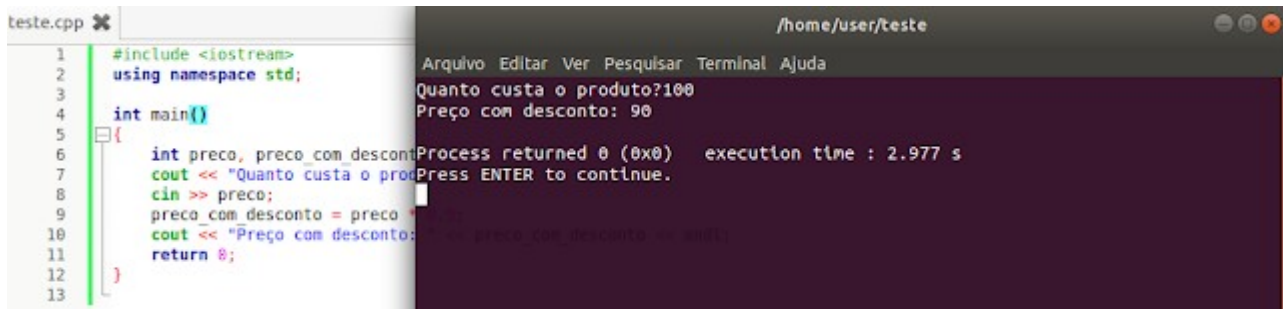
```

#include <iostream>
using namespace std;

int main()
{
    int preco, preco_com_desconto;
    cout << "Quanto custa o produto?";
    cin >> preco;
    preco_com_desconto = preco * 0.9;
    cout << "Preço com desconto: " << preco_com_desconto << endl;
    return 0;
}

```

O resultado exibido para o usuário seria:



```
teste.cpp x
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int preco, preco_com_desconto;
7      cout << "Quanto custa o produto?";
8      cin >> preco;
9      preco_com_desconto = preco * 0.9;
10     cout << "Preço com desconto: ";
11     cout << preco_com_desconto << endl;
12     return 0;
13 }
```

```
/home/user/teste
Arquivo Editar Ver Pesquisar Terminal Ajuda
Quanto custa o produto?100
Preço com desconto: 90
Process returned 0 (0x0)   execution time : 2.977 s
Press ENTER to continue.
```

## Como ser um bom programador C++

Você aprendeu a língua portuguesa em algumas semanas? Ou meses? Não, né.

Logo, com a linguagem de programação C++, é a mesma coisa. Vamos começar aos poucos, bem do básico, como se ensinássemos as vogais, depois as consoantes, depois juntamos e formamos umas sílabas...

Com a língua portuguesa você pode tanto criar o próximo clássico da literatura, como pode escrever uma letra horrível e pobre de funk. Tudo vai depender de uma coisa: do tanto que você estudar.

Não tem mistério: é sentar a bunda na cadeira e estudar, pensar, pensar, pensar mais um pouco, tentar...você vai errar muito, até hoje eu erro e todos os melhores programadores erram bastante, é praticamente impossível criar um programa de porte razoável sem vários problemas, que chamamos de *bugs*.

Não desista. Você pode chorar em posição fetal, tudo bem, faz parte. Mas depois volte e tente novamente, é assim que se aprende. Quanto mais estudar, tentar e se dedicar, melhor programador vai ser.

E não adianta só ler. Se pretende somente ler este curso, está perdendo seu tempo.

Tem que praticar, colocar a mão na massa e ir tentando programar, ok?

Não tente copiar e colar nada, escreva tudo na mão, tente fazer suas próprias soluções.



Não existe uma 'maneira de criar tal programa' ou 'passo a passo para criar um jogo', cada programador faz do seu próprio jeito, assim como cada humano faz uma redação da sua própria maneira.

## **Fontes de estudo**

Computador

Teoria da computação

# Linguagem C++: O que é? Para que serve? Como funciona? Onde é usada ?

No tutorial passado, explicamos [o que é a computação e programação de computadores](#), e lá já demos uma introdução sobre o que são linguagens de programação.

Neste tutorial de nosso **curso de C++**, vamos entrar em mais detalhes sobre esta poderosa e fantástica linguagem, explicando o que ela é de fato, para que serve, onde e como deve ser usada, para quais propósitos e outras informações importantes!

## O que é C++ ?

Resumindo, nada mais são que uma maneira, uma língua, de nos comunicarmos com os computadores, para programá-los a fazer o que quisermos, visto que são máquinas extremamente rápidas nos cálculos.

Dito isso, podemos dizer que C++ é uma linguagem de programação compilada, multi-paradigma e de uso geral. Ou seja...oi ?! Calma, vamos aprender cada detalhe sobre isso no decorrer do curso, basta você ir estudando na ordem do sumário, passo a passo, de maneira progressiva e sem pressa.

Também odeio essas definições e explicações técnicas, mas faz parte. O que podemos te adiantar é que é uma linguagem tão boa, tão versátil e tão poderosa, que é uma das mais usadas no mundo, há décadas, e tem tanto, mas tanto programa feito usando ela, que a demanda por programadores C++ ainda é absurdamente alta, e o mais bacana: pouca gente estuda e aprende, principalmente por faltar conteúdo bom e gratuito sobre ela.

Mas é aí que entra o [C++ Progressivo](#): vamos nos esforçar ao máximo para te proporcionar o maior, melhor e mais completo material sobre a linguagem C++, de maneira totalmente gratuita, online e sem pedir nada (nem login exigimos, é chegar, estudar, praticar e virar programador \$\$).

C++ é uma linguagem tratada com meio-nível, um meio termo entre linguagem de alto nível (como [Python](#)) e baixo nível (como [Assembly](#)), ou seja, possui características desses dois 'mundos'. O C++ é realmente foda, não tem outra definição.

Ela é também multiplataforma, ou seja, funciona em Windows, Mac, Linux...

## C e C++

Como o nome pode sugerir, a linguagem C++ deriva da [linguagem C \(veja: C Progressivo\)](#), o ++ significa incremento. Ou seja, podemos dizer que o C++ é uma linguagem C *incrementada*.

Seu desenvolvimento foi feito pelo querido Bjarne Stroustrup, lá por 1983, quando chamou a linguagem de *C com classes* (depois você vai entender porque).

Na época, o C era utilizado para criar e desenvolver o Unix, sistema operacional dominante na época (que deu origem a diversos sistemas operacionais atuais, como o Linux - por favor, estude e tente usar o Linux em seu PC, vale a pena, você vai ser um programador diferenciado).

Porém, o C era uma linguagem mais 'crua', 'seca' e 'direta'.

Bjarne queria dar uma melhoria nisso, dar umas novas funcionalidades ao C, deixar ele mais fácil, flexível, com mais possibilidades e implementações que facilitam a vida do programador.

Sem dúvidas, uma dessas implementações mais importantes é o fato de ser orientada a objetos e ter a STL (um pacote de código e funcionalidades prontas para o programador usar, sem precisar ficar 'reinventando a roda' - entenderemos melhor e com detalhes isso no futuro, em nosso curso).

Vale ressaltar que o C sempre foi uma linguagem rápida e eficiente, característica essa que foi devidamente respeitada, na criação do C++.

## Programas desenvolvidos em C++

É tanta coisa, mas tanta coisa feita em C++, que foi difícil separar o que seria interessante de mostrar. Mas, não tenha dúvidas, praticamente qualquer coisa que você quiser fazer, pode ser feita em C++. Vamos ver apenas **alguns** exemplos:

- Microsoft Windows (sim, na sua maior parte, o sistema operacional mais famoso do mundo é feito em C++ !!!), e seus programas, como o Office (sim, Word, Excel, Powerpoint e tudo mais que você usa, é C++)
- Adobe Photoshop
- MySQL (sistema de banco de dados mais utilizado)
- Navegadores, como: Mozilla Firefox e Internet Explorer
- Sistemas web, como Google, Youtube, Amazon.com
- Spotify
- Games (muitos, mas muitos jogos mesmo usam C++, principalmente os que usam muita memória e requisitos do sistema, pois C++ é uma linguagem muito rápida e eficiente)  
Jogos: Doom III engine, Counter Strike, Sierra On-line Birthright, Hellfire, Football Pro, Bullrider I & II, Trophy Bear, Kings Quest, Antara, Hoyle Card games suite, SWAT, Blizzard StarCraft, StarCraft: Brood War, Diablo I, Diablo II Lord of Destruction, Warcraft III, World of Warcraft, Starfleet Command, Invictus, PBS's Heritage: Civilization and the Jews, Master of Orion III, CS-XII, MapleStory, WOW
- Vídeo-games, como PC, Xbox One, PS4 e Switch
- Programas críticos, como servidores e microcontroladores
- Novas linguagens de programação
- Sistemas embarcados
- Várias bibliotecas de Machine Learning estão prontas, feitas em C++
- Programas científicos e de universidades
- Compiladores
- Renderização de imagens, animações, gráficos e vídeos, devido ao alto desempenho e performance que o C++ pode oferecer
- GUI (*Graphical User Interface*), ou seja, aquelas interfaces gráficas que você vê (janelinhas, menus, botões, formulários etc) em programas, em sua grande parte são feitas em C++
- Aplicações de bancos, corretoras e outros sistemas financeiros, pois C++ é extremamente confiável e preciso
- Novos sistemas que requerem compatibilidade com C

- Robótica e automação
- Telecomunicações

## Onde C++ não é (tão) usado

Não existe a melhor linguagem de programação, bote isso na sua cabeça. O que existe é a melhor linguagem de programação para um fim e propósito específico, quem diz que uma é melhor ou pior, são *haters* e gente com visão limitada no ramo da computação.

Se te perguntarem em uma entrevista de emprego (e vão), qual a melhor linguagem, responda: depende do projeto que deseja fazer.

Assim, é importante também saber onde não se usa muito C++:

- Aplicações *front end* da Web: por ser uma linguagem compilada, ela não vai rodar diretamente no browser, diferente de linguagens interpretadas, como [JavaScript](#), que é 'lido' linha por linha
- Aplicativos móveis, como Android. Nesse caso, a linguagem [Java](#) domina
- Aplicações do lado do servidor (*server side scripting*), lá as linguagens de script ainda dominam, como [PHP](#) e [Python](#)
- Programas bem de baixo nível, ou seja, que 'conversam' diretamente com o hardware, nesse caso recomendamos estudar C ou Assembly
- Aplicações onde a eficiência máxima possível é exigida

Falando um pouco sobre este último item: embora C++ seja um C implementado, isso não significa que seja melhor em C que tudo, se fosse, o C não existiria.

O C++ tem algumas ferramentas a mais, digamos assim. Porém, isso tem um custo, é mais 'pesado', já que vem mais coisa embutidas. Então, se você deseja a eficiência máxima (como um sistema operacional super rápido e confiável, como Linux ou um lançamento de um foguete ou servidor de uma usina nuclear, talvez você queira a máxima capacidade e confiabilidade possível, e vai usar um C ou Assembly da vida).

Nas outras 99% possibilidades do dia-a-dia (a menos que queira desenvolver sua própria usina nuclear), o C++ vai ser mais que perfeitamente suficiente

para o que você quer desenvolver (programa ou jogo, para as pessoas usarem em seus computadores).

## **Referências e fontes de estudo**

Compiled language

C++

C++ applications

# **C ou C++? Qual a diferença? Qual é melhor?**

## **Precisa saber C para aprender C++?**

### **Para aprender C++ precisa saber C?**

A dúvida inicial de quem se interessa por C++ é sempre envolvendo a linguagem C:

“Preciso estudar C antes de estudar C++ ?”

“Meu amigo disse que pra aprender C++ tem que dominar C”

“C++ é uma extensão do C, então pra estudar C++ já precisa saber C”

A apostila C++ Progressivo, vai agora explicar em detalhes e sanar todos esses tipos de dúvida.



### **É necessário saber C para aprender C++ ?**

Não, não é necessário. São duas linguagens de programação independentes, você pode estudar e criar aplicações com uma ou com outra. Se seu amigo disse que C++ é uma extensão do C, ele está certo. Se ele disse precisa estudar C antes de estudar C++, ele está errado e você deve se afastar dele.

### **Qual a diferença entre C e C++ ?**

C++ é uma extensão do C.

É o C com mais recursos e funções. E a principal diferença é que C++ é multi-paradigma: você pode programar em C++ com o paradigma Estruturado (o C é somente estruturado) ou com o paradigma de Orientação a Objetos (como a linguagem Java, por exemplo).

O símbolo ++, em programação, significa incremento. Representa o fato que o que é possível fazer com C, é possível fazer com C++.

C++ é a linguagem C incrementada. É tanto que seu criador, Bjarne Stroustrup, batizou ela como "C com classes".

Se C++ é uma extensão do C, então...

## **C++ é melhor que a linguagem C ?**

Não existe linguagem melhor que a outra, existe linguagem mais adequada para determinado tipo de problema.

Sim, o C++ tem mais recursos, mas isso não vem de graça.

Sua implementação é mais complexa, tem mais coisas ocorrendo por debaixo dos panos. Logo, C++ não é tão eficiente quanto C, consome mais memória e processamento.

## **Onde se usa C e onde se usa C++ ?**

Como dissemos, C é mais eficiente, é mais rápido. Programar em C é programar próximo ao 'metal', é ter acesso direto a memória do computador. Isso se chama programação em baixo nível.

O C++ também consegue isso, mas C++ também oferece a oportunidade de trabalhar em alto nível.

Porém, algumas aplicações exigem o máximo de velocidade e eficiência possível, por isso o C é usado para a criação de Sistemas Operacionais (como Linux e Windows) e até mesmo para trabalhar com microcontroladores.

Já C++ não serve para esses propósitos, porém serve para outros.

O problema do C ocorre quando as aplicações começam a ficar muito grandes, o que dificulta o controle e manutenção do código. É aí que entra o C++.

Devido sua Orientação a Objetos, e ao fato de possuir mais recursos e opções (como Templates), facilita muito na hora de programar e aumenta incrivelmente a eficiência do programador C++.



Ou seja, em um mesmo intervalo de tempo, o programador C++ consegue ser mais eficiente que um programador C fazendo a mesma coisa, pois o C++ tem mais coisas na ponta da agulha, pronta para o uso do programador.

Um exemplo de uso da linguagem C++ são os jogos de alto rendimento. Eles são muito complexos para se fazer, por isso não é recomendado o uso de C ou Assembly.

Porém, eles precisam de uma eficiência muito grande, por isso não pode ser feito numa linguagem de mais alto nível, como Java ou C#.

É aí que entra o C++: ele é, ao mesmo tempo, muito mais eficiente que a maioria das linguagens e é mais pronto para o uso do que as linguagens de baixo nível, como o C.

Ou seja, C++ é um equilíbrio entre alto rendimento das linguagens de baixo nível, com a eficiência da programação de linguagens de alto nível.

O Java, por exemplo, também é orientado a objetos e vêm com mais recursos, então...

## **Qual o melhor, Java ou C++?**

Por favor, pare de se perguntar se X ou Y é melhor. É sempre o mesmo que perguntar 'qual o melhor, banana ou maçã?'

Banana é o mais recomendável para fazer vitamina de banana, e maçã é o mais recomendado para fazer suco de maçã.

Não existe linguagem melhor que a outra, e sim mais recomendada para um determinado propósito.

(dica de amigo: dizer que tal linguagem é melhor que outra é coisa de adolescente leigo metido a hacker. Respeite todas as linguagens, profissionais e programas, se quiser ser alguém de respeito no meio).

A diferença começa com o fato de Java ser exclusivamente Orientado a Objetos.

Já C++ pode ser usado para programar tanto como estruturado como orientado a objetos.

A melhor característica do Java é o fato dele ser multi-plataforma, ou seja, você programa em Java e a aplicação roda em Windows, Linux, Mac, mobile e até no papel (brincando).

Isso acontece pois o Java não roda na máquina propriamente dita, e sim em uma máquina virtual (JVM - Java Virtual Machine). E isso pesa, deixando as aplicações Java mais pesadas.

Já o C++ não é pesado e é muito mais eficiente que o Java.

Porém não é multi-plataforma, pois o C++ chega, em alguns aspectos, aos níveis mais baixos de programação.

Por isso, rodar uma aplicação C++ em uma máquina pode ser que não funcione em outra, depende da arquitetura do sistema.

Se C++ é uma extensão do C, então...

## **Os comandos de C rodam em C++?**

Sim, rodam na grande maioria das vezes.

À rigor, C++ não é uma extensão do C, pois há algumas poucas coisas que são possíveis fazerem em C que não são possíveis fazer em C++ (como com o void\* e char\*). Mas no geral, em 99,99% das vezes, seus códigos C serão interpretados corretamente pelo C++.

Porém, como dissemos no início, são duas linguagens independentes, uma NÃO precisa da outra.

Então NÃO MISTURE C COM C++ !

É um péssimo e abominável hábito. Ou programe em C ou programe em C++.

Também não faz muito sentido programar de maneira estrutural em C++, pois você estaria desperdiçando os recursos a mais que o C++ oferece (relacionados com a orientação a objetos).

Se for programar estruturado, vá de C.

## **Como está o mercado de C++ no Brasil ?**

Vamos ser sinceros, e isso implica em uma notícia boa e uma ruim.

A boa que o mercado é ótimo, há uma extrema carência de BONS profissionais de C++ e no geral, ganham bem mais que a maioria dos programadores das outras linguagens.

A ruim é que não é uma linguagem fácil nem simples de aprender. Sua curva de aprendizado é maior que as outras, e se quiser ir em frente com C++ você deve estudar bastante.

E aí, tá afim de aprender C++ ?

Bem vindo a apostila de C++ online: Curso C++ Progressivo

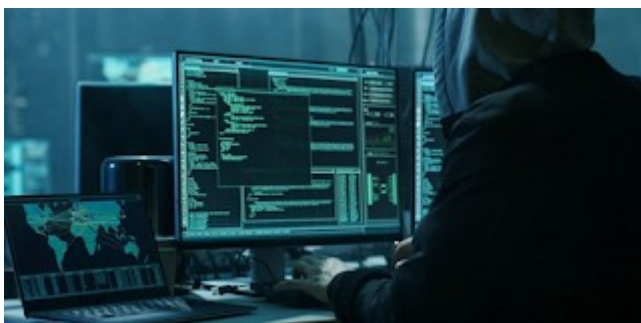
# Como começar a programar em C++

Agora que já explicamos [o que é e para que serve a linguagem C++](#), vamos aprender a baixar, instalar e configurar nosso ambiente de programação, deixando tudo pronto para começarmos a programar de fato.

## O que precisa pra programar em C++ ?

Certamente você já viu alguém programando, seja pessoalmente, ou na TV, numa novela ou filme. Geralmente é alguém num ambiente *dark*, digitando rápido uma série de códigos e palavras indecifráveis em uma tela escura, em meio a uma série de monitores com luzes piscando e texto subindo na tela.

Calma, não precisa de nada 'genial' ou 'super-hacker-do-mal', nem precisa ser super-dotado pra programar. Só sabendo ligar seu computador, entrar no nosso site e ler, você vai ser capaz de começar a programar em C++.



Basicamente vamos precisar apenas de duas coisas:

- Editor de texto
- Compilador C/C++

O editor de texto é o programa que você vai digitar o código C++, como um bloco de notas, por exemplo.

Compilador é o programa que vai pegar esse texto com o código e transformar em linguagem de máquina. Isso se faz necessário porque o *hardware* entende apenas a linguagem binária, de 0's e 1's. Você até pode, teoricamente, digitar em linguagem de máquina.

Mas é mais fácil escrever: `cout << "Curso C++ Progressivo";`

Do que: 011101001 0011001010 001010101 010 0010011 010101001010  
011101001 0011001010 001010101 010 0010011 010101001010 011101001  
0011001010 001010101 010 0010011 010101001010 011101001  
0011001010 001010101 010 0010011 010101001010 011101001  
0011001010 001010101 010 0010011 010101001010

Concorda?

Então, apenas digite os códigos do C++ e deixe que o compilador transforme isso numa linguagem que sua máquina vai entender.

## IDE: CodeBlocks

Não vamos te ensinar a programar usando um editor de texto arcaico, depois indo no *prompt* de comando para compilar, linkar e rodar seus programas, pois isso pode assustar um pouco os iniciantes, dando a impressão que programar é algo complexo.

Vamos te ensinar a programar usando uma IDE (Ambiente de Desenvolvimento Integrado), onde você digita o código, clica num botãozinho, ele compila e roda, mostrando tudo de uma maneira bem simples e fácil.

Indicamos o **Code::Blocks**.

Para isso, acesse: <http://www.codeblocks.org/> (ou digite Code Blocks no Google e entre no primeiro link).

Clique em **Downloads**: <http://www.codeblocks.org/downloads>  
Depois em *download binary release*.

Veja que o Codeblocks é multiplataforma, ou seja, você pode programar tanto no Windows, como no Mac e no Linux. Vá para a área de seu sistema operacional.

Vamos baixar e instalar o Code::Blocks já com o compilador, que no caso se chama MingW (deve ter essa palavra no link de download).

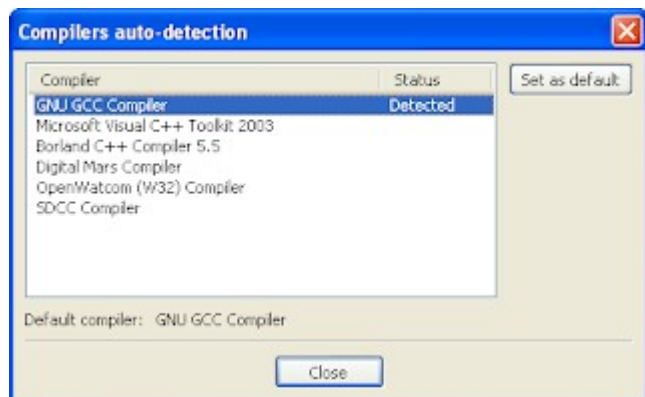
No nosso caso, estamos baixando e instalando por esse link:  
[codeblocks-17.12mingw-setup.exe](#)

Aguarde o download.

Abra o arquivo executável (se tiver no Windows, como administrador do sistema).

Vá apertando em **next** e escolha a opção *full installation*.

Rode o Code Blocks:



E prontinho, seu ambiente de programação já está devidamente montado e configurado.

Falando um pouco mais sobre o Code::Blocks, ele foi desenvolvido em C++. Usa o sistema de plugins, para que você pode baixar e instalar por fora, para incrementar e deixar sua IDE mais poderosa e flexível, inclusive para programar em diversas outras linguagens.

Possui *syntax highlighting*, ou seja, a medida que for digitando, seu código vai ficando 'colorido', realçando palavras-chaves, textos e comandos específicos, que você vai aprender no decorrer de nosso curso.

Também tem auto completar código (você começar a digitar um comando, e ele já entende qual que você quer escrever e completa automaticamente, se você quiser), lista de classes integradas e uma lista de TODO.

É um programa leve, pequeno, consome pouca memória RAM e permite uma variedade de possibilidades, como programar usando diversos tipos de compiladores e sistemas operacionais diferentes.

No próximo tutorial vamos enfim colocar a mão na massa, digitando o código, compilando e rodando nosso primeiro *software*.

## Primeiro Programa: "Olá, mundo" ("Hello, world!")

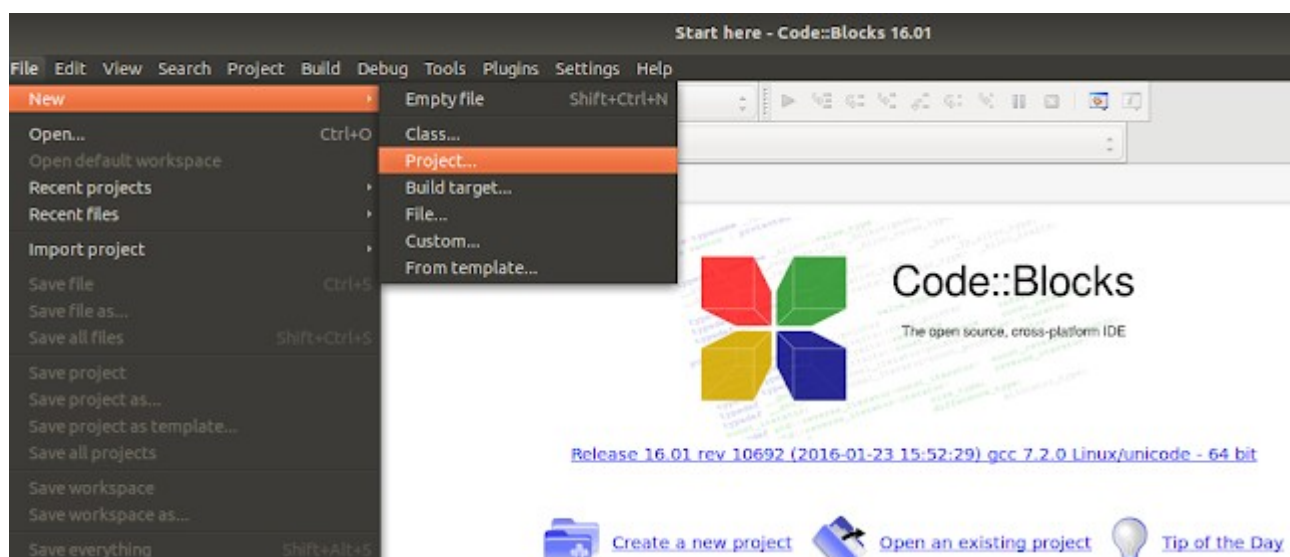
Agora, chega de papo e de teoria.

É hora de colocarmos a mão na massa, criar algum código e colocar ele pra rodar, ver resultado, ver um programa rodando na frente de nossos olhos, o famoso "Olá, mundo" ou "Hello, world" em C++.

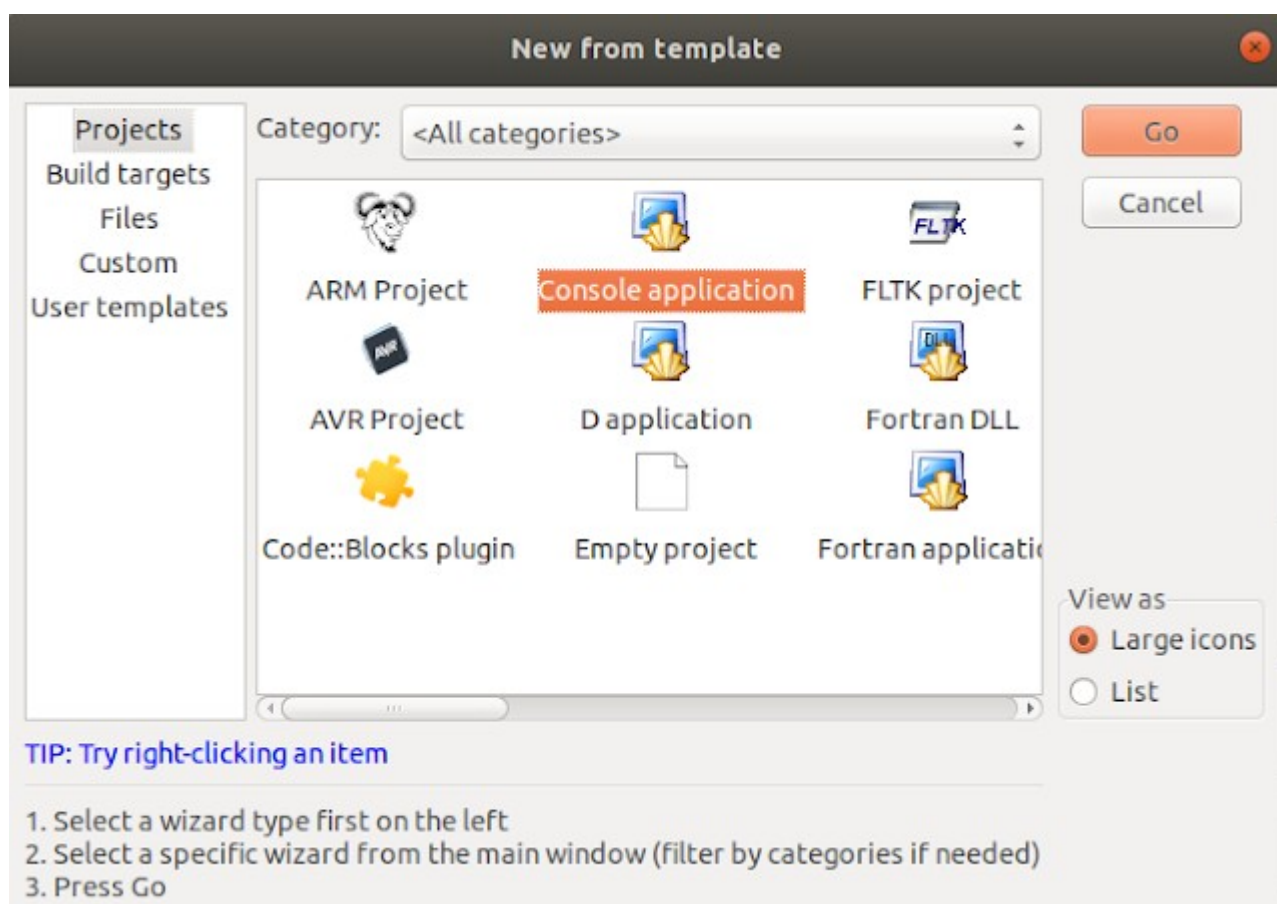
## Criando um Projeto no Code::Blocks

No tutorial passado de nosso curso, te ensinamos [como instalar tudo para começar a programar em C++](#). Agora, abra sua *IDE*, o Code Blocks. Se abrir uma janela de auto-deteção de compilador, escolha GNU GCC Compiler, e vá dando OK.

Vá em *File*, depois em *New* e por fim em *Project* (ou clique direto ali no link na página inicial do Code::Blocks, em *Create a new project*):



Na tela que vai abrir, escolha *Console application* e aperte em *Go*:



Na tela seguinte, escolha C++ e aperte em *Next*.

A seguir, você vai precisar escolher um nome para seu projeto (eu escolhi 'basico', evite acentuação e espaços em brancos), e depois um diretório (pasta) em sua máquina, para salvar o projeto.

Clique em *Next*:



Console application

Please select the folder where you want the new project to be created as well as its title.

Project title:  
basico

Folder to create project in:  
/home/user/Cpp/

Project filename:  
basico.cbp

Resulting filename:  
/home/user/Cpp/basico/basico.cbp

< Back Next > Cancel

Depois em *Finish*:

Console application

Please select the compiler to use and which configurations you want enabled in your project.

Compiler:  
GNU GCC Compiler

☒ Create "Debug" configuration: Debug

"Debug" options  
Output dir.: bin/Debug/  
Objects output dir.: obj/Debug/

☒ Create "Release" configuration: Release

"Release" options  
Output dir.: bin/Release/  
Objects output dir.: obj/Release/

< Back Finish Cancel

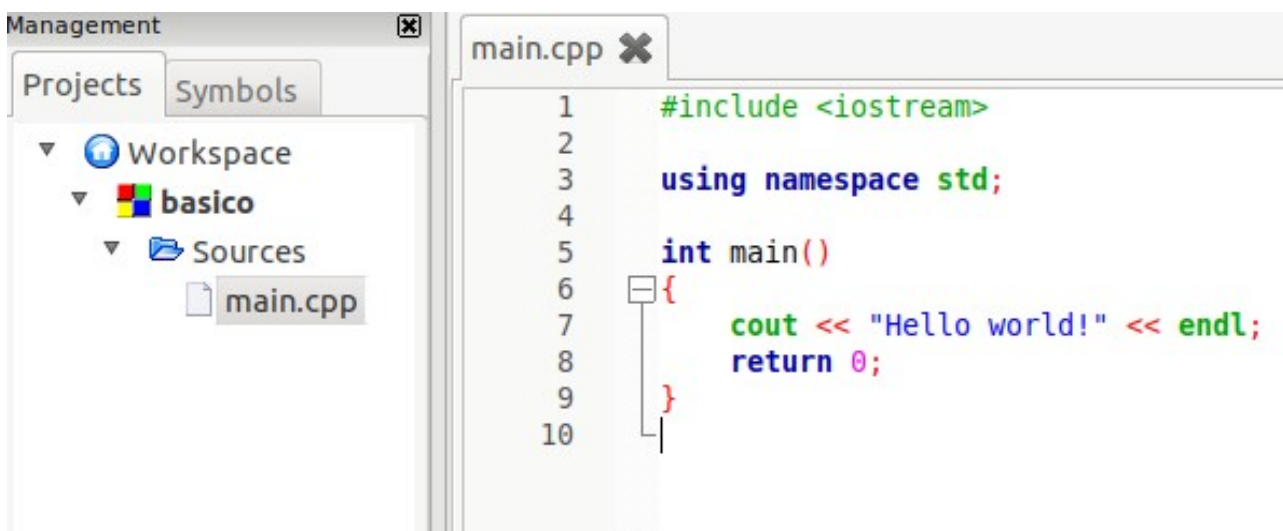
# Programando o primeiro programa em C++

Quando seguir os passos anteriores para criar um projeto de C++ no Code Blocks, vai aparecer a janela abaixo.

Tem um menu do lado esquerdo com seu *Workspace* (local onde você vai salvar seus projetos de C++). Clique no projeto que criou (o meu eu dei o nome de 'basico'). Depois em *sources* (onde vão ficar os código-fonte) e por fim no arquivo *main.cpp*

Esse arquivo, com essa extensão *cpp* (de C++, *C plus plus*), é onde iremos digitar o código C++.

Talvez já aparece um código para você, senão digite o que está abaixo:



Se já apareceu algo escrito, apague tudo. Esse passo é importante e essencial para se tornar um bom programador.

Delete tudo e escreva na mão, palavra por palavra, comando por comando, o seguinte código, com cuidado para não errar:

```
#include <iostream>
```

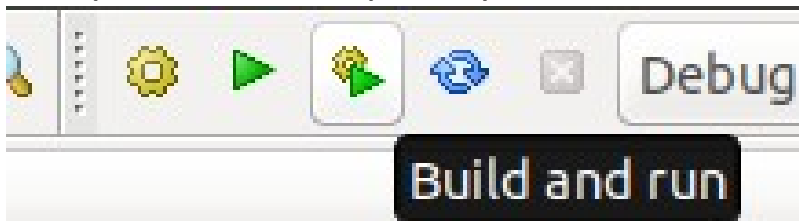
```
int main()
{
    cout << "Olá, mundo";
    return 0;
}
```

## Compilando e Executando um código C++

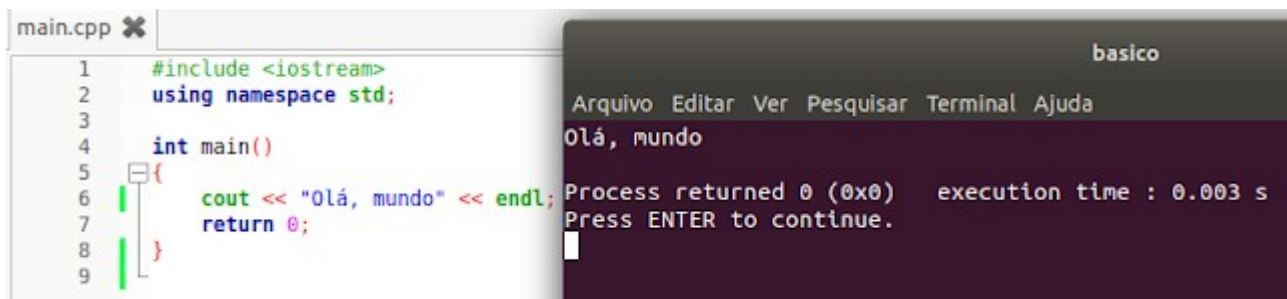
Para ver o programa funcionando, você precisando compilar esse código e executar o arquivo gerado.

Isso pode ser feito por comando num terminal (prompt de comando), aquelas telinhas escuras...mas isso mais assusta que ajuda o iniciante, pois passa a impressão que precisa ser um 'gênio' da computação, pra programar, o que não é verdade.

Simplesmente encontre o símbolo da rodinha dentada e o play, *Build and Run* (construir e rodar), e clique nele:



Vai aparecer umas coisas escritas lá em baixo, no *Build log* e em seguida uma janela preta se abre, e nela está escrito "Olá, mundo" e mais algumas informações do programa, como o tempo de execução:



Prontinho.

Você digitou um código, compilou e executou seu programa. Oficialmente você já é um programador, um mestre paladino das artes computacionais em C++.

NASA e FBI que se cuidem.

Mas vamos nos aprofundar mais, porque aqui o negócio é sério, o ensino é completo.

## Entendendo o código C++

Que tal entendermos um pouco mais o que essa sopa de letrinhas e comandos com palavras esquisitas, que digitamos ?

Não esquite se não entender muita coisa agora (na verdade, se entender algo, já está no lucro), você vai dominar tudo com o passar do tempo, em nosso curso.

Na primeira linha, temos o comando: `#include <iostream>`

Antes do código ser compilado, um treco chamado pré-processador é acionado. Ele pega as linhas começadas em `#`, que são as diretivas de pré-processador. Essa linha nada mais faz que dizer ao pré-processador para incluir (*include*) o arquivo *iostream* no código.

Esse arquivo é o responsável pelas entradas e saídas (*io - in e out*), como receber dados do teclado do usuário e exibir informações na tela. Ou seja, você não precisa programar essas funcionalidades, elas já existem, basta colocar o arquivo *iostream* que esse código é automaticamente portado para seu executável.

A próxima linha está em branco, isso não influencia absolutamente nada, só questão estética e organizacional.

A próxima linha, inicia a função *main*, ou seja, a função principal. Todo programa C++ começa nessa função. Função nada mais é que um bloco de código (tudo entre chaves faz parte da função), vamos estudar com profundidade o tema funções no futuro.

O *int* se refere a um número inteiro, diz que essa função vai retornar um valor inteiro.

De fato, ele retorna o valor 0 (veja a linha: `return 0;`)

O próximo comando é o *cout* (C out, algo como saída do C).

É um comando de saída, ele envia informações para algum lugar. No caso, para a tela de seu computador.

E que informação ele vai enviar? O que estiver após o `<<`

O que tem após esse símbolo é uma *string*, ou seja, um texto. Todo texto/string (grupo de caracteres) é colocado entre aspas duplas.

No nosso caso, nosso texto é "Olá, mundo", logo tudo que está entre as aspas duplas vai aparecer na tela de seu programa. Note que o comando termina com ponto e vírgula ;

Isso simboliza o fim do comando **cout**.

Por fim, o comando *return 0;*, que encerra a função *main()*.

Note que todo código da função está entre { e }

## **Pré-processar, Compilar, Linkar, Carregar e Rodar um programa C++**

Vamos entender um pouco agora o que ocorre por debaixo dos panos, quando clicamos no botão *Build and Run*.

O primeiro passo para programar, é digitar o texto, o código, em um editor de textos.

No caso, como o Code::Blocks é uma IDE (ambiente completo de programação), ele já vem com esse editor, nada mais é que esse espaço em branco que a gente digitou nosso código, como se fosse um bloco de notas.

Quando clicamos em compilar, antes da compilação o pré-processador é executado, e sua função nada mais é que caçar as diretivas de pré-processamento, isso nada mais é que fazer algumas manipulações no código, antes de compilar, como por exemplo, incluir alguns arquivos externos em seu programa, com códigos já prontos para você não ter que reescrever e também faz algumas substituições de texto.

No futuro estudaremos em mais detalhes o pré-processador.

## Saída simples em C++: cout, <<, endl, \n e Outros caracteres especiais

No tutorial anterior, aprendemos a [criar o primeiro programa em C++](#), famoso *Hello, world!* ou *Olá, mundo!*.

Para tal, usamos um comando bem especial: o **cout**. Vamos aprofundar nossos estudos neste assunto.

### Saída simples (imprimindo coisas na tela): **cout** e <<

O que fizemos no primeiro programa em C++ foi, simplesmente, escrever uma mensagem na tela.

Dizemos que *imprimimos* algo no *console* (telinha preta de comandos), ou mais especificamente, fizemos uma saída do C++ para a tela.

Especificamente o *cout* é um treco chamado *objeto*. Especificamente, um objeto da classe *ostream*.

Não se preocupe, por hora, com esses termos. Vamos esmiuçar ele em nossos tutoriais de orientação a objetos em C++.

Esse comando é *standard output stream*, ou seja, é a saída padrão de informação. Ou seja, através do *cout* conseguimos enviar informações para a saída padrão, que é o terminal de comando, a janelinha preta que aparece na sua frente, quando rodamos o programa.

Vamos agora exibir a mensagem: "Bem vindos ao curso C++ Progressivo":

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

```
int main()
{
    cout << "Bem vindos ao curso C++ Progressivo";
    return 0;
}
```

Rode esse programa e veja o resultado.

Já o operador << sinaliza que estamos enviando informação (no caso, o texto) para o comando cout:  
cout << algo (algo está sendo enviado para a saída padrão).

Podemos reescrever o programa da seguinte maneira:

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

int main()
{
    cout << "Bem vindos ao curso " << "C++ Progressivo";
    return 0;
}
```

Bacana, né?

Vamos agora escrever na tela:

```
Bem vindos ao curso
C++ Progressivo
```

Basta usarmos o comando **cout** duas vezes, concorda? Veja:

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

int main()
{
    cout << "Bem vindos ao curso";
    cout << "C++ Progressivo";
    return 0;
}
```

Não se esqueça do ponto e vírgula ao término de cada comando. Isso é universal em C++.

O resultado fica:

```
Bem vindos ao cursoC++ Progressivo
```

Opa! Peraí! Ficou tudo grudado, qual foi o problema?

## Quebra de linha: **endl** e **\n**

Uma string, que é o mesmo que um texto, é nada mais que uma sequência de caracteres.

Após a primeira frase, "Bem vindos ao curso" existe um caractere, embora não possamos ver.

É a quebra de linha, é o mesmo que ocorre quando você dá um enter em um texto, ele pula, quebra pra próxima linha.

Podemos resolver isso usando o comando **endl** (*end line*), que é um manipulador de *stream*, veja:

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

int main()
{
    cout << "Bem vindos ao curso " << endl;
    cout << "C++ Progressivo";
    return 0;
}
```

Uma outra maneira de quebra a linha, é usando o caractere **\n**.

No caso, ele vai dentro da string, ou seja, como se fosse um caractere de texto mesmo:

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

int main()
{
    cout << "Bem vindos ao curso\n";
    cout << "C++ Progressivo";
    return 0;
}
```

Veja que podemos inclusive fazer isso usando apenas um comando *cout*:

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

int main()
{
    cout << "Bem vindos ao curso\nC++ Progressivo";
}
```



```
    return 0;  
}
```

O caractere '\n' ocupa 1 byte de memória na execução do programa. Já o **endl** chama uma função do sistema, para fazer o mesmo efeito do \n.

## Caracteres especiais em C++

\n é chamado de caractere especial. Vejamos alguns outros:

- \t - tem o mesmo efeito do TAB num editor de texto
- \r - *Carriage return*: posicione o cursor da tela no início da linha atual, não avança para a próxima linha.
- \a - *Alert*, em máquinas antigas produzia um som (um bipe)
- \\ - Usado pra printar o caractere \
- \' - Usado para printar o caractere ' (aspas simples)
- \" - Usado para printar o caractere "" (aspas duplas)

## Fontes de estudo e consulta

<http://www.cplusplus.com/reference/iostream/cout/>

<http://www.cplusplus.com/reference/ostream/endl/>

## Exercícios de saída simples em C++: cout, <<, endl e \n

Agora que já aprendemos a exibir saídas simples em C++, usando o comando cout, vamos praticar um pouco mais.

Quero ver você tentando, até conseguir, resolver os exercícios abaixo.

### Exercícios de C++

- 1. Frase na tela** - Crie um programa que exiba na tela a frase "O primeiro programa a gente nunca esquece!".
- 2. Etiqueta** - Elabore um programa que escreva seu nome completo na primeira linha, seu endereço na segunda, e o CEP e telefone na terceira, usando **endl**.
- 3.** Repita o programa anterior, agora usando `\n` ao invés de `endl`. Você consegue fazer tudo em um comando `cout` só? Tente.
- 4. Letra de música** - Faça um programa que mostre na tela uma letra de música que você gosta (proibido letras do Justin Bieber).
- 5. Mensagem** - Escreva uma mensagem para uma pessoa de quem goste. Implemente um programa que imprima essa mensagem, tire um print e mande pra essa pessoa. Diga que foi um vírus que algum hacker instalou em seu computador.
- 6. Ao site** - Faça um programa que mostre na tela o que você deseja fazer usando seus conhecimentos de C++.
- 7. Quadrado** - Escrever um programa que mostre a seguinte figura:

```
XXXXX
X    X
X    X
X    X
XXXXX
```

**8. Tabela de notas** - Você foi contratado por uma escola pra fazer o sistema de boletim dos alunos. Como primeiro passo, escreva um script que produza a seguinte saída:

| ALUNO(A) | NOTA  |
|----------|-------|
| =====    | ===== |
| ALINE    | 9.0   |
| MÁRIO    | DEZ   |
| SÉRGIO   | 4.5   |
| SHIRLEY  | 7.0   |

**9. Letra grande** - Elabore um script para produzir na tela a letra C, de C++ Progressivo. Se fosse 'L', seria assim:

```
L
L
L
LLLLL
```

**10. Menu** - Elabore um script que mostre o seguinte menu na tela:

Cadastro de Clientes

0 - Fim

1 - Inclui

2 - Altera

3 - Exclui

4 - Consulta

Opção:

**11. Pinheiro** - Implemente um programa que desenhe um "pinheiro" na tela, similar ao abaixo. Enriqueça o desenho com outros caracteres, simulando enfeites.

```

  X
 XXX
XXXXX
XXXXXXX
XXXXXXXXX
XXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXX
  XX
  XX
 XXXX
```

# Tipos de Dados, Variáveis e Atribuição em C++: int, char, float, double e bool

Neste tutorial de nosso **curso de C++**, vamos aprender a trabalhar com dados (informações) em programação.

## Armazenando Informações

Um dos pilares da computação, é a armazenagem de informações, de dados.

O computador faz, basicamente, duas coisas: cálculos e armazenagem de dados.

Quase sempre, faz os dois: processamento de dados, como armazenar, alterar, excluir, fazer operações matemáticas etc.

Neste tutorial, embora longo seja bem simples, vamos introduzir os conceitos de tipos de dados e variáveis em C++, conhecimentos altamente importantes para trabalharmos com programação de computadores.

Vamos conhecer os mais variados tipos de dados, quais suas características e funções.

De antemão, saiba que toda variável que você vai usar para armazenar dados ao longo de seus programas, devem ser declaradas previamente.

Usar uma variável que ainda não foi declarada acarretará em erros na hora da compilação.

## O tipo de dado inteiro: **int**

Este tipo de dado serve para armazenar, especificamente, dados numéricos do tipo inteiro.

Os inteiros são:

- ..., -3, -2, -1, 0, 1, 2, 3, ...

Vamos agora criar e declarar uma variável chamada *idade*:

- `int idade;`

Pronto. Nesse ponto, o computador vai pegar um local da memória do seu computador e reservar para a variável *idade*. E o que está armazenado? Inicialmente, 'lixo'.

Vamos agora atribuir o valor 18 para esta variável:

- `idade = 18;`

Agora, sempre que usarmos a variável *numero*, ela será substituída pelo número inteiro 18.

Veja:

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

int main()
{
    int idade;
    idade = 18;

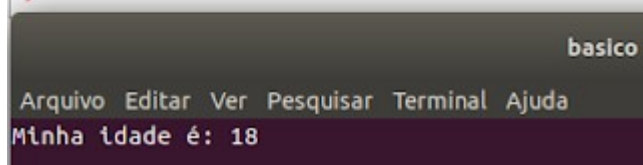
    cout << "Minha idade é: "<< idade << endl;
    return 0;
}
```

Note que o *cout* não imprime a palavra 'idade', e sim o valor contido dentro desta variável.

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

int main()
{
    int idade;
    idade = 18;

    cout << "Minha idade é: " << idade << endl;
    return 0;
}
```



basico

Arquivo Editar Ver Pesquisar Terminal Ajuda

Minha idade é: 18

## Tipo de dado caractere: **char**

Além de números inteiros, podemos também armazenar caracteres, ou seja, letras.

Para isso, usamos a palavra-chave *char* (de *character*, do inglês).

Vamos declarar uma variável do tipo *char*, de nome *letra*.

- `char letra;`

Agora vamos atribuir um valor para ela. No caso, como é um *char*, devemos armazenar alguma letra do alfabeto, como por exemplo, C:

- `letra = 'C';`

Note que os caracteres devem estar dentro de aspas simples.

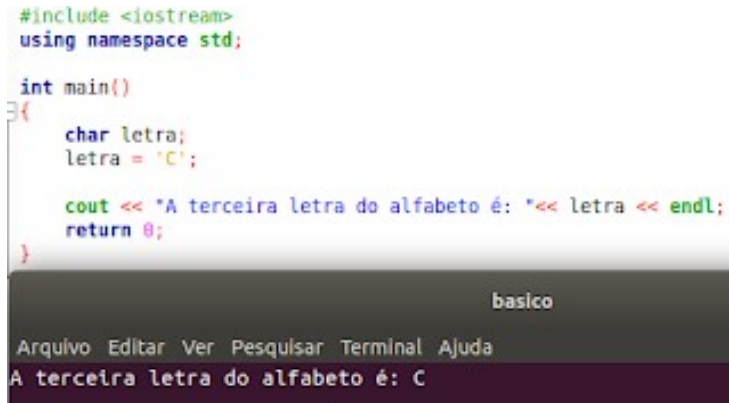
Vamos imprimir essa letra na tela:

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

int main()
{
    char letra;
    letra = 'C';
}
```

```
    cout << "A terceira letra do alfabeto é: " << letra << endl;  
    return 0;  
}
```

Resultado:



```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    char letra;  
    letra = 'C';  
  
    cout << "A terceira letra do alfabeto é: " << letra << endl;  
    return 0;  
}
```

basico

Arquivo Editar Ver Pesquisar Terminal Ajuda

A terceira letra do alfabeto é: C

Lembrando que 'a' é diferente de 'A', são dois caracteres distintos. Os caracteres são mais usados em conjuntos, ou seja, formando um texto. Vamos estudar isso em uma seção posterior de nosso curso, sobre *strings*.

Veja que:

- 5 - é um número, um dado do tipo *int*
- '5' - é um caractere, uma letra, não pode ser usado por exemplo em operações matemáticas

## Tipo de dado flutuante: **float** e **double**

Já aprendemos a lidar com números inteiros.

Porém, nem tudo na vida é um número inteiro, como sua idade.

Muitas vezes, precisamos trabalhar com valores fracionados, ou seja, 'quebrados'.

Para isso, usamos os tipos de dados *float* e *double*:

```
float preco;
```

```
double valor;
```

A diferença é que *float* tem precisão única, e *double* tem precisão dupla (ou seja, cabe uma parte fracionada maior, e um maior número de bytes da memória foi reservado para este tipo de variável).

Vejamos um uso:

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

int main()
{
    float preco;
    preco = 14.99;

    cout << "A apostila custa: R$ " << preco << endl;
    return 0;
}
```

Note que usamos ponto (.) ao invés de vírgula (,). É assim que usamos em programação, como nos Estados Unidos, por exemplo.



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

int main()
{
    float preco;
    preco = 14.99;

    cout << "A apostila custa: R$ " << preco << endl;
    return 0;
}
```

basico

Arquivo Editar Ver Pesquisar Terminal Ajuda

A apostila custa: R\$ 14.99

Assim, uma lojinha de R\$ 1,99 em nosso mundo, será lojinha de R\$ 1.99 no mundo da programação, ok?

## O tipo de dado Booleano: **bool**

Você já deve ter ouvido falar que na computação (ou na tecnologia, de um modo geral), é tudo 1 ou 0, não é?

De fato, os valores 1 e 0, são muito importantes, pois representam **verdadeiro** e **falso**, consecutivamente.



Existe um tipo de dado para armazenar somente informações do tipo *true/false* (chamados de booleanos), o *bool*.

Declarando:

- `bool verdade;`

Atribuindo um valor booleano:

- `verdade = true;`

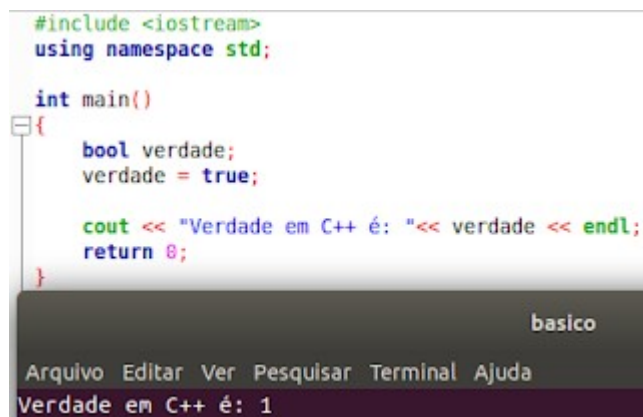
Exibindo o valor de *true* na tela:

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

int main()
{
    bool verdade;
    verdade = true;

    cout << "Verdade em C++ é: " << verdade << endl;
    return 0;
}
```

Resultado:



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

int main()
{
    bool verdade;
    verdade = true;

    cout << "Verdade em C++ é: " << verdade << endl;
    return 0;
}
```

basico

Arquivo Editar Ver Pesquisar Terminal Ajuda

Verdade em C++ é: 1

## Exercícios:

1. Faça um programa em C++ que exibe o valor de duas variáveis, do tipo booleano, mostrando cada valor atribuído possível. Use duas variáveis.
2. Refaça o exercício anterior, agora declarando apenas uma variável.

## Nomes de variáveis

Como temos total poder para escolhermos o nome que quisermos para nossas variáveis, temos também algumas responsabilidades.

A primeira delas é não escolher palavras-chaves (*keywords*), que serão listas no tópico a seguir.

Outra responsabilidade, é a da organização.

Você vai ficar tentado a usar:

```
int a, float b, char c
```

Evite esses nomes. Use:

```
int idade;
```

```
float diametro;
```

```
char letra;
```

Ou seja, nomes de variáveis que queiram significar algo relacionado ao valor que você vai armazenar ali. Isso ajuda muito quando seus programas forem se tornando maiores e mais complexos.

Evite também:

```
double salarioprogramador;
```

Use:

```
double salario_programador;
```

Ou ainda:

```
double salarioProgramador;
```

Note como assim fica mais fácil de ler e de cara já podemos antever que tipo de informação tem nessas variáveis, concorda?

## Palavras-chave reservadas

Existem algumas palavras que você não deve usar como nomes para suas variáveis, pois elas são reservadas para o funcionamento interno do C++.

São elas:

and, and\_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, class, compl, const, const\_cast, continue, default, delete, do, double, dynamic\_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, not, not\_eq, operator, or, or\_eq, private, protected, public, register, reinterpret\_cast, return, short, signed, sizeof, static, static\_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar\_t, while, xor\_eq

## Resposta do exercício

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

int main()
{
    bool valorBooleano;
    valorBooleano = true;
    cout << "Verdade em C++ é: " << valorBooleano << endl;

    valorBooleano = false;
    cout << "Falso em C++ é: " << valorBooleano << endl;
    return 0;
}
```

# A Função sizeof() em C++ e outros tipos de dados (short, long e unsigned)

Neste tutorial de nosso **curso de C++**, vamos dar continuidade ao estudo das variáveis, agora aprendendo um pouco mais sobre seus tipos, tamanho e precisão.

## Tamanho de dados: a função sizeof()

C++ é uma linguagem poderosa, e isso não é maneira de falar. Uma prova disso é a responsabilidade que o programador tem com cada byte da memória do computador, através da linguagem C++.

Sempre que declaramos uma variável estamos, diretamente, reservando bytes da memória de sua máquina, e dando acesso ao endereço daquele espaço reservado através do nome da variável.

Existe uma função (bloco de código, pronto para usar) que nos fornece o tamanho de cada tipo de dado que estudamos, é a função: **sizeof()**

Para saber o tamanho de um dado, basta colocá-lo entre parêntesis no comando: `sizeof( variavel )`

E ele 'retorna' o tanto de bytes ocupados.

O programa abaixo mostra os valores reservados para cada tipo de variável em meu computador:

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

int main()
{
    cout << "Variável 'int' ocupa   : " << sizeof(int) << " byte(s)" << endl;
    cout << "Variável 'char' ocupa  : " << sizeof(char) << " byte(s)" << endl;
    cout << "Variável 'float' ocupa : " << sizeof(float) << " byte(s)" << endl;
    cout << "Variável 'double' ocupa: " << sizeof(double) << " byte(s)" << endl;
    cout << "Variável 'bool' ocupa  : " << sizeof(bool) << " byte(s)" << endl;
    return 0;
}
```

}

Faça no seu e poste nos comentários, isso pode mudar de um sistema operacional para outro.

No meu notebook o resultado foi:

```
Variável 'int' ocupa   : 4 byte(s)
Variável 'char' ocupa  : 1 byte(s)
Variável 'float' ocupa : 4 byte(s)
Variável 'double' ocupa: 8 byte(s)
Variável 'bool' ocupa  : 1 byte(s)
```

## Tipo de dado inteiro: short, long e unsigned

Veja que um inteiro ocupa 4 bytes.

Seja pra armazenar o número 0, o 1, o 2112 ou o 2147483647, ele guarda 4 bytes da memória do seu computador. Mas, as vezes isso pode ser desperdício.

Vamos supor que você está trabalhando com um microcontrolador ou quer fazer o sistema de um relógio digital ou timer de microondas. Nesses casos, memória é um recurso RARO e LIMITADÍSSIMO, quanto menos desperdiçarmos, melhor.

A solução é em vez de declarar 'int variavel', fazer:

**short variavel;**

Pois vai utilizar menos espaço em memória.

E caso esteja fazendo um software pra NASA estudar planetas que estão a bilhões de trilhões de km de distância e precisa usar número gigantescos? Aí você usa o *long*:

**long variavel;**

Lembrando que números inteiros são de dois tipos: os negativos e os positivos.

Se você vai trabalhar com idade, por exemplo, não faz sentido trabalhar com números negativos, concorda?

Nesse caso, use variáveis do tipo *unsigned*:

**unsigned variavel;**

A extensão numérica de cada tipo é:

- short: de -32.768 até +32.767
- unsigned short: de 0 até +65,535
- int: de -2.147.483.648 até +2.147.483.647
- unsigned int: de 0 até 4.294.967.295
- long: de -2.147.483.648 até +2.147.483.647
- unsigned long: de 0 até 4.294.967.295

Vamos ver quanto de espaço ocupa cada um desses tipos?

O código é:

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

int main()
{
    cout << "Variável 'short' ocupa      : " << sizeof(short) << " bytes" <<
endl;
    cout << "Variável 'unsigned short' ocupa : " << sizeof(unsigned short) <<
" bytes" << endl;
    cout << "Variável 'unsigned int' ocupa   : " << sizeof(unsigned int) << "
bytes" << endl;
    cout << "Variável 'long' ocupa          : " << sizeof(long) << " bytes" <<
endl;
    cout << "Variável 'unsigned long' ocupa  : " << sizeof(unsigned long) << "
bytes" << endl;
    return 0;
}
```

E o resultado:

```
Variável 'short' ocupa      : 2 bytes  
Variável 'unsigned short' ocupa : 2 bytes  
Variável 'unsigned int' ocupa  : 4 bytes  
Variável 'long' ocupa        : 8 bytes  
Variável 'unsigned long' ocupa : 8 bytes
```

## Precisão de float e double

Já que estamos falando de variáveis inteiras, seu alcance, tamanho e precisão, vamos revisar novamente as variáveis float e double, usadas para representar números decimais:

- float: precisão única, ocupa 4 bytes e vai de  $3,4 \times 10^{(-38)}$  até  $3,4 \times 10^{38}$
- double e long double: precisão dupla, ocupam 8 bytes e vão de  $1,7 \times 10^{(-308)}$  até  $1,7 \times 10^{308}$

## Referências

[https://www.tutorialspoint.com/cplusplus/cpp\\_data\\_types.htm](https://www.tutorialspoint.com/cplusplus/cpp_data_types.htm)

# Matemática em C++: Operadores de soma (+), subtração (-), multiplicação (\*), divisão (/) e resto da divisão (%)

Sem exageros ou simplificações demasiadas, podemos dizer que um computador nada mais é que uma mega-super-hiper calculadora. Uma calculadora especial, bem incrementada e potente.

Fazer cálculos, contar, computar...é a base da computação, e consequentemente, da programação.

Neste tutorial, vamos aprender a fazer contas em C++.

## Soma em C++: operador +

Para realizarmos a operação de adição, usamos o operador + entre dois valores.

Pode ser na forma literal: 1 + 1

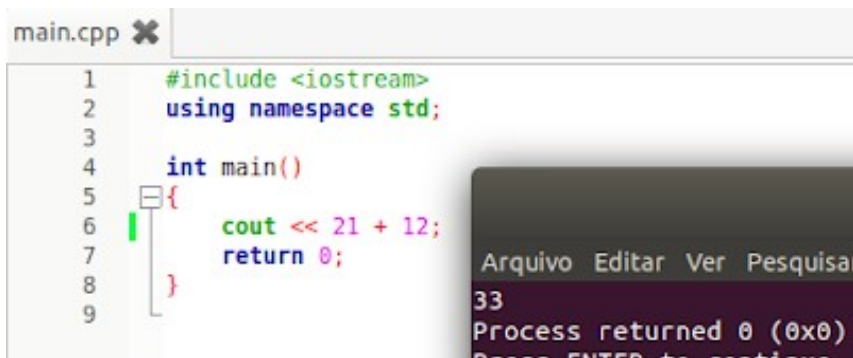
Ou valores armazenados em variáveis: valor1 + valor2

Um programa que mostra a soma dos números 21 e 12:

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

int main()
{
    cout << 21 + 12;
    return 0;
}
```

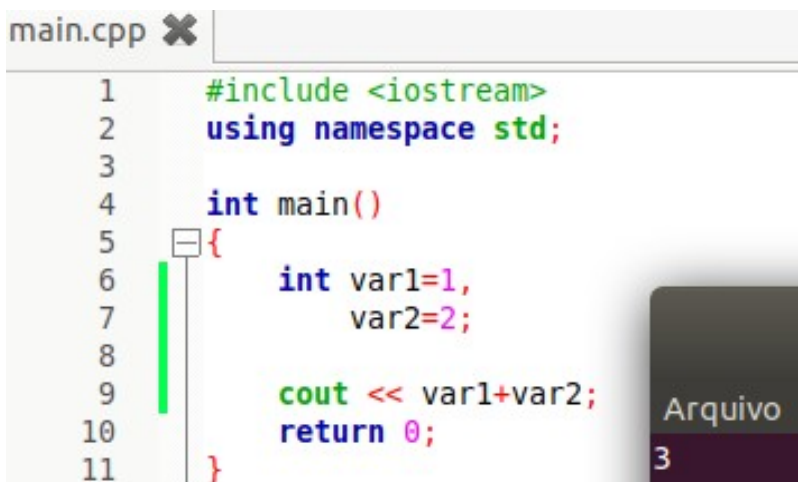




```
main.cpp X
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << 21 + 12;
7      return 0;
8  }
```

Arquivo Editar Ver Pesquisa  
33  
Process returned 0 (0x0)  
Press ENTER to continue

Esses valores poderiam estar previamente armazenado em variáveis, var1 e var2, por exemplo e somados depois, veja:



```
main.cpp X
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int var1=1,
7          var2=2;
8
9      cout << var1+var2;
10     return 0;
11 }
```

Arquivo  
3

Note como declaramos as variáveis var1 e var2. Como as duas são inteiras, não precisamos fazer:  
int var1=1;  
int var2=2;

Podemos 'resumir', escrever menos e fazer:  
int var1=1,  
 var2=2;

Bem mais chique, não?

## Subtração em C++: operador -

Assim como somamos, podemos subtrair, e para isto, basta usar o operador - (menos).

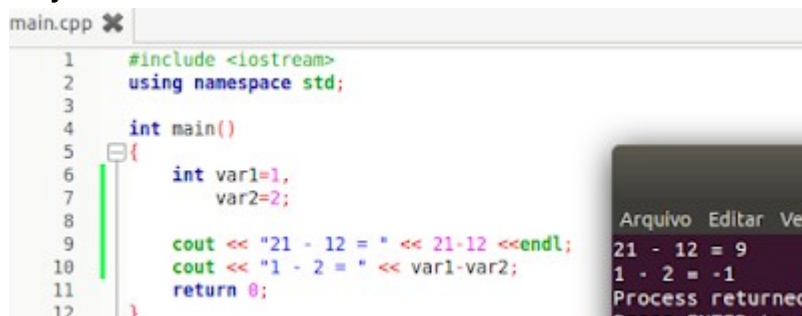
Vamos subtrair 21 de 12, e depois fazer 1 menos 2:

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

int main()
{
    int var1=1,
      var2=2;

    cout << "21 - 12 = " << 21-12 << endl;
    cout << "1 - 2 = " << var1-var2;
    return 0;
}
```

Veja o resultado:



The screenshot shows a code editor with the following C++ code:

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int var1=1,
7       var2=2;
8
9     cout << "21 - 12 = " << 21-12 << endl;
10    cout << "1 - 2 = " << var1-var2;
11    return 0;
12 }
```

To the right, a terminal window displays the output:

```
Arquivo  Editar  Ver
21 - 12 = 9
1 - 2 = -1
Process returned
Press ENTER to...
```

Note que usamos strings:

"21 - 12 = "

"1 - 2 = "

E cálculos matemáticos:

21 - 12

var1 - var2

São duas coisas diferentes, são dois [tipos de dados](#) diferentes.

Fizemos isso e colocamos tudo junto no [comando cout \(saída simples em C++\)](#), para deixarmos a saída bem bonitinha e organizada.

## Multiplicação em C++: operador \*

No nosso dia-a-dia, para descrever a operação de multiplicação, usamos o símbolo **x**, de vezes, não é verdade?

Porém, em programação C++, 'x' é uma letra.

Para identificar o operador de produto, usamos o asterisco: \*

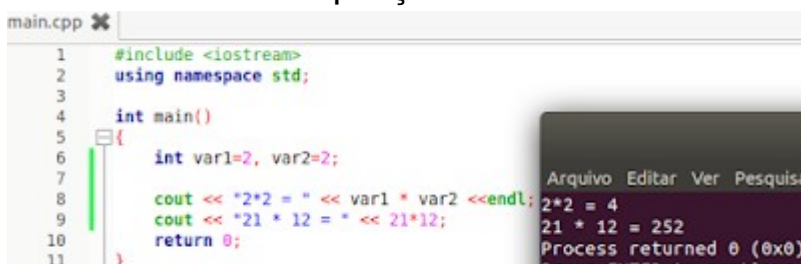
Assim, para multiplicarmos 2 por 2, ou 21 por 12, fazemos:

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

int main()
{
    int var1=2, var2=2;

    cout << "2*2 = " << var1 * var2 << endl;
    cout << "21 * 12 = " << 21*12;
    return 0;
}
```

O resultado da compilação:



```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int var1=2, var2=2;
7
8      cout << "2*2 = " << var1 * var2 << endl;
9      cout << "21 * 12 = " << 21*12;
10     return 0;
11 }
```

Arquivo Editar Ver Pesquisa  
2\*2 = 4  
21 \* 12 = 252  
Process returned 0 (0x0)

Note que declaramos as variáveis todas na mesma linha. Fizemos isso pois elas são do mesmo tipo, do tipo inteiro, e separamos por vírgula.

Poderíamos também ter feito:

```
int var1=2;
```

E depois: `var1 * var1`

Afinal, são de valores iguais, concorda?

## Divisão em C++: operador /

No dia-a-dia, o operador de divisão também é diferente, é o “÷”.

Já na programação, o símbolo é o : /


Vamos dividir 4 por 2, e depois 1 por 3.  
O código fica:

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

int main()
{
    float var1, var2;
    var1=1;
    var2=3;

    cout << "4 / 2 = " << 4/2 << endl;
    cout << "1 / 3 = " << var1/var2;
    return 0;
}
```

A compilação resulta em:

The image shows a screenshot of a C++ IDE. On the left, the source code for 'main.cpp' is displayed with line numbers 1 through 14. The code includes the iostream header, uses the std namespace, and defines a main function that declares two float variables, initializes them to 1 and 3, and prints the results of integer division (4/2) and floating-point division (1/3). On the right, a terminal window shows the output of the program: '4 / 2 = 2' and '1 / 3 = 0.333333', followed by a message indicating the process returned 0 and a prompt to press ENTER to continue.

Note que primeiro declaramos as variáveis:  
float var1, var2;

E só depois inicializamos com os valores 1 e 3:  
var1=1;  
var2=3;

É uma outra forma de declarar e inicializar variáveis.

No primeiro cout, dividimos 4 por 2. Dois números inteiros, com resultados da divisão inteiro, e o C++ mostrou o inteiro resultante: 2.

Mas ao dividir 1 por 3, o resultado é decimal. Por isso declaramos as variáveis `var1` e `var2` como **float**, pois sabíamos que precisaríamos das casas decimais para representar a operação de divisão.

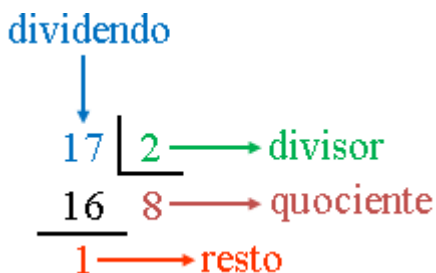
Altere a declaração de **float** para **double**, o que aconteceu?

Altere a declaração de **float** para **int**, o que aconteceu?

## Resto da divisão em C++: operador %

Por fim, vamos usar o operador aritmético `%`, chamado de módulo ou resto da divisão.

Vamos voltar pra escolinha e relembrar como fazíamos continhas:



Tá vendo aquele 'resto'? É o resto da divisão de 17 por 2.

Para obter esse resultado, faça:

- $17 \% 2 = 1$

Esse operador é especial pois só se usa com números inteiros, ok?

No futuro, em nosso **curso de C++**, vamos usar o operador de módulo (ou resto da divisão), para alguns algoritmos específicos, como achar números primos e trabalhar com múltiplos.

## Exercícios de Matemática em C++

**01.** Escreva um programa que some os número 10, 20 e 30

**02.** Escreva um programa que divida 21 por 12

**03.** Qual o módulo, ou resto da divisão, do número 21 por 12?

**04.** Qual o erro do código abaixo?

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

int main()
{
    numero = 2112;
    int numero;

    cout << "Numero: "<< numero << endl;
    return 0;
}
```

**05.** Armazene na variável 'soma' o resultado exercício 1. Em seguida, divida por 3 para achar a média, que valor encontrou?

**06.** Refaça o exercício anterior, mas sem usar nenhuma variável, ou seja, imprima o valor do resultado direto no *cout*.

**07.** Se fizer:  $10 + 20 + 30/3$ , qual o resultado? É a média? Se não for, por quê deu errado?

Em breve, ainda em nossa seção de [Introdução ao C++](#), vamos voltar a falar dos operadores, em relação a sua precedência.

Esses exercícios estão resolvidos e comentados em nossa [Apostila C++ Progressivo](#).

## Fontes de estudo

<http://www.cplusplus.com/doc/tutorial/operators/>

# Precedência de Operadores e Agrupamento de Expressões com Parêntesis

Agora que você aprendeu tudo no tutorial passado, sobre [operações matemáticas em C++](#), responda, de cabeça, pra gente, quanto valem as seguinte expressões:

- $1 + 6 * 3$
- $(1 + 6) * 3$
- $1 + 6 / 2$
- $(1 + 6 / 2)$

## Ordem dos operadores matemáticos

Vamos pegar a primeira e a terceira expressão.

Na primeira, temos duas soluções que vocês podem achar:

- $1 + 6 * 3 = 1 + 18 = 19$

Ou:

- $1 + 6 * 3 = 7 * 3 = 21$

No primeiro cálculo, fizemos primeiro a multiplicação e só depois a soma.

No segundo cálculo, fizemos primeiro a soma do 1 com o 6, só depois multiplicamos por 3.

Qual o certo?

Já a terceira expressão, pode ser resolvida assim:

- $1 + 6 / 2 = 1 + 3 = 4$

Alguns podem calcular assim:

- $1 + 6 / 2 = 7 / 2 = 3.5$

No primeiro caso, fizemos primeiro a operação de divisão. No segundo caso, primeiro somamos 1 com 6, pra só depois fazer a divisão.

Vamos colocar ambas expressões num programa, pro C++ nos responder qual o correto.

Mas, antes, pense aí qual você acha que é o certo.

Depois, faça um programa para calcular as duas expressões e veja o resultado.

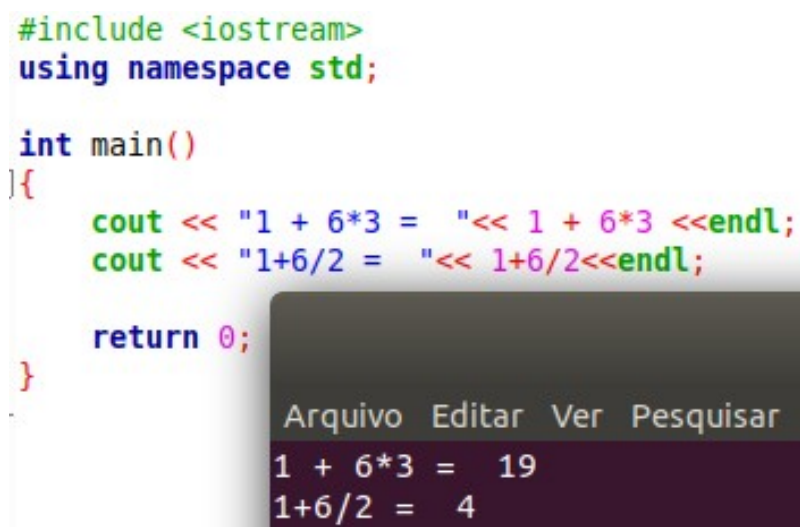
Nosso código fica assim:

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

int main()
{
    cout << "1 + 6*3 = " << 1 + 6*3 << endl;
    cout << "1+6/2 = " << 1+6/2 << endl;

    return 0;
}
```

E o resultado:



The image shows a screenshot of a C++ program being executed. The code is displayed in a text editor with syntax highlighting. Below the code, a terminal window shows the output of the program. The output consists of two lines: "1 + 6\*3 = 19" and "1+6/2 = 4". The terminal window has a menu bar with "Arquivo", "Editar", "Ver", and "Pesquisar".

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

int main()
{
    cout << "1 + 6*3 = " << 1 + 6*3 << endl;
    cout << "1+6/2 = " << 1+6/2 << endl;

    return 0;
}
```

Arquivo Editar Ver Pesquisar

1 + 6\*3 = 19  
1+6/2 = 4

## Precedência de Operadores Matemáticos no C++

Imagina se a NASA faz a simulação de um lançamento e um cálculo resulta no valor 21.

Mas aí, durante o lançamento oficial, outra máquina faz o mesmo cálculo e resulta no valor 12 ?

Não dá né?



Por isso, o C++ estabeleceu uma ordem para fazer os cálculos, uma precedência entre os operadores.  
Como pudemos ver, foi realizado primeiro o cálculo da multiplicação e divisão, antes da adição.

Ordem dos operadores é a seguinte:

1. \* / %
2. + -

Ou seja, o C++ viu uma expressão matemática, a primeira coisa que vai ver é se tem uma multiplicação divisão ou operador de módulo. Se tiver um, resolve ele primeiro.  
Se tiver mais de um desses, resolve da esquerda pra direita.

Só depois, vai checar se tem alguma adição ou subtração.  
Se tiver mais de uma? Vai resolvendo da esquerda pra direita também, ok?

## Parêntesis () - Organização e Precedência

Se colocarmos uma operação entre parêntesis, ela vai sempre ser realizada primeiro.  
É como se os ( ) tivesse uma precedência superior aos dos operadores matemáticos.

Por exemplo, na conta:  $1 + 6 * 3$

Se quisermos que a soma seja feita primeiro, fazemos:  $(1+6)*3$

Se quisermos garantir que a multiplicação seja feita primeiro:  $1 + (6*3)$

Vamos supor que queiramos calcular a média aritmética de 5 números, em Matemática isso é dado por:

$$m = \frac{a + b + c + d + e}{5}$$

Uma pessoa que não aprendeu bem precedência de operadores, pode fazer:  
 $m = a+b+c+d+e/5$  ;

O erro é que somente a variável **e** vai ser dividida por 5, consegue perceber isso?

O correto é usar parêntesis:

$$m = (a+b+c+d+e)/5 ;$$

Assim, além de ficar certa a conta, fica mais organizada também.

## Exercícios de C++

1. A equação da uma reta é dada por:  $y=mx + c$

Escreva essa equação em C++ usando os operadores corretamente.

2. Escreva as equação algébricas abaixo na forma correta em C++:

$$y = 3\frac{x}{2}$$

$$z = 3bc + 4$$

$$a = \frac{3x+2}{4a-1}$$

3. Uma equação do segundo grau é dada por:  $ax^2 + bx + c = 0$

Escreva ela na linguagem C++.

4. Ainda na equação anterior, como calcularíamos o delta?

5. E as raízes da equação do segundo grau ?

## Respostas:

1.  $y = m*x + c;$

2.

$$y = 3*x/2;$$

$$y = 3*b*c + 4;$$

$$a = (3*x + 2) / (4*a - 1)$$

3.  $a*x*x + b*x + c = 0;$

4.  $\text{delta} = b*b - 4 * a * c;$

5.

$$\text{raiz1} = (-b + \text{delta})/(2*a);$$

$raiz2 = (-b - \text{delta}) / (2 * a);$

(Note que o – de -b não é bem uma operação de subtração, se refere ao sinal da variável)

### **Referências de estudo**

Lista completa de precedência de outros operadores também:

[https://en.cppreference.com/w/cpp/language/operator\\_precedence](https://en.cppreference.com/w/cpp/language/operator_precedence)

# Recebendo Dados do Teclado - O Objeto cin do C++

Neste tutorial de nosso **curso de C++**, vamos aprender como receber informações do usuário, pelo teclado, no terminal de comando, utilizando o comando **cin <<**

## Trocando Informações

Para você acessar um site, precisa digitar o endereço da URL e dar enter no navegador.

Para abrir uma foto, você precisa procurar o arquivo no meio das pastas e clicar nela.

Para escutar uma música ou vídeo, também precisa fazer o mesmo.

Quer digitar um texto? Tem que abrir o editor de texto e ir digitando, letra por letra, e apertando os botões, menus e opções do programa, para formatar e deixar tudo bonitinho.

Ou seja, você precisa passar informações para o computador (digitar algo, clicar aqui ou ali, inserir uma URL etc). E baseado no que você faz, ele vai fazer algo.

Se você clica em um botão, ele faz uma coisa.

Se clicar em outro botão, ele faz outra coisa totalmente diferente.

Ele **reage** de maneira diferente dependendo da informação que o usuário passa.

É isso que vamos começar a aprender a fazer em C++, e o primeiro passo é aprender como **receber** informações do usuário pelo teclado. Vamos lá?

## Como Receber Informações do Usuário: **cin >>**

Até o momento, todas as informações necessárias para nossos programas funcionarem eram dadas no código, como inicialização de variáveis com seus valores.

Se quiséssemos fazer o programa rodar com outros valores, tínhamos que mudar eles no código, compilar e rodar tudo de novo. Imagina ter que mexer no código de cada programa que você utiliza?

Agora, vamos aprender como fazer o usuário inserir informações e o programa trabalhar em cima desses dados, seja lá quais forem.

Para fazer isso, usamos o objeto **cin** (vamos aprender na seção de orientação a objetos, o que é um objeto, não se preocupe), que é um objeto de fluxo de entrada (input stream object) seguido do símbolo **>>** (operador de extração de fluxo - stream extraction operator) e uma variável:

- `cin >> nome_variavel;`

O que o C++ vai fazer quando chegar nesse código é simplesmente parar e esperar algo ser digitado no teclado, e quando você apertar ENTER, o valor que você escreveu vai ser armazenado na variável `nome_variavel`.

## Recebendo Números do Usuário com **cin >>**

Bom, vamos lá.

O programa abaixo pede um inteiro ao usuário, o valor de sua idade, armazena na variável **age** e em seguida exibe o valor digitado:

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

int main()
{
    int age;

    cout << "Digite sua idade: ";
    cin >> age;
    cout << "Sua idade é: " << age;

    return 0;
}
```

O resultado é:

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

int main()
{
    int age;

    cout << "Digite sua idade: ";
    cin >> age;
    cout << "Sua idade é: " << age;

    return 0;
}
```



Já o programa abaixo pede dois números ao usuário. Armazena nas variáveis *num1* e *num2*.

Soma esses valores e armazena na variável *sum*, então exibe o valor da soma:

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

int main()
{
    float num1, num2, sum;

    cout << "Primeiro numero: ";
    cin >> num1;
    cout << "Segundo numero : ";
    cin >> num2;

    sum = num1 + num2;

    cout << "A soma é: " << sum;

    return 0;
}
```

Veja o resultado:

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

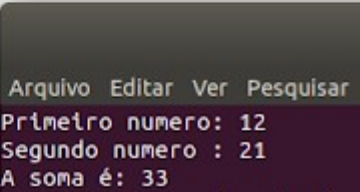
int main()
{
    float num1, num2, sum;

    cout << "Primeiro numero: ";
    cin >> num1;
    cout << "Segundo numero : ";
    cin >> num2;

    sum = num1 + num2;

    cout << "A soma é: " << sum;

    return 0;
}
```



Podemos também receber múltiplos valores do usuário, basta colocar um ou mais operadores >>, um junto do outro. Vamos refazer o exemplo anterior. Digite um número, dê um espaço e digite outro número:

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

int main()
{
    float num1, num2, sum;

    cout << "Digite dois numeros ";
    cin >> num1 >> num2;
    sum = num1+num2;

    cout << "A soma é: " << sum;

    return 0;
}
```

## Importante: vírgula é ponto

No sistema numérico brasileiro, usamos a vírgula para valores 'quebrados'. Por exemplo: R\$ 1,99

Porém, na programação usamos o PONTO .  
Em computação: R\$ 1.99

Teste os exercícios anteriores digitando números com vírgula e depois ponto

## Recebendo textos (strings) do usuário

Uma string (texto, em programação) nada mais é que um grupo de caracteres.

Por isso, para lermos uma string que o usuário digitar, precisamos declarar um grupo de caracteres.

Por exemplo, para armazenar um espaço de 50 caracteres, declaramos:  
`char nome[50];`

Na nossa seção de Strings estudaremos melhor porque a declaração é assim.

O programa abaixo pergunta seu nome e te saúda:

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

int main()
{
    char name[50];

    cout << "Digite seu nome : ";
    cin >> name;

    cout << "Olá, " << name << "! Tudo bem?";

    return 0;
}
```

O resultado é:



The image shows a code editor with the C++ program from the previous block. Overlaid on the bottom right is a terminal window showing the program's execution. The terminal has a menu bar with 'Arquivo', 'Editar', 'Ver', and 'Pesquisar'. The output shows the prompt 'Digite seu nome :', the user input 'Neill', and the resulting greeting 'Olá, Neill! Tudo bem?'.

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

int main()
{
    char name[50];

    cout << "Digite seu nome : ";
    cin >> name;

    cout << "Olá, " << name << "! Tudo bem?";

    return 0;
}
```

Arquivo Editar Ver Pesquisar  
Digite seu nome : Neill  
Olá, Neill! Tudo bem?



Agora teste um nome completo, que tenha espaço em branco.  
O que aconteceu? Entenderemos melhor isso na seção de strings de nosso curso.

## Recebendo dados Booleanos em C++

Por fim, podemos também receber valores booleanos.

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

int main()
{
    bool val;

    cout << "Digite algo : ";
    cin >> val;

    cout << "Isso representa o booleano: " << val;

    return 0;
}
```

Teste inserindo o valor 0. O que apareceu?

Teste inserindo qualquer outro valor inteiro. O que aconteceu?

Digite um caractere. O que retorna?

## Exercícios usando **cin>>**

1. Rode o código abaixo.

Ele deu funcionou corretamente? Qual o problema dele ?

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

int main()
{
    float num1, num2, sum;
    sum = num1 + num2;
```

```

cout << "Primeiro numero: ";
cin >> num1;
cout << "Segundo numero : ";
cin >> num2;

cout << "A soma é: " << sum;

return 0;
}

```

2. Escreva um programa que peça um número e exiba o dobro dele.
3. Faça um programa que recebe o lado de um quadrado e retorna o valor de sua área.
4. Crie um programa em C++ que receba os dois lados de um retângulo e calcule sua área (use apenas um comando **cin**)
5. Faça um programa que recebe seu nome completo (incluindo espaços) e imprima ele na tela. Estude e aprenda como fazer isso usando o link do site de referência.

## Resposta dos exercícios

1. Ele faz a soma antes de receber os números do usuário.  
O correto é primeiro receber os números, só depois somar, isso é bem óbvio.

O resultado maluco que aparece é porque nas variáveis, ao declarar, elas estão em determinados locais da memória, com algum valor aleatório qualquer, que chamamos de lixo.

2.

```

#include <iostream>
using namespace std;

```

```

int main()
{

```

```

float numero;
cout << "Digite um número: ";
cin >> numero;

cout << "O dobro de " << numero << " é " << 2*numero;

return 0;
}

```

3.

```

#include <iostream>
using namespace std;

int main()
{
    float lado;
    cout << "Lado do quadrado: ";
    cin >> lado;

    cout << "Área do quadrado: " << lado*lado;

    return 0;
}

```

4.

```

#include <iostream>
using namespace std;

int main()
{
    float lado1, lado2;
    cout << "Valor dos lados (separados por espaço): " << endl;
    cin >> lado1 >> lado2;

    cout << "Área do retângulo: " << lado1*lado2;

    return 0;
}

```

5.

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

int main()
{
    string name;

    cout << "Nome completo: ";
    getline(cin, name);

    cout << "Olá, " << name << "! Tudo bem?";
    return 0;
}
```

## Referência de estudo

[http://www.cplusplus.com/doc/tutorial/basic\\_io/](http://www.cplusplus.com/doc/tutorial/basic_io/)

## Como comentar códigos em C++: // e /\* \*/

Neste tutorial, vamos aprender uma prática de extrema importância no mundo da programação: como comentar códigos.

### Comentários: O que são? Para que servem? Por que usar?

Uma das partes mais importantes, de um código de programação, por incrível que pareça, é algo que ignorado pelo compilador: os comentários.

Usamos os comentários para uma comunicação humana, para que você programador diga algo para outro programador que vai ler seu código (pode até pra você mesmo, no futuro).

Eles são muito usados para descrever, em nossa linguagem humana, o que as linhas de códigos seguintes vão fazer, como funcionam, para que servem e dar o maior número detalhes possíveis.

Pode parecer bobagem no início, e de fato é, pois no começo nossos códigos são bem pequenos.

Mas a medida que vão ficar maiores e mais complexos, se torna uma tarefa árdua tentar entender o que um código faz apenas olhando, seria muito mais fácil se ele visse com explicações, e é aí que entram os comentários.

Imagine que você programou um game em C++, dezenas de milhares de linhas, lançou, ficou rico e tudo mais. Porém, 5 meses depois, descobriram um bug.

Amigo, pode ter certeza: ao rever o código, você vai ter extrema dificuldade de saber o que cada trecho faz e qual sua função, a gente esquece mesmo e dá um trabalho absurdo tentar entender um código só vendo aquela sopa de letrinhas.

Comente seus códigos, explique o que são, para que servem, o que determinada lógica ou algoritmo está fazendo e dê o máximo de informações possíveis. Isso é um **excelente** hábito de programação: comentar códigos.

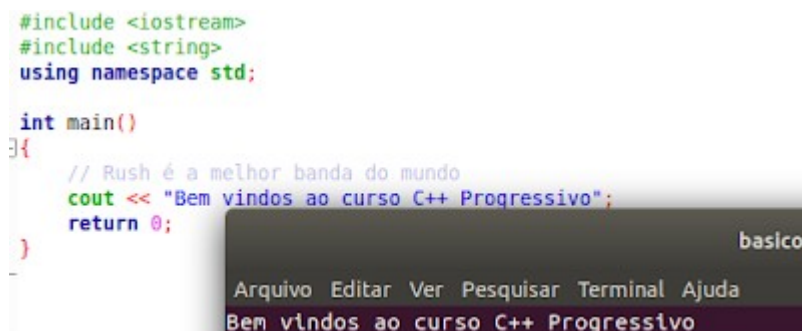
## Como comentar uma linha em C++: //

Para comentar uma linha (ou seja, fazer com que uma linha do código seja um comentário), basta iniciar ela com duas barras:

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

int main()
{
    // Rush é a melhor banda do mundo
    cout << "Bem vindos ao curso C++ Progressivo";
    return 0;
}
```

Note que o texto "Rush é a melhor banda do mundo", não aparece na tela do usuário:



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

int main()
{
    // Rush é a melhor banda do mundo
    cout << "Bem vindos ao curso C++ Progressivo";
    return 0;
}
```

basico

Arquivo Editar Ver Pesquisar Terminal Ajuda

Bem vindos ao curso C++ Progressivo

Vamos um exemplo realmente útil de código. O programa abaixo calcula a área de um círculo:

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

int main()
{
    float raio, area;

    // Pedindo o raio
    cout << "Valor do raio: ";
    cin >> raio;
```

```
// Calculando a área
area = 3.14 * raio * raio;

cout << "Area: " << area;
return 0;
}
```

Lembre-se: pode parecer bobo agora, mas é porque são exemplos simples. A medida que nossos algoritmos forem ficando maiores e mais complexos, você vai notar a importância de comentar seus códigos.

É uma característica clara e presente nos melhores programadores, e como você estudou pelo C++ Progressivo, certamente faz parte desse seleto grupo. Então, COMENTE SEUS CÓDIGOS.

## Comocomentar várias linhas: **/\* \*/**

Muitas vezes, é interessantes fazermos uso de várias linhas para comentar corretamente um código.

Isso é bem comum no início dos arquivos, dando uma breve descrição do programa, suas funções, falando do autor etc.

Isso ficaria muito trabalhoso usando as barras **//** em toda linha.

Para comentar várias linhas de uma vez só, comece com: **/\***

Escreva o que quiser.

Termine com: **\*/**

Prontinho, tudo que estiver entre esses dois símbolos, é considerado um comentário.

Vamos fazer um programa em C++ que pede o raio, e mostra o valor do perímetro e da área de um círculo.

```
/*
Programa: circulo.cpp
Autor: C++ Progressivo
Objetivo: esse programa recebe o valor do raio,
          do usuário, e retorna o valor do perímetro
          e da área do círculo
*/
```

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

int main()
{
    float raio, area, perim;

    cout << "Raio: ";
    cin >> raio;

    area = 3.14 * raio * raio;
    perim = 2 * 3.14 * raio;

    cout << "Perímetro: " << perim << endl;
    cout << "Area: " << area;
    return 0;
}
```

Bem bonito e organizado, não?



## Exercícios Básicos de C++

Parabéns, você finalizou a parte básica de nosso **Curso de C++**.

Agora vamos praticar.

Tente, de todo o coração, com toda calma, esforço e concentração, fazer os exercícios abaixo.

Deixe um comentário com suas soluções, ok?

### Questões de C++

01. Escreva um programa que pede o raio de um círculo, e em seguida exiba o perímetro e área do círculo.
02. Faça um programa que recebe o raio de uma esfera e calcula seu volume.
03. Faça um programa que recebe um inteiro, representado um valor em anos. Mostre quantos dias tem esse intervalo de tempo, assumindo que um ano tenha 365 dias.
04. Faça um programa que receba duas variáveis: o valor das horas e do minutos. Em seguida, converta tudo para minutos e também para segundos.
05. Crie um programa que receba a temperatura em graus Celsius e converta para Fahrenheit.
06. Faça o contrário do exercícios anterior.
07. Faça um programa que pergunte o ano atual e sua idade, em seguida exibe seu ano de nascimento.

### Exercícios de Porcentagem em C++

08. Faça um programa que calcula 12% de 2112

09. Faça um programa que recebe um valor do usuário e calcula 12% desse total

10. Faça um programa que recebe um valor de porcentagem do usuário, e calcula quanto isso representa de um segundo valor que ele digitou.

11. Programe um software que recebe dois números, onde o primeiro deve ser menor que o segundo.

Em seguida, ele calcula a porcentagem que o primeiro representa do segundo.

Por exemplo, se digitou 12 e 21, isso quer dizer que 12 representa 57,14% de 21

12. Você se tornou programador C++, e agora está ganhando super bem. Mas, mesmo assim, vai ter que pagar impostos.

Crie um software que recebe o valor do seu salário e calcula os 7% do imposto de renda.

A saída do seu programa deve ser o salário bruto (sem abatimento), o tanto de imposto que vai pagar e o seu salário líquido (após descontar o IR).

13. Devido a inflação, todo ano seu salário deve ser ajustado baseado nela. Por exemplo, se a inflação foi 2,5% então seu salário deve crescer nesse mesmo montante, para não perder valor.

Crie um programa em C++ que pergunta o salário da pessoa, a inflação do último ano e aplique essa inflação. Mostre o salário anterior, o aumento devido a inflação e o novo salário.

14. Na cidade de C++lândia há uma tolerância de 15% do limite de velocidade, para não se levar uma multa. Faça um programa que pede ao usuário a velocidade máxima de uma via e calcula até que velocidade o carro pode transitar sem ser multado. Seu código vai ser embarcado no sistema de GPS do carro, para avisar o limite de velocidade que o carro deve percorrer.

## Questões de Média em C++

15. Crie um programa que peça duas notas ao usuário, e retorne a média dele.

16. Faça o mesmo do exercício anterior, mas para 3 notas.

17. A prova do vestibular do IME tem peso 3 para Matemática, 2.5 para Física, 2.5 para Química, 1.0 para Português e também 1.0 para Inglês. Crie um sistema que peça as notas do usuário e retorne a média dele.

## Soluções

01.

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

int main()
{
    float raio;
    cout << "Raio: ";
    cin >> raio;

    cout << "Perimetro= " << 2*3.14*raio << endl;
    cout << "Área = " << 3.14*raio*raio ;

    return 0;
}
```

02.

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

int main()
{
    float raio;
    cout << "Raio: ";
    cin >> raio;

    cout << "Volume: " << 4 * 3.14 * raio*raio*raio/3;
```

```
    return 0;
}
```

03.

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

int main()
{
    float anos;
    cout << "Anos: ";
    cin >> anos;

    cout << 365*anos << " dias ";

    return 0;
}
```

04.

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

int main()
{
    int h, m;
    cout << "Horas: ";
    cin >> h;

    cout << "Minutos: ";
    cin >> m;

    cout << "Minutos :" << 60*h + m << endl;
    cout << "Segundos :" << 60*60*h + m*60 << endl;

    return 0;
}
```

05.

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

```

int main()
{
    float C, F;

    cout << "Graus em Celsius: ";
    cin >> C;

    F = (9*C/5) + 32;

    cout << "Representa em Fahrenheit: " << F;

    return 0;
}

```

06.

```

#include <iostream>
using namespace std;

int main()
{
    float C, F;

    cout << "Graus em Fahrenheit: ";
    cin >> F;

    C = 5*(F-32)/9;

    cout << "Representa em Celsius: " << C;

    return 0;
}

```

07.

```

#include <iostream>
using namespace std;

int main()
{
    int ano, idade;

    cout << "Que ano estamos: ";

```

```

cin >> ano;

cout << "Qual sua idade: ";
cin >> idade;

cout << "Você nasceu no ano de " << ano-idade;

return 0;
}

```

## 08.

O próprio nome nos dá uma pista: porcentagem...por centagem ... por cem  
Ou seja, devemos dividir por 100.

12 dividido por 100 fica 0.12, então 12% de 2100 é:  
 $0.12 * 2112$

```

#include <iostream>
using namespace std;

int main()
{
    cout << "12% de 2112= " << 0.12*2112;
    return 0;
}

```

## 09.

Agora, ao invés de multiplicar por 2112, vamos multiplicar por um valor que o usuário inseriu e armazenamos na variável **num**:

```

#include <iostream>
using namespace std;

int main()
{
    float num;
    cout << "Digite um valor: ";
    cin >> num;
    cout << "12% de " << num << " = "
        << 0.12 * num;
}

```

```
    return 0;
}
```

10.

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

int main()
{
    float percentage, num;
    cout << "Valor da porcentagem: ";
    cin >> percentage;
    percentage = percentage/100;

    cout << "Segundo valor: ";
    cin >> num;

    cout << percentage << "% de "
         << num << " = " << percentage*num;
    return 0;
}
```

Note o usuário vai digitar um valor, como 12 para calcular 12%  
Mas no cálculo, não usamos 12 e sim 0.12 (esse valor por um cento, ou seja, esse valor dividido por 100).

Então, vemos essa novidade:  
 $percentage = percentage / 100;$

A priori, parece algo estranho, mas estamos apenas dividindo a variável *percentage* por 100.

Isso quer dizer: o valor novo de *percentage* é igual ao valor antigo dividido por 100.

Você poderia ter feito isso declarando outra variável *aux*:  
 $aux = percentage / 100;$   
 $percentage = aux;$

Mas aí ia alocar memória à toa, melhor fazer o que fizemos no código, é mais simples, rápido e eficiente (sim, programadores são preguiçosos, querem sempre escrever a menor quantidade de código possível).

11.

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

int main()
{
    float first, second;
    cout << "Primeiro numero: ";
    cin >> first;

    cout << "Segundo numero: ";
    cin >> second;

    cout << first << "% de "
        << second << " = " << (first/second)*100.0;
    return 0;
}
```

Aqui, a única coisa de diferente é lembrar de multiplicar por 100, para exibir o resultado em porcentagem.

12.

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

int main()
{
    float salary;
    cout << "Qual seu salário: ";
    cin >> salary;

    cout << "Salário bruto: $ " << salary << endl;
    cout << "7% de imposto: $ " << 0.07*salary << endl;
    cout << "Salário líquido: $ " << 0.93*salary << endl;

    return 0;
}
```



}

Se é descontado 7%, significa que você recebe 93% do salário bruto, que é o líquido.

### 13.

O aumento do salário corresponde ao valor anterior multiplicado por 0.025 (2,5%).

Já o novo salário é o anterior mais esses 2,5%:  $1 + 0.025 = 1.025 * (\text{salário anterior})$ .

Veja:

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

```
int main()
{
    float salary;
    cout << "Qual seu salário: ";
    cin >> salary;

    cout << "Salário anterior: $ " << salary << endl;
    cout << "Aumento de 2.5%: $ " << 0.025*salary << endl;
    cout << "Novo salário: $ " << 1.025*salary << endl;

    return 0;
}
```

### 14.

Se a velocidade máxima da via é *speed*, então a velocidade máxima com a tolerância pra não levar multa é:  $100\% + 15\% = 115\% = 1.15 * speed$

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

```
int main()
{
    float speed;
    cout << "Velocidade máxima: ";
    cin >> speed;
```

```
cout << "Velocidade máxima: " << speed << endl;  
cout << "Tolerância de 15%: " << 0.15*speed << endl;  
cout << "Velocidade máxima sem multa " << 1.15*speed << endl;
```

```
    return 0;  
}
```

15, 16 e 17.

## Média Aritmética Simples em C++

A média mais básica de todas é a chamada aritmética, a simples, onde você basicamente soma todos os termos e divide pelo total de termos.

Para calcular a média de dois números:

$(a+b)/2$

De três números:

$(a+b+c)/3$

De quatro números:

$(a+b+c+d)/4$

De n números:

$(a+b+c...)/n$

### Exercícios de Média aritmética Simples

- "Crie um programa que peça duas notas ao usuário, e retorne a média dele."

Nosso código fica:

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    float nota1, nota2, media;
```

```

cout << "Nota 1: ";
cin >> nota1;

cout << "Nota 2: ";
cin >> nota2;

media = (nota1+nota2)/2;

cout << "Média: " << media;

return 0;
}

```

Note que, embora seja uma fórmula bem simples, é de bom praxe colocar a soma dentro de parêntesis, para evitar cometer erros do tipo:

$a + b/2$

(nesse caso, estaríamos somando  $a$  com  $b/2$ )

- "Faça o mesmo do exercício anterior, mas para 3 notas."

Nosso código fica:

```

#include <iostream>
using namespace std;

int main()
{
    float nota1, nota2,
          nota3, media;

    cout << "Nota 1: ";
    cin >> nota1;

    cout << "Nota 2: ";
    cin >> nota2;

    cout << "Nota 3: ";
    cin >> nota3;

    media = (nota1+nota2+nota3)/3;

    cout << "Média: " << media;
}

```

```
    return 0;
}
```

## Média Ponderada em C++

Na média aritmética, todos os termos tem o mesmo 'peso', ou seja, contribuem igualmente para o valor final da média.

Já na ponderada, cada termo tem um peso, veja a fórmula:

$$\frac{p_1 \cdot x_1 + p_2 \cdot x_2 + p_3 \cdot x_3 + \dots + p_n \cdot x_n}{p_1 + p_2 + p_3 + \dots + p_n}$$

Os termos são x1, x2, x3, ...

Os respectivos pesos são p1, p2, p3, ...

Neste caso, somando todos termos multiplicados cada um por seu peso, e dividimos pela soma dos pesos.

- "A prova do vestibular do IME tem peso 3 para Matemática, 2.5 para Física, 2.5 para Química, 1.0 para Português e também 1.0 para Inglês. Crie um sistema que peça as notas do usuário e retorne a média dele."

Veja como fica o código:

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

int main()
{
    float math, phy, chem,
          port, eng, media;

    cout << "Nota de Matemática: ";
    cin >> math;

    cout << "Nota de Física: ";
    cin >> phy;
```

```

cout << "Nota de Química: ";
cin >> chem;

cout << "Nota de Português: ";
cin >> port;

cout << "Nota de English: ";
cin >> eng;

media = (3*math + 2.5*phy +
         2.5*chem+port+eng) / 10;

cout << "Média: " << media;

return 0;
}

```

Note que a soma dos pesos é 10.

## Exercícios de Média em C++

**01.** Resolva os exercícios anteriores, agora sem usar a variável a *media*.

**02.** Faça um programa que recebe a quantidade de litros que uma pessoa abasteceu no carro e a quantidade de km que ela percorreu com aquele combustível, em seguida calcule a média (ou seja, quantos km/l ele faz)

**03.** Faça um programa que peça o tamanho de um arquivo para download (em MB) e a velocidade de um link de Internet (em Mbps), calcule e informe o tempo aproximado de download do arquivo usando este link (em minutos)

**04.** Um novo modelo de carro, super econômico foi lançado.  
Ele faz 20 km com 1 litro de combustível.  
Cada litro de combustível custa R\$ 5,00.

Faça um programa que pergunte ao usuário quanto de dinheiro ele tem e em seguida diga quantos litros de combustível ele pode comprar e quantos kilometros o carro consegue andar com este tanto de combustível.

Seu script será usado no computador de bordo do carro.

# Calculadora Simples em C++: Como Programar

Para finalizar com chave de ouro nossa seção de Introdução ao C++, vamos fazer um programinha bem simples, mas bastante útil e interessante: uma calculadora.

Antes de ver o código, tente resolver.

Ela deve receber dois números do usuário, e exibir as operações de:

- Soma
- Subtração
- Multiplicação
- Divisão do primeiro pelo segundo
- Resto da divisão do primeiro pelo segundo
- Porcentagem do primeiro em relação ao segundo
- Média aritmética

Seu programa deve ficar + - assim, veja:

```
Primeiro numero: 12
Segundo numero: 21
Soma      : 33
Subtracao : -9
Multiplicacao: 252
Divisao   : 0.571429
Modulo    : 12
Porcentagem : 57.1429
Media     : 16.5

Process returned 0 (0x0)   execution time : 2.090 s
Press ENTER to continue.
```

Tente aí!

## Como fazer uma calculadora em C++

Veja como ficou nosso código:

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

int main()
{
```

```
float num1, num2;
```

```
//Recebendo os dados
```

```
cout << "Primeiro numero: ";
```

```
cin >> num1;
```

```
cout << "Segundo numero: ";
```

```
cin >> num2;
```

```
//Exibindo as operações
```

```
cout << "Soma      : " << num1 + num2 << endl;
```

```
cout << "Subtracao  : " << num1 - num2 << endl;
```

```
cout << "Multiplicacao: " << num1 * num2 << endl;
```

```
cout << "Divisao     : " << num1 / num2 << endl;
```

```
cout << "Modulo      : " << (int)num1 % (int)num2 << endl;
```

```
cout << "Porcentagem  : " << 100.0*(num1/num2) << endl;
```

```
cout << "Media       : " << (num1 + num2)/2 << endl;
```

```
return 0;
```

```
}
```

O seu, como ficou? Diferente?

Veja que a única coisa diferente foi o trecho:

```
(int) num1 % (int)num2
```

Isso se chama *casting*, ou seja, como o operador de módulo (resto da divisão) só tem sentido com valores inteiro e nossas variáveis são do tipo *float*, nós colocamos (int) antes das variáveis para dizer ao C++ que queremos tratar aquelas variáveis, naquele momento, como inteiros.

# Teste Condicionai em C++: IF e ELSE

Se você concluiu seus estudos em [Introdução ao C++](#), nossos mais sinceros parabéns.

Você já é um programador e o mais difícil ficou atrás.

Agora é hora de deixar nossos programas melhores, maiores, mais complexos, fazendo coisas cada vez mais incríveis. E vamos começar isso tudo aprendendo como fazer testes.

Sim, simplesmente isso: testar as coisas.

A computação é toda em cima de teste.

Seu celular fica testando se você está mexendo nele, senão estiver, ele desliga a tela.

Ao inserir nome de usuário e senha, o sistema faz um teste para saber se você digitou as informações corretas.

Ao fazer um saque no caixa eletrônico, é feito um teste para saber se você tem saldo e não ultrapassou o limite ainda.

E por aí vai...tudo é teste, é teste o tempo inteiro e teste pra todo lado.

Duvida? Vamos testar pra ver se é verdade.



# Operadores Relacionais (de comparação) em C++: > , < , >= , <=, == e !=

Neste tutorial de nosso **Curso de C++**, vamos aprender quais são e como usar os operadores relacionais, também conhecidos como operadores de comparação, que são:

- > maior que
- < menor que
- >= maior ou igual
- <= menor ou igual
- == igual
- != diferente

## Como comparar coisas em C++

Embora este tutorial seja um pouco chato e teórico, ele é necessário para entendermos melhor o uso dos testes condicionais IF e ELSE, que iremos usar nos próximos tutoriais.

O que vamos fazer agora é ensinar você a realizar a operação de comparação, ela é feita assim:

**operando1 operador operando2**

Ou seja, vamos usar dois operandos (dados, como números ou caracteres) e um operador (no caso, um relacional).

E esta operação sempre retorna um dos valores:

- TRUE (verdadeiro ou 1)
- FALSE (falso ou 0)

Sempre.

Operação de relação é sempre uma pergunta que a resposta é 1 ou 0, true ou false.

Ok?

## Operador de igualdade em C++: ==

Bom, vamos lá.

Se quisermos saber se um valor x é igual a um y, fazemos:

`x == y`

Veja bem, é: `x == y`

E não: `x = y`

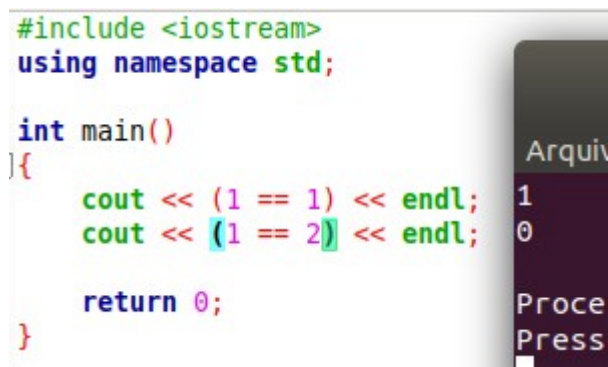
Vamos testar se 1 é igual 1 e depois se é igual a 2, rode o código abaixo:

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

int main()
{
    cout << (1 == 1) << endl;
    cout << (1 == 2) << endl;

    return 0;
}
```

O resultado foi:



The screenshot shows a code editor with the same C++ code as above. To the right, a terminal window displays the output of the program. The first line of output is '1' and the second line is '0'. The terminal window also shows a file explorer on the right with 'Arquiv' and 'Proce' visible, and a 'Press' button at the bottom.

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

int main()
{
    cout << (1 == 1) << endl;
    cout << (1 == 2) << endl;

    return 0;
}
```

Arquiv  
1  
0  
Proce  
Press

Realmente, 1 é igual 1 (deu 1, TRUE, o resultado da comparação).  
E 1 não é igual a 2 (deu 0, FALSE, o resultado da comparação).

## Operador de diferença em C++: !=

Assim como tem um operador que verifica a igualdade, tem outro que verifica a não-igualdade.

Se dois valores forem diferentes, a operação de comparação com != vai resultar TRUE.

Se forem iguais, resulta em FALSE.

Vamos ver na prática, compile e rode:

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

int main()
{
    cout << ('a' != 'b') << endl;

    return 0;
}
```

Ou seja, o caractere 'a' é diferente de 'b'.

Agora teste se 'a' é diferente de 'A'.  
É ?

## Operador de maior em C++: >

Se quisermos testar se um valor X é maior que um valor Y, usamos o operador: >

$X > Y$

Se X for maior, ele retorna TRUE (1).

Se for igual ou menor, ele retorna FALSE (0).

## Operador de maior igual em C++: **>=**

Se quisermos testar se um valor é maior, ou igual, usamos: **>=**

**X >= Y**

Ele retorna 1 (true) se X for maior ou igual que Y, e 0 caso X seja menor.

Por exemplo, para testar se uma pessoa já pode dirigir, beber, entrar no motel...devemos fazer:

**idade >= 18**

## Operador Menor e Menor igual em C++: **<** e **<=**

Analogamente, existem os operador *menor* e *menor igual*.

Para testarmos se uma pessoa pode doar sangue, devemos testar:

**idade < 65**

Para saber quem é isento de declarar imposto de renda, devemos testar:

**valor <= 28559.70**

## Cuidado com comparações

Um erro muito grave é confundir:

**x = y;**

Com:

**x == y;**

No primeiro caso, é uma atribuição.

A variável x está recebendo o valor da variável y.

No segundo caso é uma comparação.

Sempre leia comparação como uma pergunta: x é igual a y?

E quando você pergunta as coisas pro C++ ele só te dá duas respostas: 1 ou 0

Assim, se quiser armazenar o valor de uma comparação, declare um *booleano*, ok?

- **Exercício de C++**

Faça um programa que pede dois valores ao usuário (pode ser um número ou um caractere), em seguida exiba todas as operações de comparação, como no exemplo abaixo:

```
Primeiro valor: 21
Segundo valor: 12
21 > 12 : 1
21 < 12 : 0
21 >= 12 : 1
21 <= 12 : 0
21 == 12 : 0
21 != 12 : 1
```

Tentou? Conseguiu?

Nosso código ficou assim:

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

```
int main()
{
    int op1, op2;

    cout << "Primeiro valor: ";
    cin >> op1;

    cout << "Segundo valor : ";
    cin >> op2;

    cout << op1 << " > " << op2 << " : "<<(op1>op2)<<endl;
    cout << op1 << " < " << op2 << " : "<<(op1<op2)<<endl;
    cout << op1 << " >= " << op2 << " : "<<(op1>=op2)<<endl;
    cout << op1 << " <= " << op2 << " : "<<(op1<=op2)<<endl;
    cout << op1 << " == " << op2 << " : "<<(op1==op2)<<endl;
    cout << op1 << " != " << op2 << " : "<<(op1!=op2)<<endl;
    return 0;
}
```

E o seu?

# O teste condicional IF

Neste tutorial de C++, vamos aprender a usar o teste condicional IF.

Assim, nossos programas vão ficar mais flexíveis e dinâmicos.

## Tomando decisões em C++

Até o momento, todos nossos códigos foram puramente sequenciais.

Ou seja, rodaram de cima pra baixo, na função *main*, executando linha por linha, comando por comando, e sempre desse jeito.

Se rodarmos nossos programas 1 milhão de vezes, eles fazem o mesmo 1 milhão de vezes.

Mas será que é isso que ocorre nos programas do dia-a-dia?

Quando você clica numa coisa, abre uma coisa, se clica em outra, outra coisa abre.

Se digita um comando, algo acontece. Se digita outro, outra coisa diferente acontece.

Ou seja, o que seu computador vai fazer depende do que você faz, ele precisa fazer alguns testes antes de tomar que decisão ele deve tomar.

Vamos aprender como tomar essas decisões agora, usando o teste condicional **if**.

## O comando IF no C++

A estrutura do comando IF é bem simples e fácil de entender, veja:

```
if ( teste_condicional ){  
    // código  
    // código  
    // código  
}
```

Começamos com o comando **if**, em seguida abrimos parêntesis, e dentro deles deve ter algum valor booleano (1 ou 0, true ou false), no caso o mais

comum é fazermos alguma operação relacional (teste de comparação), depois algum código entre chaves.

O funcionamento é bem simples: a instrução **if** vai analisar o que tem dentro dos parêntesis.

Se o que tiver lá for verdadeiro, ela executa o código entre chaves.

Se o que tiver ali dentro for falso, ela não faz nada, o **if** é pulado, como se não existisse.

IF em inglês significa 'se'.

Então, leia: *se for verdade, então executa isso*.

## Como usar o IF em C++

O programa abaixo tem um teste condicional:

$2 > 1$

Se este teste for verdade, o *cout* dentro do IF é executado:

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

int main()
{
    if( 2 > 1){
        cout << "Realmente, dois é maior que 1";
    }
    return 0;
}
```

Obviamente, é verdade, por isso, a mensagem sempre aparece na tela.

Agora teste com  $2 < 1$

O que aconteceu?

## Exemplo de uso do IF

*Faça um programa que peça a idade ao usuário e informe caso ele seja de maior.*

Uma pessoa é de maior se tem 18 ou mais anos de idade, então nosso código fica assim:

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

int main()
{
    int age;

    cout << "Qual sua idade: ";
    cin >> age;

    if( age >= 18){
        cout << "Você é de maior";
    }

    return 0;
}
```

Você consegue resolver esse exercício usando o operador > ao invés do >= ?

## Exemplo de uso do IF

*Pergunte uma nota ao usuário, e caso ela seja menor que 6 diga que ele está reprovado.*

Veja como fica o código:

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

int main()
{
    float grade;
```



```

cout << "Qual sua nota: ";
cin >> grade;

if( grade < 6)
    cout << "Reprovado";

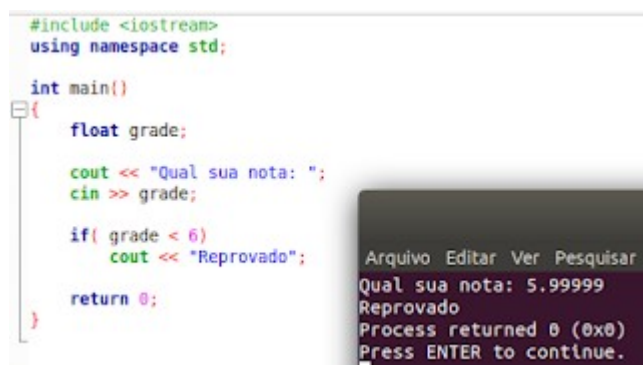
return 0;
}

```

Note que não usamos chaves, isso é porque o código do IF é composto apenas de uma linha de instrução. Nesses casos, não precisa colocar entre chaves.

Mas se a instrução dentro do IF for maior, use chaves.

Veja como o C++ é um professor rigoroso, coloquei a nota 5.99999 e mesmo assim ele me reprovou:



```

#include <iostream>
using namespace std;

int main()
{
    float grade;

    cout << "Qual sua nota: ";
    cin >> grade;

    if( grade < 6)
        cout << "Reprovado";

    return 0;
}

```

```

Arquivo Editar Ver Pesquisar
Qual sua nota: 5.99999
Reprovado
Process returned 0 (0x0)
Press ENTER to continue.

```

## Usando o IF em C++

*Faça um programa que peça dois números ao usuário, e informe caso sejam iguais.*

```

#include <iostream>
using namespace std;

int main()
{
    float num1, num2;

```

```
cout << "Numero 1: ";  
cin >> num1;  
  
cout << "Numero 2: ";  
cin >> num2;  
  
if( num1 == num2)  
    cout << "São iguais!";  
  
return 0;  
}
```

Bem simples, não é?

Note que se você digitar números diferentes, o IF recebe o resultado *false* no teste condicional e não executa seu código.

Ou seja, nada acontece, o programa simplesmente pula o IF.

## Exercício

*Escreva um programa que pede uma senha numérica ao usuário e mostre uma mensagem de erro caso ele ERRE a senha. A senha é 2112.*

# A instrução IF e ELSE

Neste tutorial de nosso **Curso online de C++**, vamos te ensinar a usar a instrução IF ELSE, sem dúvidas um dos comandos mais importantes da programação.

## A Instrução IF e ELSE do C++

No tutorial passado, aprendemos a usar o [teste condicional IF](#), e vimos que ele faz um teste (dã, jura?), e caso o resultado seja VERDADEIRO, ele executa um bloco específico de código. E se for FALSO, não executa, simplesmente pula o IF.

Bom, tem um problema aí...se for TRUE, faz algo...seria interessante se fizesse outra coisa caso fosse FALSE, concorda? É aí que entra o comando **ELSE**.

A dupla IF ELSE tem a seguinte sintaxe:

```
if (teste){  
    // Código caso o  
    // teste seja TRUE  
}  
else {  
    // Código caso o  
    // teste seja FALSE  
}
```

Veja que eu disse dupla. O ELSE só existe se tiver um IF.

O IF pode até vir sozinho, como vimos no tutorial anterior, mas o ELSE só vem com o IF e sempre depois dele, conforme mostrado no escopo desse comando.

Vamos praticar?

## Exemplo de IF e ELSE

*Você foi contratado para fazer um programa para uma casa de shows. Em determinado trecho do código, você vai pedir a idade do cliente. Se for maior de idade (tem 18 anos ou mais), avise que ele pode comprar o ingresso. Se tiver menos, seu programa deve informar que ele não pode entrar na casa.*

Primeiro, pedimos e armazenamos a idade do usuário na variável inteira *age*. Agora, vamos testar se ela é maior ou igual a 18.

Se sim, cai no IF e ele é de maior.

Se não, cai no ELSE e avisamos que ele não pode entrar na casa de shows.

Veja como ficou nosso código:

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

int main()
{
    int age;

    cout << "Sua idade: ";
    cin >> age;

    if (age >= 18){
        cout << "Ok, você é de maior!" << endl;
    }
    else {
        cout << "Você é de menor, não pode entrar!" << endl;
    }

    return 0;
}
```

Você consegue reescrever o programa acima usando o operador relacional > ao invés de >= ?

## Como usar IF e ELSE

*Seu professor te contratou para criar um programa que recebe a nota do aluno. Caso ela seja 7.0 ou mais, ele foi aprovado. Caso contrário, ele está de recuperação.*

Aqui, sem mistério, não é?

Só fazer o teste condicional para maior ou igual a 7:

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

int main()
{
    float grade;

    cout << "Sua nota: ";
    cin >> grade;

    if (grade >= 7)
        cout << "Você passou." << endl;
    else
        cout << "Você está de recuperação." << endl;

    return 0;
}
```

Veja que não usamos as chaves. Nesse caso está ok, pois após o comando IF e após o ELSE, existe apenas uma única instrução de código.

Se dentro do seu IF ou ELSE tiver mais de uma linha de código pra rodar, coloque tudo entre chaves, ok?

## Usando IF e ELSE em C++

*Escreva um programa que pergunta o salário de uma pessoa. Se for maior que 3 mil, tem que pagar 20% de imposto. Se for menor, tem que pagar 15%. Exiba o valor do imposto que a pessoa tem que pagar. Imposto é roubo ?*

Vamos armazenar na variável *sal* o salário e na *tax*, o valor do imposto. Nosso código fica assim:

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

int main()
{
    float sal, tax;

    cout << "Seu salário: ";
    cin >> sal;

    if (sal > 3000){
        tax = 0.20;
        cout << "Imposto devido: R$ " << tax*sal <<endl;
    }
    else{
        tax = 0.15;
        cout << "Imposto devido: R$ " << tax*sal <<endl;
    }
    return 0;
}
```

Primeiro, testamos se o salário é maior que 3 mil. Se for, o valor da taxa será de 20%. Senão, será de 15%.

Depois, apenas calculamos o valor do imposto devido. Bem simples, não?

Consegue encurtar o código anterior?

# Par ou Ímpar: Como descobrir (e outros múltiplos)

Neste tutorial de nossa **Apostila de C++**, vamos aprender como descobrir se um determinado número é par ou ímpar (bem como utilizar outros múltiplos, além do 2).

## Par ou Ímpar em C++

Agora que já aprendemos sobre o [teste condicional IF / ELSE](#), bem como os [operadores matemáticos](#), vamos aprender como usar o operador de resto da divisão (%) para verificar e descobrir se um determinado número é par ou ímpar.

Números pares são, na verdade, números que são múltiplos de 2. Ou seja, que são formados pelo número 2 multiplicado por outro número inteiro qualquer.

Veja alguns exemplos:

$$2 \times 1 = 2$$

$$2 \times 2 = 4$$

$$2 \times 3 = 6$$

$$2 \times 4 = 8$$

...

Ou seja, para saber se um número *num* armazena um par, basta testar se seu resto da divisão por 2 é igual a 0. O teste condicional que representa isso é:

•  $(num \% 2 == 0)$

Se tal operação retornar verdadeiro, é par.

Caso contrário (ELSE), é porque o número é ímpar (é sempre par ou ímpar).

Veja como fica nosso código C++, de um programa que pede um inteiro ao usuário e diz se é par ou ímpar:

```
#include <iostream>
```

```

using namespace std;

int main()
{
    int num;

    cout << "Digite um numero: ";
    cin >> num;

    if (num%2==0)
        cout <<"É par"<<endl;
    else
        cout <<"É impar"<<endl;
    return 0;
}

```

## Múltiplo de 3

Como dissemos, o número par nada mais é que um número múltiplo de 2. Existem também os múltiplos de 3, de 4, de 5, ...

Por exemplo, os números múltiplos de 3 são:

$3 \times 1 = 3$   
 $3 \times 2 = 6$   
 $3 \times 3 = 9$   
 $3 \times 4 = 12$

Para descobrir se determinado número é múltiplo de 3, basta verificar o resto da divisão de qualquer número por 3, veja como fica nosso código:

```

#include <iostream>
using namespace std;

int main()
{
    int num;

    cout << "Digite um numero: ";
    cin >> num;

    if (num%3==0)

```



```

    cout << "É múltiplo de 3" << endl;
else
    cout << "Não é múltiplo de 3" << endl;
return 0;
}

```

## Números múltiplos

*Faça um programa que recebe dois números inteiros: num1 e num2. Em seguida, verifique se num1 é múltiplo de num2.*

```

#include <iostream>
using namespace std;

int main()
{
    int num1, num2;

    cout << "Primeiro numero: ";
    cin >> num1;

    cout << "Segundo numero : ";
    cin >> num2;

    if (num1%num2==0)
        cout << num1 << " é múltiplo de " << num2 << endl;
    else
        cout << num1 << " não é múltiplo de " << num2 << endl;
    return 0;
}

```

## Operador Condicional Ternário ?:

Neste tutorial de nosso **curso de C++**, vamos aprender o que é, para que serve e como usar o operador condicional ternário, o **?:**

### Operador Condicional ?:

O operador **?:** é o único ternário na linguagem, ou seja, o único que aceita três operandos.

A sintaxe desse operador condicional ternário é:

- **condição ? instrução1 : instrução2 ;**

Seu funcionamento é bem simples, a condição **condição** é testada.

Se ela for verdadeira a instrução **instrução1** é executada.

Caso a condição seja falsa, a **instrução2** que é executada.

É basicamente um teste condicional IF ELSE, mas um simples, curtinho, para executar instruções de uma linha só.

Vamos refazer alguns exemplos de programa que fizemos usando IF e ELSE, mas agora usando o operador ternário **?:**

### Como usar o Operador Ternário ?:

*Você foi contratado para fazer um programa para uma casa de shows. Em determinado trecho do código, você vai pedir a idade do cliente. Se for maior de idade (tem 18 anos ou mais), avise que ele pode comprar o ingresso. Se tiver menos, seu programa deve informar que ele não pode entrar na casa.*

Nosso código fica assim:

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

int main()
{
    int age;

    cout << "Sua idade: ";
```

```
cin >> age;
```

```
age>=18 ? (cout << "Ok, você é de maior!") :  
          (cout << "Você é de menor, não pode entrar!") ;
```

```
return 0;
```

```
}
```

Bem simples, não é?

## Operador Ternário **?:** como utilizar em C++

*Seu professor te contratou para criar um programa que recebe a nota do aluno. Caso ela seja 7.0 ou mais, ele foi aprovado. Caso contrário, ele está reprovado.*

Usando o operador ternário:

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

```
int main()
```

```
{
```

```
    float grade;
```

```
    cout << "Sua nota: ";
```

```
    cin >> grade;
```

```
    grade>=7 ? (cout << "Aprovado"):(cout << "Reprovado");
```

```
    return 0;
```

```
}
```

Bem mais enxuto o código.

## Exemplo de uso de **?:**

*Escreva um programa que pergunta o salário de uma pessoa. Se for maior que 3 mil, tem que pagar 20% de imposto. Se for menor, tem que pagar 15%. Exiba o valor do imposto que a pessoa tem que pagar. Imposto é roubo ?*

```
#include <iostream>
```

```

using namespace std;

int main()
{
    float sal, tax;

    cout << "Seu salário: ";
    cin >> sal;

    tax = sal>3000 ? 0.20 : 0.15;
    cout << "Imposto devido: R$ " << tax*sal <<endl;

    return 0;
}

```

Essa solução ficou bem mais geniosa.

A variável *tax* vai receber o valor que retorna do operador condicional, que é 0.20 ou 0.15.

## Usando o operador ternário ?: em C++

*Escreva, usando o operador condicional ternário, um programa que recebe um inteiro e diz se ele é par ou ímpar.*

```

#include <iostream>
using namespace std;

int main()
{
    int num;

    cout << "Digite um numero: ";
    cin >> num;

    num%2==0 ? cout <<"É par" : cout <<"É impar";

    return 0;
}

```

## IF e ELSE aninhados

Neste tutorial de nossa **apostila de C++**, vamos aprender a usar uma técnica muito importante e usada na programação, que são IF ELSE dentro de IF ELSE, aninhados.

### IF e ELSE dentro de IF e ELSE

Vamos explicar a técnica de se aninhar IF e ELSE através de exemplos.

*Crie um programa em C++ que pede dois números, e diz se são iguais ou um é maior que o outro (mostrando quem é quem).*

Vamos lá, vamos armazenar os números nas variáveis: *num1* e *num2*  
O primeiro teste é o *if(num1 == num2)*, para saber se são iguais.  
Se forem, avisa e acabou o programa.

Se não for, temos duas opções:

- 1.num1 é maior que num2
- 2.num2 é maior que num1

Ou seja, temos que fazer mais um teste dentro do ELSE.

Vamos inserir lá dentro o teste:

*if (num1 > num2)*

Se ele for verdade, diz que *num1* é maior que *num2*.

E se for falso?

Aí ele deve cair dentro de outro ELSE, desse segundo *IF* aninhado, entendeu?

Vamos ver como fica nosso código:

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

```
int main()
{
    float num1, num2;

    cout << "Primeiro numero: ";
```

```

cin >> num1;

cout << "Segundo numero: ";
cin >> num2;

if(num1==num2)
    cout <<"São iguais" << endl;
else
    if(num1 > num2)
        cout << num1 <<" é maior que " << num2 << endl;
    else
        cout << num2 <<" é maior que " << num1 << endl;

return 0;
}

```

Veja que fizemos uma indentação, ou seja, demos um espaçamento de modo que cada else fique na mesma vertical do seu respectivo if.

## Exemplo de IF e ELSE aninhados

*Você foi contratado por um hospital para verificar se as pessoas podem doar sangue ou não.*

*Elas só podem doar se tiverem 18 anos ou mais e não tiverem nenhuma doença.*

*Faça um programa em C++ que pergunta a idade da pessoa e se ela possui alguma doença, depois diga se ela pode doar sangue ou não.*

Vamos primeiro ao código:

```

#include <iostream>
using namespace std;

int main()
{
    int age, dis;

    cout <<"Sua idade: ";
    cin >> age;

    cout <<"Você tem alguma doença?"<<endl;
    cout <<"1. Não" << endl;
}

```

```

cout << "2. Sim" << endl;
cin >> dis;

if( age >= 18 ){
    if(dis == 1)
        cout << "Você pode doar sangue!";
    else
        cout << "Você não pode doar sangue, pois está doente";
} else {
    cout << "Você precisa ter 18 anos ou mais para doar sangue";
}

return 0;
}

```

Bem, vamos lá.

Primeiro armazenamos a idade na variável *age*.

Depois, o usuário deve digitar 1 caso não tenha doença e 2 caso tenha, essa resposta ficará armazenada na variável *dis*.

Agora os testes.

Primeiro, verificamos se tem 18 anos ou mais. Se não tiver, cai no ELSE e avisa que precisa ter 18 anos ou mais para doar.

Se sim, se tiver 18 ou mais, vamos verificar se ele não possui doença, ou seja, se digitou 1.

Se não tiver, avisamos que pode doar.

Caso ele tenha digitado 2, avisa que pessoas doentes não podem doar sangue.

Bacana não?

Agora um *teste hacker*: rode o programa acima e encontre algum erro.

## IF e ELSE aninhados em C++

*Se você tiver menos de 16, não pode votar. De 16 até 18 é facultativo. A partir de 65 também.*

*De 18 até 65, é obrigatório votar. Faça um programa em C++ que peça a idade do usuário e diga se ele é obrigado a votar, se é opcional ou se não pode votar.*

Bom, vamos lá.

No primeiro IF vamos logo colocar a 'galera' que tem menos de 16 anos, elas não pode votar.

Se tiver 16 ou mais, vai pro ELSE.

Dentro desse ELSE, vamos verificar se é menor de 18 com um IF, se for, diz que é opcional.

Se não for menor, é porque tem 18 ou mais.

Vamos testar se tem menos de 65 anos, se tiver é obrigado a votar.

Se não tem menos, cai no último ELSE e a pessoa tem mais de 65 anos, votando opcionalmente.

Veja como ficou nosso código:

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

int main()
{
    int age;

    cout << "Sua idade: ";
    cin >> age;

    if(age < 16)
        cout << "Não pode votar." << endl;
    else
        if(age < 18)
            cout << "Voto facultativo." << endl;
        else
            if(age < 65)
                cout << "É obrigado a votar." << endl;
            else
                cout << "Voto facultativo." << endl;
```



```
    return 0;  
}
```

## O comando IF/ELSE IF

*Crie um programa que peça a nota de uma prova para um aluno, que valia de 0.0 até 10 .*

*Se a nota for maior ou igual a 9, diga que ele tirou A.*

*Se for de 8.0 até 8.9, diga que essa pessoa tirou B.*

*Se for de 7.0 até 7.9, diga que ela tirou C.*

*Se for de 6.0 até 6.9, diga que ela tirou D.*

*Se for abaixo de 6.0, ela tirou F.*

O primeiro teste a fazer é ver se a nota é maior ou igual a 9.0, se for, acaba no primeiro IF a execução do programa.

Se não for, vai pro ELSE.

Dentro ELSE precisamos testar com um novo IF se o valor é maior ou igual a 8.0.

Veja bem, só cai nesse IF interno se for de 8.0 até 8.9, acima disso teria caído no primeiro IF.

Se não for, vai cair dentro de um outro ELSE interno.

Dentro desse ELSE, vamos testar (IF) se é de 7.0 até 7.9

Se não for, criamos mais um ELSE e dentro dele um IF para saber se é de 6.0 até 6.9

Por fim, se não for nenhuma das opções acima, é nota F.

Vamos ver como ficou nosso código:

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

```
int main()  
{  
    float grade;
```

```

cout << "Sua nota ";
cin >> grade;

if(grade >= 9)
    cout << "Nota A"<<endl;
else
    if(grade >=8 )
        cout << "Nota B"<<endl;
    else
        if(grade >=7 )
            cout << "Nota C"<<endl;
        else
            if(grade >=6 )
                cout << "Nota D"<<endl;
            else
                cout << "Nota F"<<endl;

return 0;
}

```

Note que o código C++ vai 'indo' pra direita.

Imagina se tivesse mais 10 condições para testar, como ficaria nosso código?

Uma bagunça, não é verdade?

Agora se eu pegar o mesmo código e fizer isso:

```

#include <iostream>
using namespace std;

int main()
{
    float grade;

    cout << "Sua nota ";
    cin >> grade;

    if(grade >= 9)
        cout << "Nota A"<<endl;
    else if(grade >=8 )
        cout << "Nota B"<<endl;
    else if(grade >=7 )

```

```
    cout << "Nota C"<<endl;
else if(grade >=6 )
    cout << "Nota D"<<endl;
else
    cout << "Nota F"<<endl;

return 0;
}
```

Não fica mais legível?

É uma boa técnica de programação, pois facilita a leitura e entendimento do código.

# Operadores Lógicos em C++: AND (&&), OR (||) e NOT (!)

Neste tutorial de nosso **curso de C++**, vamos aprender mais três operadores, chamados operadores lógicos, que nos permitirão fazer testes mais complexos e interessantes. São eles:

- && - AND ou E
- || - OR ou OU
- ! - NOT ou de negação

## Operador Lógico AND: &&

O operador && serve para unir duas expressões lógicas em uma só, assim:

- expressão1 && expressão2

O resultado dessa expressão é verdade somente se a expressão1 E a expressão2 forem verdadeiras.

Se qualquer uma delas for falsa, a expressão geral se torna falsa também.

A tabela verdade do operador && é:

| A     | B     | A && B |
|-------|-------|--------|
| true  | true  | true   |
| true  | false | false  |
| false | false | false  |
| false | true  | false  |

Por exemplo, para você doar sangue, precisa ser de duas condições:

Ser maior de idade

Não ter doenças

Ou seja, pra dor você precisa ser maior E TAMBÉM não ter doenças:

- maioridade && não ter doenças

Se qualquer uma dessas condições não for satisfeita (for *false*), você não pode doar sangue.

Vamos refazer o programinha que diz se é obrigatório você votar ou não.

Para ser obrigatório, deve satisfazer duas condições:

Ter 18 anos ou mais  
Ter menos de 65 anos

Ou seja, precisa ter 18 anos E (AND) precisa ter menos de 65 anos.  
Nosso código C++ fica assim:

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

int main()
{
    int age;

    cout << "Sua idade: ";
    cin >> age;

    if( age >= 18 && age < 65 )
        cout << "Obrigatório votar.";
    else
        cout << "Voto facultativo";

    return 0;
}
```

Veja que ficou bem mais enxuto, pois fizemos dois testes condicionais de uma vez só dentro do IF.

Se a pessoa tiver menos de 18 anos, a primeira expressão vai dar falsa e o teste inteiro se torna falso, indo pro ELSE.

Se ela tiver 65 ou mais, a segunda expressão já se torna falsa, deixando o resultado do IF falso também, indo pro ELSE.

Em ambos casos, o voto é facultativo.

## Operador Lógico OR: ||

O operador lógico || (OU) une duas expressões assim:  
expressão1 || expressão2

Ele retorna verdadeiro se qualquer uma das expressões forem verdadeiras.

Ou seja, só retorna falso se ambas forem falso. Confira a tabela verdade deste operador:

| A     | B     | A    B |
|-------|-------|--------|
| true  | true  | true   |
| true  | false | true   |
| false | true  | true   |
| false | false | false  |

Por exemplo, se numa rodovia a velocidade máxima é de 80 km/h, andar acima disso e abaixo da metade (40 km/h) dá multa. Vamos testar:

- `velocidade > 80 || velocidade < 40`

Ou seja: se a velocidade for maior que 80 km/h OU for abaixo de 40 km/h, o motorista vai levar uma multa. Se alguma das expressões for verdadeira, o teste inteiro é verdadeiro.

Veja um programa que recebe a velocidade de um carro e diz se vai gerar multa ou não:

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

int main()
{
    int speed;

    cout << "Velocidade: ";
    cin >> speed;

    if( speed > 80 || speed < 40 )
        cout << "Vai levar multa.";
    else
        cout << "Tudo ok.";

    return 0;
}
```

O IF detecta se você vai levar multa, ou seja, pra não levar precisa cair no ELSE, e só cai no ELSE se ambas as expressões forem falsas, logo, você não está nem acima de 80 nem está abaixo de 40, logo está entre 40 e 80, que é o correto. Sacou ? OU uma coisa OU outra.

## Operador Lógico NOT: !

Por fim, temos o operador de negação.

Ele transforma o que é falso em verdadeiro, e o que é verdadeiro em falso.

Veja a tabela verdade dele:

| A     | !A    |
|-------|-------|
| true  | false |
| false | true  |

Ou seja:

!true é o mesmo que false

!false é o mesmo que true

Refazendo o primeiro código:

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

int main()
{
    int age;

    cout << "Sua idade: ";
    cin >> age;

    if( !(age>=18 && age<65) )
        cout << "Voto facultativo.";
    else
        cout << "Obrigatório votar";

    return 0;
}
```

Refazendo o segundo código:

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

int main()
{
    int speed;

    cout << "Velocidade: ";
    cin >> speed;

    if( !(speed>80 || speed<40) )
        cout << "Tudo ok.";
    else
        cout << "Vai levar multa.";

    return 0;
}
```

Pode parecer mais complicado ou a mesma coisa, de fato é a mesma coisa, mas muitas vezes vai ser mais lógico e vai fazer mais sentido usar o operador NOT em alguns testes condicionais. Veremos isso algumas vezes, no decorrer de nosso **curso de C++**.

A precedência de operadores entre eles é, de cima pra baixo:

!

&&

[]

## Exercício de Operadores Lógicos

No tutorial passado, sobre IF e ELSEs aninhados, propusemos uma questão sobre converter uma nota numérica em A, B, C... mas tem um bug ali.

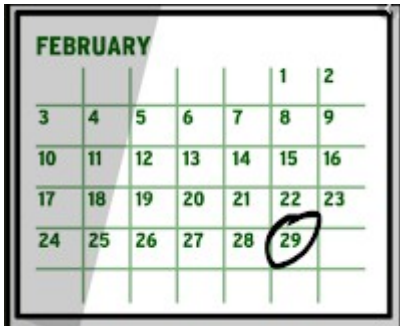
E se o usuário digitar menos que 0 ou mais que 10?

É um erro. Ajeite o programa usando o operador lógico || para caso digitem notas abaixo de 0 ou acima de 10.



# Ano bissexto em C++

Vamos usar nossos conhecimentos de [operadores lógicos](#) para aprender como detectar se um ano é bissexto ou não.



## Anos bissextos

Normalmente, o ano tem 365 dias.

Na verdade, é um valor quebrado, são 365 dias e 6h ou 365,25 dias...mas ia ser estranho um dia com menos de 24 horas, ia ser confuso.

Então, pra compensar essas partes quebradas de dia, vez e outra temos um ano com 366 dias, que é a data 29 de fevereiro.

E eles ocorrem a cada 4 anos, menos nos múltiplos de 100 que não são múltiplos de 400.

Ou seja, ele também é bissexto se for múltiplo de 400.

Calma, respira fundo, leia de novo...é um pouco confuso mesmo no começo. Dê uma estudada sobre a história dos anos bissextos, calendários etc, pra relaxar um pouco:

[https://pt.wikipedia.org/wiki/Ano\\_bissexto](https://pt.wikipedia.org/wiki/Ano_bissexto)

Basicamente, de 4 em 4 anos temos anos bissextos:

1996, 2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036...

De 100 em 100 anos não temos ano bissexto...:

100, 200, 300 ... não são bissextos

... exceto se forem múltiplos de 400:

400, 800, 1200, 1600... são bissextos

Assim:

...

400 – é bissexto

500 – não é bissexto

600 – não é bissexto

700 – não é bissexto

800 – é bissexto

...

1200 – é bissexto

1300 – não é bissexto

1400 – não é bissexto

1500 – não é bissexto

1600 – é bissexto

1700 – não é bissexto

1800 – não é bissexto

1900 – não é bissexto

2000 – é bissexto

2100 – não é bissexto

2200 – não é bissexto

2300 – não é bissexto

2400 – é bissexto

...

## Ano bissexto em C++

Ok, vamos colocar isso em prática, criando código C++.

Um problema difícil nada mais é que vários problemas fáceis.

Então vamos quebrar esse algoritmo em um menor, mais simples.

Primeiro, vamos verificar os anos múltiplos de 400. Se for múltiplo de 400, já era, é só sucesso, é bissexto e acabou:

E como verifica isso? Simples:

- `if (year % 400 == 0)`

Prontinho. Só dizer que é bissexto.

E se não for? Bom, aí cai no else

Agora vem a parte do 4 anos...a cada 100 anos...um rolo só.

Bom, é assim, precisamos verificar duas coisas:

- Se é múltiplo de 4 (ocorre a cada 4 anos, a partir do ano 0: 4, 8, 12, ... 1996, 2000)
- Se não é múltiplo de 100

Fazendo isso em C++:

- Múltiplo de 4: (year % 4 == 0)
- Não é múltiplo de 100: (year % 100 != 0)

Unindo os dois, temos o próximo IF (dentro do else), usando o operador lógico AND &&:

- (year % 4 == 0) && (year % 100 != 0)

Assim, nosso código fica:

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

int main()
{
    int year;

    cout << "Ano: ";
    cin >> year;

    if(year % 400 == 0)
        cout << "É bissexto" << endl;
    else
        if( (year % 4 == 0) && (year % 100 != 0) )
            cout << "É bissexto" << endl;
        else
            cout << "Não é bissexto" << endl;

    return 0;
}
```

# Algoritmo do ano bissexto em C++

Programador gosta de escrever pouco.

Quanto mais simples e direto for o código, melhor.

Note no código anterior que temos duas condições para o ano ser bissexto:

1. (year % 400 == 0)
2. ( (year % 4 == 0) && (year % 100 != 0) )

Se acontecer uma coisa ou outra, é bissexto.

Ou...huumm...ou lembra o que? Operador lógico ||

Ou seja, podemos unir as duas condições e ter somente um teste condicional IF:

(year % 400 == 0) || ( (year % 4 == 0) && (year % 100 != 0) )

Nosso código fica assim então:

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

int main()
{
    int year;

    cout << "Ano: ";
    cin >> year;

    if( (year % 400 == 0) || ( (year % 4 == 0) && (year % 100 != 0) ) )
        cout << "É bissexto" << endl;
    else
        cout << "Não é bissexto" << endl;

    return 0;
}
```

Bonito, hein?

# O comando SWITCH em C++

Neste tutorial de nossa **apostila de C++**, vamos aprender a usar mais um comando bem interessante, a instrução switch, também chamada *instrução switch case*. Vamos ver para que serve, como funciona, quando e como usar ela.

## Tomando caminhos diferentes - Ramificações

Durante nossos estudos de [testes condicionais em C++](#), aprendemos na prática que os comandos IF e ELSE são usados para dar rumos diferentes aos nossos códigos.

Se você digita algo, uma coisa acontece.

Se digita outra, outra coisa diferente acontece.

E por aí vai. O computador reage aos seus comandos.

Ocorrem as chamadas *ramificações* (branches) de um programa.

Isso tudo foi feito com os testes condicionais IF e ELSE.

Porém, existe uma outra possibilidade de trabalharmos com ramificações, que é usando a instrução **switch** em C++.

## Comandos SWITCH e CASE em C++

O *switch* é uma instrução de opção múltipla, usada para tomar ações diferentes dependendo do valor fornecido. Vamos ver o escopo do comando switch:

```
switch (expressão)
{
    case value1:
        // código caso expressão
        // seja igual a value1

    case value2:
        // código caso expressão
        // seja igual a value2

    case value3:
        // código caso expressão
```

```
// seja igual a value3
```

**default:**

```
// código caso expressão  
// não seja nenhum case acima
```

```
}
```

Funciona assim...

Primeiro, o *switch* vai testar a *expressão*.

Depois ela vai comparar com cada um dos *case* (ramificações).

Se *expressão* for igual a *value1*, todo o código daquele *case* em diante será executado.

Se *expressão* for igual a *value2*, tudo que vier depois daquele *case* em diante será executado (o primeiro *case* vai ser ignorado).

E assim por diante.

Se o valor de *expressão* (que deve ser sempre um inteiro) não 'bater' com nenhum *case*, o que está no *default* é que será executado.

## Criando um menu com o comando **SWITCH**

Sem dúvidas, a maior utilidade da instrução *switch* é criar menus.

Vamos criar um menu simples, de um sistema bancário.

Vamos exibir alguns *couts* com as opções:

1. Saque
2. Extrato
3. Transferência
4. Depósito.

Em cada *case* do *switch*, dizemos que opção ela escolheu.

Caso a pessoa digite qualquer coisa que não seja essa opção, cai no *default* que avisa que a opção é inválida.

Nosso código fica:

```

#include <iostream>
using namespace std;

int main()
{
    int op;

    cout << "1. Saque" << endl;
    cout << "2. Extrato" << endl;
    cout << "3. Transferência" << endl;
    cout << "4. Depósito" << endl;
    cout << "Digite sua opção: ";
    cin >> op;

    switch(op)
    {
        case 1:
            cout << "Opção selecionada: Saque"<<endl;
        case 2:
            cout << "Opção selecionada: Extrato"<<endl;
        case 3:
            cout << "Opção selecionada: Transferência"<<endl;
        case 4:
            cout << "Opção selecionada: Depósito"<<endl;
        default:
            cout << "Opção inválida"<<endl;
    }

    return 0;
}

```

Agora teste...digite 1, por exemplo.  
 Ou 2...notou algo de estranho? Algo de errado?

Veja:

```

1. Saque
2. Extrato
3. Transferência
4. Depósito
Digite sua opção: 2
Opção selecionada: Extrato
Opção selecionada: Transferência
Opção selecionada: Depósito
Opção inválida

Process returned 0 (0x0)   execution time : 1.970 s
Press ENTER to continue.

```

Não era pra aparecer as opções debaixo, é como se tivesse executado os cases de baixo também.

Até o default está sempre sendo selecionado pelo switch! Não é assim que a gente queria nosso programa!

Por que isso ocorreu?

## O comando **BREAK** no Switch

Note uma coisa interessante na nossa definição e explicação sobre a instrução *switch case*: ela executa o case que bater com o valor a ser comparado, dali pra baixo. Ou seja, executa aquele case em questão, e todos os outros que estiverem abaixo! Inclusive o default.

Para que somente o código de cada case seja executado, basta adicionar ao final de cada case o seguinte comando:

`break;`

Veja como fica nosso código:

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

int main()
{
    int op;

    cout << "1. Saque" << endl;
    cout << "2. Extrato" << endl;
    cout << "3. Transferência" << endl;
    cout << "4. Depósito" << endl;
    cout << "Digite sua opção: ";
    cin >> op;

    switch(op)
    {
        case 1:
            cout << "Opção selecionada: Saque"<<endl;
            break;
```



```

    case 2:
        cout << "Opção selecionada: Extrato"<<endl;
        break;

    case 3:
        cout << "Opção selecionada: Transferência"<<endl;
        break;

    case 4:
        cout << "Opção selecionada: Depósito"<<endl;
        break;

    default:
        cout << "Opção inválida"<<endl;
}

return 0;
}

```

Agora rode e veja que menu bonitinho e bacana! Você seleciona a opção, e ele executa o código correto, com o case correto!

## Exemplo de uso do **SWITCH CASE**

*Crie um programa que recebe um valor numérico do usuário, de 1 até 7, e diga que dia da semana é. Por exemplo, domingo é 1, segunda é 2, terça é 3... Diga que ele digitou um valor errado também, caso o faça.*

A título de curiosidade, veja como ficaria usando IF e ELSE:

```

#include <iostream>
using namespace std;

int main()
{
    int day;

    cout << "Dia da semana: ";
    cin >> day;
}

```

```

if(day==1)
    cout <<"Domingo \n";
else if(day==2)
    cout <<"Segunda \n";
else if(day==3)
    cout <<"Terça \n";
else if(day==4)
    cout <<"Quarta \n";
else if(day==5)
    cout <<"Quinta \n";
else if(day==6)
    cout <<"Sexta \n";
else if(day==7)
    cout <<"Sábado \n";
else
    cout <<"Dia inválido \n";

return 0;
}

```

Veja que coisa medonha, feia e imoral.

Agora vamos ver como isso fica bem bonito e arrumadinho com SWITCH case:

```

#include <iostream>
using namespace std;

int main()
{
    int day;

    cout << "Dia da semana: ";
    cin >> day;

    switch(day)
    {
        case 1:
            cout <<"Domingo \n";
            break;
        case 2:
            cout <<"Segunda \n";
            break;

```

```

    case 3:
        cout << "Terça \n";
        break;
    case 4:
        cout << "Quarta \n";
        break;
    case 5:
        cout << "Quinta \n";
        break;
    case 6:
        cout << "Sexta \n";
        break;
    case 7:
        cout << "Sábado \n";
        break;
    default:
        cout << "Dia inválido";
}

return 0;
}

```

Beeeeem melhor, não acha?

## Cases acumulados

Uma técnica muito conhecida e usada é a de *acumular* cases, fazendo o programa executar vários de uma vez. Vamos para um exemplo:

*Escreva um programa, usando switch case, que solicita uma letra ao usuário: A, B ou C, e diga que letra foi digitada. Certifique-se que ele escreveu tanto A como a, por exemplo, é a mesma coisa.*

Pessoal, por mais que você diga: digite 'B' para o usuário, alguns vão digitar 'b' outros 'bê', o limite da criatividade do usuário é infinito, e você tem que imaginar as besteiras que eles podem fazer e tratar esses casos.

Veja como fica a solução usando cases acumulados:

```

#include <iostream>
using namespace std;

int main()
{
    char let;

    cout << "Digite uma letra: ";
    cin >> let;

    switch(let)
    {
        case 'a':
        case 'A':
            cout << "Você digitou A\n";
            break;

        case 'b':
        case 'B':
            cout << "Você digitou B\n";
            break;

        case 'c':
        case 'C':
            cout << "Você digitou C\n";
            break;

        default:
            cout << "Letra inválida";
    }

    return 0;
}

```

Note que se digitar 'a', cai no *case 'a':* que também executa o *case 'A'*, pois não tem *break* ali.

Ou seja, aquele case é selecionado e roda o de baixo.

*char* é um tipo de dado, caractere. Caracteres são representados por letras entre aspas simples.

Aprenderemos mais sobre isso na seção de **strings em C++**.

## Exercício de SWITCH CASE massa

*Usando os conceitos de switch case, faça um programa que pergunte o mês ao usuário (número de 1 até 12), e diga quantos dias aquele mês possui. Fevereiro tem 28 dias (não é bissexto).*

No [tutorial de exercícios propostos de teste condicionais](#), vamos propor e resolver esse exercício. Veja a solução lá, mas não antes de tentar bastante resolver.

## Exercícios de Testes Condicionais em C++

Pessoal, encerramos mais uma vez uma seção de nosso **curso de C++**. Parabéns se chegou aqui. Agora vamos praticar nossos conhecimentos fazendo alguns exercícios.

E lembrem-se: tentem bastante, mas muito, muito mesmo.

Não de? Pensa mais um pouco. Depois mais, dá uma volta, esfria a cabeça, e tentem de novo.

Resolver problemas é sinônimo de ser programador, portanto tente resolver os exercícios abaixo, usando seus conhecimentos básicos de C++ e de testes condicionais.

### Problemas de testes condicionais em C++

01. Faça um programa que peça dois números, e diga qual é maior ou se são iguais.
02. Faça um programa que diz se um número recebido é positivo ou negativo.
03. Faça um programa que pede dois inteiro e armazene em duas variáveis. Em seguida, troque o valor das variáveis e exiba na tela
04. Faça um programa que peça 3 números e os coloque em ordem crescente.
05. Faça um programa que peça 3 números e os coloque em ordem decrescente.
06. Programe um software que recebe as coordenadas X e Y do usuário, e diga em qual quadrante o ponto está.
07. Faça um programa de calculadora simples, que pede dois números ao usuário, em seguida exibe um menu onde ele vai escolher que operação será realizada. A operação e a saída devem estar em um switch case.

08. Usando os conceitos de switch case, faça um programa que pergunte o mês ao usuário (número de 1 até 12), e diga quantos dias aquele mês possui. Fevereiro tem 28 dias (não é bissexto).

09. Usando os conceitos de switch case, faça um programa que pergunte o mês ao usuário (número de 1 até 12), e diga quantos dias aquele mês possui, incluindo se é ano bissexto ou não.

10. Faça um programa que recebe os três lados de um triângulo e diz se é equilátero, isósceles ou escaleno.

11. Faça um programa que calcule as raízes de uma equação do segundo grau, na forma  $ax^2 + bx + c$ . O programa deverá pedir os valores de a, b e c e fazer as consistências, informando ao usuário nas seguintes situações:

Se o usuário informar o valor de A igual a zero, a equação não é do segundo grau e o programa não deve fazer pedir os demais valores, sendo encerrado;

Se o delta calculado for negativo, a equação não possui raízes reais. Informe ao usuário e encerre o programa;

Se o delta calculado for igual a zero a equação possui apenas uma raiz real; informe-a ao usuário;

Se o delta for positivo, a equação possui duas raiz reais; informe-as ao usuário;

# Soluções dos exercícios de Testes condicionais

01.

## Como comparar dois números em C++

Vamos armazenar os números nas variáveis *num1* e *num2*.

Primeiro, vamos testar se a primeira é maior que a segunda, se for cai no primeiro IF e diz isso no cout.

Se não for, cai no ELSE.

Nesse ELSE, o segundo número é maior que o primeiro, ou pode ser igual.

Testamos isso com um novo IF aninhado, para saber se num2 é maior que num1, se for, avisamos isso.

Caso num1 não seja maior que num2, ou num2 não seja maior que num1, então cai no ELSE aninhado e temos necessariamente que os números são iguais.

Veja como ficou nosso código:

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

int main()
{
    int num1, num2;

    cout << "Numero 1: ";
    cin >> num1;
    cout << "Numero 2: ";
    cin >> num2;

    if(num1>num2)
        cout << num1 << " é maior que " << num2 << endl;
    else
        if(num2>num1)
            cout << num2 << " é maior que " << num1 << endl;
        else
            cout << "São iguais" << endl;
```



```
    return 0;
}
```

02.

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

int main()
{
    int num;

    cout << "Numero: ";
    cin >> num;

    if(num<0)
        cout <<"Negativo"<<endl;
    else
        cout <<"Positivo"<<endl;
}
```

03.

## Como trocar o valor de dois números em C++

Vamos pedir ao usuário dois números e armazenar nas variáveis `num1` e `num2`.

Agora vamos trocar, inverter esses valores.

A primeira variável recebe o valor da segunda:

- `num1 = num2;`

Agora a segunda recebe o valor da primeira:

- `num2 = num1;`

Simple e fácil, não?

**Não.** Tá errado!

A primeira operação ta ok, agora o valor de *num1* é o de *num2*.

Porém, quando fazemos a segunda variável pegar o valor da primeira, esse valor da primeira mudou, não é mais aquele original, ele se perdeu, agora o que tem na primeira variável é o valor da segunda.

O que temos que fazer é armazenar o valor inicial dessa primeira variável. Vamos guardar numa variável auxiliar, a *aux*.

- `aux = num1;`

Pronto, agora fazemos:

- `num1 = num2;`

E agora como pegamos o valor antigo de `num1`? Só pegar da *aux*:

- `num2 = aux;`

Prontinho, valores invertidos!

```
Numero 1: 21
Numero 2: 12
Primeiro valor: 21
Segundo valor: 12

Invertendo...
Primeiro numero: 12
Segundo numero: 21
```

Veja como fica nosso código:

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

int main()
{
    int num1, num2, aux;

    cout << "Numero 1: ";
    cin >> num1;

    cout << "Numero 2: ";
    cin >> num2;
```

```

cout << "\nPrimeiro valor: " << num1 << endl;
cout << "Segundo valor: " << num2 << endl;
cout << "\nInvertendo...\n";

aux = num1;
num1 = num2;
num2 = aux;

cout << "Primeiro numero: " << num1 << endl;
cout << "Segundo numero: " << num2 << endl;

return 0;
}

```

04. e 05.

### 3 números em ordem crescente

Vamos armazenar os números nas variáveis num1, num2 e num3, e também usaremos uma variável temporária, uma auxiliar, a temp.

A ideia é a seguinte: queremos colocar o maior valor digitado na variável num1. O segundo maior valor, na variável num2 e o menor na num3.

Para isso, vamos fazer três comparações:

1. Comparar num1 com num2, se num2 for maior, invertamos os valores. Após esse primeiro teste, garantimos que o valor contido em num1 é maior que num2.
2. Comparar num1 com num3, se num3 for maior, invertamos os valores. Após esse segundo teste, garantimos que o valor contido em num1 é maior que em num2 e agora maior que num3.
3. Agora temos num2 e num3, se num3 for maior, devemos inverter os valores, assim teremos o valor contido em num2 maior que num3.

Para fazer as inversões de valores, usaremos o tutorial anterior.

Nosso código fica:

```

#include <iostream>
using namespace std;

int main()

```

```

{
    int num1, num2, num3, temp;

    cout << "Numero 1: ";
    cin >> num1;

    cout << "Numero 2: ";
    cin >> num2;

    cout << "Numero 3: ";
    cin >> num3;

    if(num2 > num1){
        temp = num1;
        num1 = num2;
        num2 = temp;
    }

    if(num3 > num1){
        temp = num1;
        num1 = num3;
        num3 = temp;
    }

    if(num3 > num2){
        temp = num2;
        num2 = num3;
        num3 = temp;
    }

    cout << num1 << " >= " << num2 << " >= " << num3 << endl;
}

```

### 3 números em ordem decrescente

A lógica é a mesma da anterior, apenas queremos que num1 tenha o menor valor, num2 o segundo menor valor e num3 o maior valor.

Nosso código fica:

```

#include <iostream>
using namespace std;

int main()

```

```

{
    int num1, num2, num3, temp;

    cout << "Numero 1: ";
    cin >> num1;

    cout << "Numero 2: ";
    cin >> num2;

    cout << "Numero 3: ";
    cin >> num3;

    if(num2 < num1){
        temp = num1;
        num1 = num2;
        num2 = temp;
    }

    if(num3 < num1){
        temp = num1;
        num1 = num3;
        num3 = temp;
    }

    if(num3 < num2){
        temp = num2;
        num2 = num3;
        num3 = temp;
    }

    cout << num1 << " <= " << num2 << " <= " << num3 << endl;
}

```

08.

## Quantos dias tem no mês

Vamos lá.

Meses com 28 dias: 2

Meses com 30 dias: 4, 6, 9 e 11

Meses com 31 dias: 1, 3, 5, 7, 8, 10 e 12

Vamos fazer acumulando cases, para você fixar melhor esta importante técnica usando switch.

A variável *days* inicia com 0.

A lógica é a seguinte...todos os meses tem 28 dias ou mais. Assim, queremos que o 'case 2' seja sempre executado, então ele vai ficar lá embaixo.

Lá, iremos adicionar o valor 28 a variável *days*.

Pronto, qualquer mês que você digite vai ter pelo 28 dias, pois não usaremos *break* nos cases, pra acumular.

- `days = days + 28;`

Os cases do meio são os dos meses 4, 6, 9 e 11 tem 30 dias.

Como já vamos adicionar 28 dias, só temos que adicionar mais 2 dias pra ficar 30, nesses cases:

- `days = days + 2;`

Veja bem: se digitar 2, cai só no 'case 2', que coloca 28 dias na variável. Se digitar 4, 6, 9 ou 11, cai nesses cases que soma 2 e depois cai no 'case' que soma mais 28 dias, totalizando 30 dias.

Então, caso digitem 1, 3, 5, 7, 8, 10 ou 12, devemos somar apenas +1:

- `days = days + 1;`

Veja como ficou nosso código:

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

int main()
{
    int days=0, month;

    cout << "Numero do mes: ";
    cin >> month;

    switch(month)
    {
        case 1: case 3: case 5: case 7:
```

```
case 8: case 10: case 12:
```

```
    days = days + 1;
```

```
case 4: case 6:
```

```
case 9: case 11:
```

```
    days = days + 2;
```

```
case 2:
```

```
    days=days+28;
```

```
    break;
```

```
default:
```

```
    cout <<"Mês inválido"<<endl;
```

```
}
```

```
cout <<"O mês " <<month<<" tem " <<days<<" dias.\n";
```

```
}
```

Deu pra entender? É uma solução bem *charmosa*.

09.

Aqui, além do que tem no código anterior, temos que perguntar o ano ao usuário. Se for bissexto, devemos adicionar 29 no 'case 2' na variável *days*, se não for, continuamos adicionando apenas 28.

Fazemos isso com uma instrução IF ELSE interno ao case do *switch*.



Veja como fica o código:

```
#include <iostream>
```

```
using namespace std;
```

```

int main()
{
    int days=0, month, year;

    cout << "Numero do mes: ";
    cin >> month;

    cout << "Ano: ";
    cin >> year;

    switch(month)
    {
        case 1: case 3: case 5: case 7:
        case 8: case 10: case 12:
            days = days + 1;

        case 4: case 6:
        case 9: case 11:
            days = days + 2;

        case 2:
            if( (year % 400 == 0) || ( (year % 4 == 0) && (year % 100 != 0) ) )
                days = days+29;
            else
                days = days+28;
            break;

        default:
            cout << "Mês inválido"<<endl;
    }

    cout << "O mês " << month << " tem " << days << " dias.\n";
}

```

10.

## Tipos de triângulo em C++

Existem três tipos de triângulo:

1. Equilátero: todos os lados são iguais
2. Isósceles: apenas dois lados são iguais
3. Escaleno: todos os lados são diferentes



Então tudo que temos que fazer é pedir três lados de um triângulo, e sair testando se os lados são todos iguais, ou tem dois iguais ou se é tudo diferente.

Uma maneira de fazer é testando se é tudo igual em um IF, pra saber se é equilátero:

- `if( (a==b) && (b==c) )`

Depois se são todos diferentes, pra saber se é escaleno:

- `if( (a!=b) && (a!=c) && (b!=c) )`

Se não for nenhuma das alternativas acima, é porque ele é isósceles.

Nosso código fica assim:

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

int main()
{
    int a, b, c;

    cout << "Lado a: ";
    cin >> a;

    cout << "Lado b: ";
    cin >> b;

    cout << "Lado c: ";
    cin >> c;

    if( (a==b) && (b==c) )
        cout<<"Equilátero\n";
    else if( (a!=b) && (a!=c) && (b!=c))
        cout<<"Escaleno\n";
    else
        cout<<"Isósceles\n";
}
```

## Equilátero, Isósceles ou Escaleno ?

Uma coisa interessante sobre a programação é que o código é uma espécie de impressão digital.

Cada um faz do seu jeito, cada um tem seus métodos, linhas de raciocínio e criatividade.

As vezes é comum fazermos um código grande, feio e confuso.

Depois, vemos alguém usando metade das linhas que usamos, fazendo algo bem mais bonito e abrangente.

Por isso a importância de estudar por livros, sites, tutoriais e códigos de outras pessoas, pra 'pegar' o raciocínio dos outros. Nunca perca esse costume, ok?

Vamos para mais uma solução.

Primeiro, vamos testar se tem dois lados iguais:

```
(a==b) || (a==c) || (b==c)
```

Se tiver, das duas uma: ou é equilátero ou só isósceles.

Então, testamos se os três lados são iguais:

```
(a==b) && (b==c)
```

Se for, dizemos que é equilátero. Senão, cai no ELSE aninhado e dizemos que é isósceles.

Se não tiver pelo menos dois lados iguais, é escaleno.

Veja:

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

```
int main()
{
    int a, b, c;

    cout << "Lado a: ";
    cin >> a;
```

```

cout << "Lado b: ";
cin >> b;

cout << "Lado c: ";
cin >> c;

if( (a==b) || (a==c) || (b==c))
    if( (a==b)&&(b==c) )
        cout<<"Equilátero\n";
    else
        cout<<"Isósceles\n";
else
    cout<<"Escaleno\n";
}

```

## Teste hacker C++

Hacker é aquele que acha brechas, erros, problemas em códigos. Tem um problema no código acima: as condições de existência de um triângulo.

Não é só digitar três valores e temos um triângulo, não. Pesquise sobre as condições de existência de um triângulo, e antes de decidir se é equilátero, isósceles ou escaleno, veja se o triângulo pode sequer existir.

11.

## Fórmulas de Bháskara em C++

Primeiro, pedimos os três coeficientes  $a$ ,  $b$  e  $c$ . Também declaramos a variável *delta* para armazenar o delta (jura?) e root1 e root2, para armazenar as raízes.

O primeiro teste é verificar se  $a$  é diferente de 0, somente se for é que podemos calcular as raízes.

Se não for, cai no ELSE e avisamos que a equação não existe.

Sendo diferente de 0, temos que calcular o delta:

De Bháskara:  $\Delta = b^2 - 4ac$

Agora, vamos testar o delta.

Se for negativo, dizemos que não tem raiz real.

Se for 0, calculamos a única raiz:  $(-b/2a)$

Se for maior que 0, calculamos as duas raízes  $(-b-\text{raiz}(\Delta))/2a$  e  $(-b+\text{raiz}(\Delta))/2a$

Veja como ficou nosso código:

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

int main()
{
    float a, b, c, delta,
          root1, root2;

    cout << "Coeficiente a: ";
    cin >> a;

    cout << "Coeficiente b: ";
    cin >> b;

    cout << "Coeficiente c: ";
    cin >> c;

    if(a != 0){
        delta = (b*b) - (4*a*c);

        if(delta<0){
            cout <<"Não tem raízes reais\n";
        }
        else if (delta==0){
            root1=(-b)/(2*a);
            cout << "Possui apenas uma raiz real: "<<root1<<endl;
        }else{
            root1=(-b - sqrt(delta))/(2*a);
            root2=(-b + sqrt(delta))/(2*a);
        }
    }
```

```

        cout << "Raiz 1: "<<root1<<endl;
        cout << "Raiz 2: "<<root2<<endl;
    }
} else{
    cout <<"a=0, não é uma equação do segundo grau\n";
}
}

```

## Desafio de C++

Você consegue resolver essa questão para dar raízes complexas?  
Veja o código-fonte:

```

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

int main()
{
    float a, b, c, delta,
          root1, root2;

    cout << "Coeficiente a: ";
    cin >> a;

    cout << "Coeficiente b: ";
    cin >> b;

    cout << "Coeficiente c: ";
    cin >> c;

    if(a != 0){
        delta = (b*b) - (4*a*c);

        if(delta<0){
            delta = -delta;
            cout<<"Raiz 1: "<<(-b/(2*a))<<"+ "<<(sqrt(delta)/(2*a))<<"i\n";
            cout<<"Raiz 2: "<<(-b/(2*a))<<"- "<<(sqrt(delta)/(2*a))<<"i\n";
        }
        else if (delta==0){
            root1=(-b)/(2*a);
            cout << "Possui apenas uma raiz real: "<<root1<<endl;
        }
    }
}

```

```

    }else{
        root1=(-b - sqrt(delta))/(2*a);
        root2=(-b + sqrt(delta))/(2*a);
        cout << "Raiz 1: "<<root1<<endl;
        cout << "Raiz 2: "<<root2<<endl;
    }
}else{
    cout <<"a=0, não é uma equação do segundo grau\n";
}
}

```

# Laços e Loopings

Na seção anterior, de [Testes condicionais](#), aprendemos como direcionar o fluxo de um programa, através do uso de IF, ELSE e SWITCH, que são chamadas também de estruturas de seleção.

Agora, vamos aprender a usar as estruturas de repetição, também conhecidas como laços ou loopings.

Com elas, vamos permitir que nossos programas repitam determinados trechos de códigos o quanto quisermos. E isso é absurdamente comum, muito utilizado no mundo da computação.

Iremos aprender a usar os loopings:

1. while
2. do while
3. for

# Operadores de Atribuição, de incremento e decremento

Antes de entrarmos de cabeça nos estudos dos [laços e loopings em C++](#), precisamos conhecer e aprender a usar alguns operadores que são importantes no estudo das estruturas de repetição.

Esses operadores vão nos permitir 'economizar' código, ou seja, escrever menos.

Obviamente, nós programadores não somos preguiçosos. Apenas somos eficientes, ok?

## Operadores de Atribuição

Já usamos algumas vezes expressões do tipo:

```
x = x + 1;
```

Isso quer dizer: "faça com que a variável x tenha o seu antigo valor somado de 1".

Porém, isso pode ser abreviado para:

```
x += 1;
```

Significa a mesma coisa.

Se quisermos subtrair y de x, podemos fazer:

```
x = x - y;
```

Ou de maneira mais simplificada:

```
x -= y;
```

Para fazer x receber o valor do produto de x por y:

```
x = x * y;
```

Ou de uma maneira mais preguiçosa:

```
x *= y;
```

O mesmo vale para divisão e resto da divisão (módulo):



$x /= y$ ; é o mesmo que  $x = x / y$ ;  
 $x \%= y$ ; é o mesmo que  $x = x \% y$ ;

Bem simples, não? Apenas uma maneira de escrever menos.

## Operadores de Incremento e Decremento

Acha que fazer:

$x += 1$ ;

$x -= 2$ ;

É escrever pouco?

Existem duas maneiras de fazer isso, para cada expressão:

$x++$  ou  $++x$

$x--$  ou  $--x$

Se usarmos essas expressões de maneira 'solta':

$x++$ ; e  $++x$ ;

$x--$ ; e  $--x$ ;

Elas vão ter o mesmo efeito: incrementar no primeiro caso e decrementar no segundo caso, de uma unidade.

As variações:  $++x$  e  $x++$  são chamadas de pré-incremento e pós-incremento

As variações:  $--x$  e  $x--$  são chamadas de pré-decremento e pós-decremento

Elas funcionam da seguinte maneira...suponha a expressão:

$y = x++$ ;

Acontece duas coisas, nessa ordem:

1.  $y$  recebe o valor de  $x$

2.  $x$  é incrementado em uma unidade

Já na expressão:

$y = ++x$ ;

Ocorre o seguinte:

- 1.o valor de x é incrementado em uma unidade
- 2.y recebe o novo valor de x, incrementado

Veja o seguinte programa. Ele imprime o que na tela?

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

int main()
{
    int x=1;

    cout << x++;
    return 0;
}
```

A resposta é 1. Primeiro imprime x, só depois incrementa o x.  
Coloque outro *cout* para você ver o novo valor de x depois.

Já o programa a seguir, qual saída terá?

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

int main()
{
    int x=1;

    cout << ++x;
    return 0;
}
```

Agora sim, imprime diretamente 2.  
Primeiro ocorre o incremento (++), e só depois é impresso o valor de x.

Agora vejamos outra sequência de operações:

```
x = 2;
y = 3;
```

$z = x * ++y;$

Quais os valores de x, y e z após esses três procedimentos?  
Vamos lá.

Primeiro, o y é incrementado, e passa a valer 4.

Então, temos  $z = x * y = 2 * 4 = 8$

Os novos valores são 2, 4 e 8.

Agora vejamos essa sequência de operações:

$x = 2;$

$y = 3;$

$z = x * y--;$

O valor de z é:  $z = x * y = 2 * 3 = 6$

Como o sinal de decremento veio depois da variável y, só depois ela é decrementada e passa a valer 2. Os novos valores são 2, 2 e 6.

# O laço WHILE em C++

Neste tutorial de laços e loopings, vamos apresentar a estrutura de repetição **while**, em C++.

## Laço WHILE: Estrutura de Repetição

Como o nome pode sugerir na introdução, é uma estrutura que nos permite fazer repetições de determinados trechos de código em C++.

A estrutura do C++ é a seguinte:

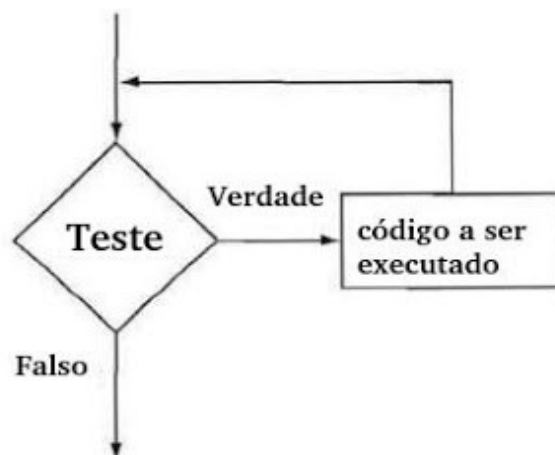
```
while(teste){  
    // Código caso o  
    // teste seja verdadeiro  
}
```

Então, começa com o comando *while*, depois parêntesis, e dentro destes deve ter algum valor booleano, geralmente é uma expressão, um teste condicional que vai retornar verdadeiro ou falso.

Enquanto (*while* em inglês) aquele valor dentro dos parêntesis for verdadeiro, o código dentro da instrução *while* vai rodar.

Cada repetição é chamada iteração. Podem existir nenhuma, uma, mil, um milhão ou infinitas iterações em um laço (*loopings* infinitos), vai depender de como se comporta o teste condicional.

Veja o fluxograma do laço WHILE:



## Exemplo de uso de **WHILE** - Estrutura de Repetição

Vamos para os exemplos, assim aprendemos melhor.

Vamos criar um programa que exibe na tela os números de 1 até 10. Para isso, vamos usar uma variável de controle, vamos chamar de *count*, pois será nosso contador.

Inicializamos com valor 1.

O teste condicional é: *enquanto count for menor ou igual a 10*

Dentro do WHILE damos o *cout* para imprimir o número.

Em seguida, temos que incrementar o contador em uma unidade, veja o código:

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

int main()
{
    int count=1;

    while(count<=10){
        cout << count << endl;
        count++;
    }
    return 0;
}
```

O que ocorre é o seguinte. O código chega no WHILE, e lá ele testa se a variável é menor ou igual a 10.

Como ela vale 1, o teste é verdadeiro e entra no código. Lá imprimimos a variável, e incrementamos o contador.

Depois, o teste ocorre novamente, agora nossa variável vale 2 e dá verdadeiro para o teste, sendo executado seu código.

Depois, o teste ocorre com o contador valendo 3 , e assim vai indo, até o contador ser 10.

Nesse caso, o teste ainda é verdadeiro.

Porém dentro do código, o contador incrementa e vira 11.

Quando for testar de novo, o resultado é falso, então a estrutura de repetição para de rodar e termina o laço.

Veja como seria o código do contrário, exibindo de 10 até 1:

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

int main()
{
    int count=10;

    while(count>0){
        cout << count << endl;
        count--;
    }
    return 0;
}
```

## Como usar laço WHILE em C++

O exemplo abaixo é um looping infinito, mas o teste dentro do laço WHILE é sempre verdadeiro, então ele vai ficar imprimindo a mensagem na tela infinitamente...

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

int main()
{
    while(1)
        cout << "Curso C++ Progressivo\n";

    return 0;
}
```

Podemos também fazer uma contagem infinita:

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

```

int main()
{
    int count=1;

    while(count++)
        cout << count << endl;

    return 0;
}

```

Note que colocamos o operador de incremento diretamente dentro do laço while.

Também não usamos chaves, isso é permitido pois o código do while tem apenas uma linha.

## WHILE em C++: Validando entradas

Uma das funcionalidades do comando WHILE é ficar validando a entrada do usuário.

Por exemplo, vamos pedir uma nota ao usuário, de 0 até 10.

Se ele digitar menor que 0 ou maior que 10, vai entrar no WHILE.

Dentro da estrutura de repetição, avisamos que ele inseriu uma nota errada e pedimos que ele insira a nota novamente.

O laço while é um laço teimoso...ele se repete quantas vezes forem necessárias.

Quer ver? Teste colocar notas menor que 0 ou maior que 10, ele só vai parar quando a nota digitada for entre 0 e 10:

```

#include <iostream>
using namespace std;

int main()
{
    int grade;

    cout << "Digite uma nota: ";
    cin >> grade;
}

```

```

while(grade<0 || grade>10){
    cout <<"Nota inválida, digite outra vez: ";
    cin >> grade;
}

return 0;
}

```

Bem simples, porém poderoso, essa estrutura de controle.

## Exemplo de uso de **WHILE** em C++

*Crie um programa que pede um número ao usuário e exibe a tabuada deste número.*

Vamos armazenar o número digitado na variável *num*. Nossa variável de controle é a *aux*, que vai percorrer de 1 até 10.

Veja como fica nosso código:

```

#include <iostream>
using namespace std;

int main()
{
    int num, aux=1;

    cout << "Digite um numero: ";
    cin >> num;

    while(aux<=10){
        cout <<num<<" x "<<aux<<" = "<<num*aux<<endl;
        aux++;
    }

    return 0;
}

```



# DO ... WHILE looping em C++

Neste **tutorial de C++**, vamos aprender a usar o looping *do...while* em C++, através de exemplos prontos com código comentado.

## O Looping DO ... WHILE

Recapitulando: *loop* é um determinado trecho de código que pode se repetir quantas vezes desejarmos. Já estudamos o [looping WHILE em C++](#), agora vamos conhecer o DO WHILE.

Lembrando que no laço WHILE, o teste condicional ocorre antes da execução do *loop*.

Essa é a principal diferença pro laço DO WHILE, neste, o teste condicional vai acontecer somente depois de cada iteração (repetição do código interno do laço).

A estrutura do looping DO WHILE é:

```
do
{
    // código
    // código
} while (condição) ;
```

Ou seja, primeiro o código é executado (DO significa faça, em inglês).

Depois, ocorre o teste condicional dentro dos parêntesis do *while*.

Se for verdade, o código é repetido novamente, e de novo e de novo, enquanto a condição for verdadeira.

Veja o fluxograma:



## Como usar **DO WHILE** em C++

A primeira coisa que você deve ter em mente ao decidir usar o DO WHILE, é que ele vai executar o código **pelo menos** uma vez! Ou seja, sempre executa uma iteração do laço!

Um uso muito comum é em *menus*. Um menu é sempre exibido pelo menos uma vez (seja num jogo ou no seu sistema bancário).

Vamos criar um menu de um banco.

Veja nosso código:

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

int main()
{
    int op;

    do
    {
        cout << "Escolha uma opção:\n";
        cout << "0. Sair\n";
        cout << "1. Saldo\n";
        cout << "2. Extrato\n";
```

```

    cout << "3. Saque\n";
    cout << "4. Transferência\n";
    cin >> op;

} while(op);

return 0;
}

```

Cada vez que você digita uma opção, ele pode ir para uma seção diferente do sistema (aprenderemos como fazer isso usando funções), e só sai se você digitar 0 (ou seja, enquanto o valor *op* for diferente de 0, o laço ocorre, pois todo valor diferente de 0 é verdadeiro, só o 0 é False).

Note que nem precisamos inicializar a variável (se fossemos usar o laço *while*, precisaríamos inicializar, para garantir que o *loop* ocorresse pelo menos uma vez).

## Exemplo de uso de DO WHILE

Usamos o looping do *while* também quando queremos fazer algo pelo menos uma vez, e queremos dar a opção de tudo se repetir novamente.

Por exemplo, o código abaixo calcula a média de duas notas que o usuário fornecer.

Ao final, ele pergunta se você deseja calcular outra média:

```

#include <iostream>
using namespace std;

int main()
{
    float grade1, grade2;
    int op;

    do
    {
        cout << "Primeira nota: ";
        cin >> grade1;
    }
}

```

```

cout << "Segunda nota: ";
cin >> grade2;

cout << "Média: " << (grade1+grade2)/2 << endl;
cout << "Calcular novamente ?\n";
cout << "1. Sim\n";
cout << "2. Não\n";
cin >> op;

} while(op!=2);

return 0;
}

```

Ou seja, a pessoa que usar esse programa poderá calcular quantas médias quiser, até infinitamente, graças ao *looping* do while.

Então, sempre que precisar rodar um código pelo menos uma vez (mas não sabe quantas vezes, depende do usuário), use o looping DO WHILE.

Por exemplo:

```

do
{
    cout << "Blá blá blá (ou digite 0 pra sair): ";
    //código
    //do seu sistema
} while(op);

```

Bacana ele, não é?

# Estrutura de Repetição FOR - Laço controlado

Finalizando a apresentação das **estruturas de repetição** de nosso Curso de C++, vamos apresentar o laço FOR, o looping controlado.

## Estrutura de repetição FOR em C++

A instrução de repetição FOR tem a seguinte sintaxe:

```
for(inicialização ; teste_condicional ; atualização){  
    // código que executa enquanto  
    // o teste condicional  
    // for verdadeiro  
}
```

Vamos lá.

A estrutura de repetição FOR tem três expressões dentro dela, separadas por ponto e vírgula.

O laço começa com algum tipo de inicialização, geralmente uma variável de contagem, com algum valor inicial.

Após essa inicialização, ocorre o teste condicional. Se for verdadeiro, o código dentro das chaves do laço FOR é executado.

Após cada iteração, ocorre a 'atualização', onde geralmente atualizamos o valor do contador, muito comumente é uma variável que vai se incrementar ou decrementar.

Então, novamente o teste condicional é realizado, em caso de ser verdadeiro, novamente o código do FOR é executado. Após essa iteração, ocorre mais uma vez a atualização.

## Exemplo de uso do laço **FOR**

Em programação, um exemplo vale mais que mil palavras.

Vamos contar de 1 até 10, usando o laço for, o código é o seguinte:

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

int main()
{
    int count;

    for(count=1; count<=10 ; count++){
        cout << count << endl;
    }

    return 0;
}
```

A nossa variável de controle é a *count*, que vai iniciar valendo 1. Vamos *printar* ela na tela enquanto seu valor for menor ou igual a 10. A cada iteração, incrementamos ela em uma unidade (count++), pois queremos que ela vá de 1 até 10.

Note que já fazíamos isso no laço WHILE, mas inicializávamos as variáveis antes do laço, dentro dele a gente atualizava a variável a cada loop e ocorria o teste condicional dentro dos parêntesis do WHILE.

Ocorre a mesma maneira no laço FOR, mas de maneira mais organizado.

## Como usar a estrutura de repetição **FOR**

Vamos fazer o contrário agora, uma contagem regressiva, que conta de 100 até 1.

Para isso, inicializamos nossa variável de controle como 100.

O teste a ser realizado é: `count > 0`

Ou seja, enquanto a variável tiver um valor acima de 0, as iterações do laço FOR vão ocorrer.

E a cada iteração temos que decrementar o *count*, pois ele vai de 100 pra 1, de um em um.

Veja como ficou nosso código:

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

int main()
{
    int count;

    for(count=100; count>0 ; count--){
        cout << count << endl;
    }

    return 0;
}
```

Nem sempre, porém, vamos incrementar ou decrementar de 1 em 1. Podemos atualizar nossas variáveis da maneira que quisermos.

Por exemplo, vamos imprimir na tela todos os números pares de 1 até mil. O primeiro par é o 2, então inicializamos nossa variável com esse valor. Vamos incrementar de 2 em 2: `count += 2`  
E o teste é enquanto a variável for menor ou igual a mil: `count <= 1000`

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

int main()
{
    int count;

    for(count=2; count<=1000 ; count+=2){
        cout << count << endl;
    }

    return 0;
}
```

Veja a incrível velocidade com que esse código é executado.  
Duvida da capacidade do C++? Coloque 1 milhão.

## Quando usar o laço FOR

Muitas vezes, queremos fazer loopings com determinado número de iterações.

Por exemplo, ao pedir as notas de um aluno, vamos pedir o tanto de matérias que existem, para calcular a média.

Para calcular seu imposto de renda, precisamos somar todo seu salário do ano, ou seja, 13 salários (tem o décimo terceiro).

O laço FOR é ideal quando você sabe exatamente o número de iterações que vai fazer: "*ah, quero calcular isso nesse intervalo de x até y*", então pimba, use a estrutura de repetição FOR.

Quando não sabe quando o looping deve terminar ou tem menos controle sobre quantas iterações devem ocorrer, aí use o laço WHILE.

Lembrando que, no fundo, eles são absolutamente a mesma coisa. Só vai ser mais fácil trabalhar com FOR algumas vezes e com WHILE em outras ocasiões.

## Estrutura de Repetição FOR

*Crie um programa que pede quantas notas você quer calcular a média, em seguida pede cada uma dessas notas e por fim exibe a média.*

Vamos armazenar o número de notas que vamos pedir na variável *n*. O próximo passo é pedir nota por nota, e aqui vem o pulo do gato: calcular a soma de todas as notas.

Dentro do FOR, a variável de controle *aux* vai da nota 1 até a nota *n*, pedindo uma por uma e armazenando essa nota na variável *grade*.

Vamos armazenar na variável *sum*, a soma de todas essas notas.



Por fim, exibimos  $sum/n$  para exibir a média.  
Veja nosso código C++:

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

int main()
{
    int aux, n;
    float grade, sum=0;

    cout <<"Quantas matérias: ";
    cin >> n;

    for(aux=1; aux<=n ; aux++){
        cout <<"Nota "<<aux<<": ";
        cin >> grade;
        sum += grade;
    }

    cout << "Média: "<<(sum/n)<<endl;

    return 0;
}
```

Note como essa estrutura de repetição é controlada: ela vai sempre rodar 'n' iterações, seja lá qual o valor de 'n' (obviamente, o número de notas deve ser um valor inteiro positivo).

Pode preencher com 2 notas, 3 notas, mil notas, um milhão de notas...  
Poderoso, esse laço FOR não é?

# Como fazer tabuada com os laços

## Tabuada em C++ com FOR

Primeiro, pedimos ao usuário um número e armazenamos na variável *num*. Vamos usar também uma variável de controle *aux*.

Essa variável, dentro do laço FOR, vai de 1 até 10, pra montarmos a tabuada.

Em seguida, é só multiplicar *num* por *aux*, em cada iteração e exibir o resultado.

Veja como ficou nosso código:

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

int main()
{
    int num, aux;

    cout << "Tabuada do numero: ";
    cin >> num;

    for(aux=1 ; aux<=10 ; aux++)
        cout<<num<<" * "<<aux<<" = " << num*aux <<endl;

    return 0;
}
```

## Tabuada em C++ com WHILE e DO WHILE

Também é possível fazer o mesmo com o looping WHILE, veja:

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

int main()
{
    int num, aux=1;
```

```

cout << "Tabuada do numero: ";
cin >> num;

while(aux<=10){
    cout<<num<<" * "<<aux<<" = " << num*aux <<endl;
    aux++;
}

return 0;
}

```

Note que temos que inicializar antes a variável *aux* e incrementar ela dentro do WHILE, igual como fazemos no cabeçalho da estrutura FOR.

Podemos também incrementar nosso código e usar do while, para ficar exibindo quantas tabuadas o usuário quiser, só para quando ele digitar 0:

```

#include <iostream>

using namespace std;

int main()
{
    int num, aux;

    do{
        cout << "Tabuada do numero: ";
        cin >> num;

        for(aux=1; aux<=10 ; aux++)
            cout<<num<<" * "<<aux<<" = " << num*aux <<endl;
        cout<<endl;
    }while(num);

    return 0;
}

```

# Somatório e Fatorial com laços

Neste tutorial, vamos resolver duas questões de nossa [lista de exercícios de laços](#), vamos aprender como calcular o somatório e fatorial de um número, usando apenas laços FOR ou WHILE, em C++.

## Somatório usando laços em C++

O somatório de um número  $n$  nada mais é que a soma dos números de 1 até  $n$ .

Então, primeiro pedimos ao usuário um inteiro positivo e armazenamos na variável  $n$ .

Vamos usar também uma variável auxiliar  $aux$ , que vai percorrer os valores de 1 até  $n$ , dentro do looping.

Também vamos usar a variável  $sum$ , que vai armazenar a soma de todos esses números. Obviamente, devemos inicializar ela com valor 0.

Veja como fica nosso código usando laço FOR:

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

int main()
{
    int n, aux, sum=0;

    cout << "Somatório de: ";
    cin >> n;

    for(aux=1 ; aux<=n ; aux++)
        sum += aux;

    cout << "Somatório: " << sum << endl;

    return 0;
}
```

Agora com laço WHILE:

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

int main()
{
    int n, aux=1, sum=0;

    cout << "Somatório de: ";
    cin >> n;

    while(aux<=n){
        sum += aux;
        aux++;
    }

    cout << "Somatório: " << sum << endl;

    return 0;
}
```

Com looping DO WHILE, podendo ser calculada várias vezes e digitando 0 pra terminar o laço:

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

int main()
{
    int n, aux, sum;

    do{
        cout << "Somatório de: ";
        cin >> n;
        sum = 0;

        for(aux=1 ; aux<=n ; aux++)
            sum += aux;

        cout << "Somatório: " << sum << endl;
        cout<<endl;
    }
```

```
    }while(n);  
  
    return 0;  
}
```

## Fatorial usando loopings em C++

Se o somatório soma todos os números de 1 até n, o fatorial multiplica todos os números de 1 até n.

O símbolo do fatorial de um número é !.

Por exemplo:

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Em vez de *sum* vamos usar *prod* pra armazenar o produto.

E ao invés de somar (+=), vamos multiplicar (\*=).

Usando laço FOR:

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int n, aux, prod=1;  
  
    cout << "Fatorial de: ";  
    cin >> n;  
  
    for(aux=1 ; aux<=n ; aux++)  
        prod *= aux;  
  
    cout << "Fatorial: " << prod << endl;  
  
    return 0;  
}
```

WHILE:

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

int main()
{
    int n, aux=1, prod=1;
    cout << "Fatorial de: ";
    cin >> n;

    while(aux<=n){
        prod *= aux;
        aux++;
    }

    cout << "Fatorial: " << prod << endl;

    return 0;
}
```

DO WHILE:

```
#include <iostream>
using namespace std;
int main()
{
    int n, aux, prod;

    do{
        cout << "Fatorial de: ";
        cin >> n;
        prod = 1;

        for(aux=1 ; aux<=n ; aux++)
            prod *= aux;

        cout << "Fatorial: " << prod << endl;
        cout<<endl;
    }while(n);

    return 0;
}
```

Simples, né?

# Exponenciação usando laços em C++

Neste tutorial de nosso **curso de C++**, vamos aprender como usar laços para criar a operação matemática de exponenciação.

## Exponenciação na Matemática

Chamamos de exponenciação, a operação matemática que tem dois números: a base e o expoente.

É o famoso 'x elevado a y'.

Por exemplo:

$3^2$  (3 elevado a 2, ou 3 elevado ao quadrado) - 3 é base e 2 o expoente

$4^3$  (4 elevado a 3, ou 4 ao cubo) - 4 é a base e 3 o expoente.

Agora calculando essas operações:

$$3^2 = 3 * 3 = 9$$

$$4^3 = 4 * 4 * 4 = 64$$

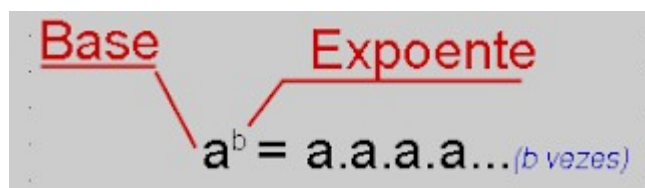
$$5^4 = 5 * 5 * 5 * 5 = 625$$

Note como o valor da base se repete o tanto de vezes o valor do expoente...se repete...huuum...repetição...lembra o quê? Huuum, laços!

## Exponenciação com laços no C++

Se temos um valor:

$a^b$  (a elevado a b), isso significa que o valor de  $a$  vai se repetir  $b$  vezes:


$$a^b = a.a.a.a...(b \text{ vezes})$$

Vamos pedir ao usuário as variáveis *base* e *expo*.

O resultado, vamos armazenar em *res*, esse valor é inicializado com 1, pois vamos fazer uma série de multiplicações nessa variável.



Dentro do laço FOR, temos uma variável auxiliar *aux* que vai contar de 1 até *expo*, para realizar *expo* repetições, concorda?

Veja como fica nosso código usando laço FOR:

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

int main()
{
    int base, expo, res=1, aux;

    cout << "Base: ";
    cin >> base;

    cout << "Expoente: ";
    cin >> expo;

    for(aux=1 ; aux<=expo ; aux++)
        res *= base;

    cout << base << "^" << expo << " = " << res << endl;

    return 0;
}
```

Agora com laço WHILE:

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

int main()
{
    int base, expo, res=1, aux=1;

    cout << "Base: ";
    cin >> base;

    cout << "Expoente: ";
    cin >> expo;

    while(aux<=expo){
```

```

        res *= base;
        aux++;
    }

    cout << base << "^" << expo << " = " << res << endl;

    return 0;
}

```

E por fim, com a estrutura de repetição DO WHILE (basta digitar 0 e/ou 0 para encerrar os cálculos):

```

#include <iostream>
using namespace std;

int main()
{
    int base, expo, res, aux;

    do{
        cout << "Base: ";
        cin >> base;

        cout << "Expoente: ";
        cin >> expo;

        res=1;
        for(aux=1 ; aux<=expo ; aux++)
            res *= base;

        cout << base << "^" << expo << " = " << res << endl;
        cout << endl;
    }while(base || expo);

    return 0;
}

```

Obviamente, esse algoritmo funciona apenas para expoentes inteiros, para decimais, o buraco é mais embaixo.

Mas vejam que bacana, a utilidade e versalidade das estruturas de repetição, nossos amados laços, servem até para fazer operações matemáticas.

# Laços aninhados em C++ - Laço dentro de laço

Agora que já treinamos bastante o conceito de laços e loopings, aprendendo a fazer, vamos aprender outro conceito importante, o de laços aninhados, ou laços dentro de laços.

## Estruturas de Repetição aninhadas em C++

Veja nos tutoriais anteriores, onde usamos exercícios famosos para fixar nossos conhecimentos em estruturas de repetição, especificamente, nos códigos do laço DO WHILE:

- Tabuada
- Somatório e fatorial
- Sequência de Fibonacci
- Exponenciação

Colocamos um laço FOR dentro do laço DO WHILE.

Ou seja, existe um looping maior, um laço pai, o DO WHILE, que vai ficar repetindo uma estrutura base.

Dentro de cada iteração dessa, um laço FOR é executado.

Esse laço de dentro está *aninhado* ao laço de fora.

Consegue ver e entender sua função, por que ele funciona?

É bem simples, mas é uma técnica muito poderosa.

## Usando laços aninhados em C++

Vamos usar a técnica das estruturas de repetição aninhadas para imprimir um tabuleiro N x N, ou seja, de N linhas e N colunas. Veja um exemplo de tabuleiro 3x3, pra jogar jogo da velha:

```
- - -  
- - -  
- - -
```

O segredo aqui é usar um laço pras linhas e outro, interno ao primeiro, para cuidar das colunas em cada linha.

O primeiro FOR vai de 1 até N, é o responsável pelas linhas.  
Note que ao final dele tem uma quebra de linha, para imprimir a próxima iteração na linha de baixo.

Dentro do FOR de linha, vamos imprimir N tracinhos.

Veja como fica nosso código:

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

int main()
{
    int lin, col, N;

    cout<<"Tamanho do tabuleiro N x N: ";
    cin >> N;

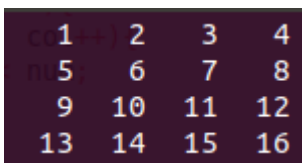
    for(lin=1 ; lin<=N ; lin++){
        for(col=1 ; col<=N ; col++){
            cout<<" - ";

            cout <<endl;
        }

        return 0;
    }
```

## Como usar laços aninhados em C++

Vamos criar um programa para imprimir a seguinte tabela da imagem:



|    |    |    |    |
|----|----|----|----|
| 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  |
| 9  | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

Veja que tabela bonita, que tabela formosa, que tabela bem feita.

Vamos aprender como fazer ela?

Primeiro, note duas duas, ela tem:

- 4 linhas
- 4 colunas

Vamos usar dois laços, um para cuidar das linhas e outro para cuidar das colunas.

Para controlar esses laços, vamos usar as variáveis *lin* e *col*, bem como uma variável que vai receber os números de 1 até 16, a *num*.

Vamos, de início, trabalhar nas linhas:

```
for(lin=1 ; lin<=4 ; lin++){  
    // alguma coisa  
    cout <<endl;  
}
```

Note que ao término de cada iteração, temos que dar uma quebra de linha, para ir para a linha de baixo.

Dentro de cada iteração desta, precisamos imprimir as colunas, que são 4 por linhas.

Então vamos usar outro laço aninhado:

```
for(lin=1 ; lin<=4 ; lin++){  
    for(col=1 ; col<=4 ; col++){  
        // alguma coisa  
    }  
    cout <<endl;  
}
```

Dentro desse for aninhado, temos imprimir os números.

Como a variável *col* vai de 1 até 4, ele imprime quatro números em cada linha.

Vamos usar a variável *num*, inicialmente com valor 1, para imprimir esses valores.

Após a iteração interna, temos que incrementar essa variável.

Para a tabela sair bonitinha e formatada, vamos usar o comando **setw**, para definir como tamanho de 4 espaçamentos (você precisa incluir a biblioteca *iomanip*, pra usar esse comando).

Veja como ficou nosso código:

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

int main()
{
    int lin, col, num=1;

    for(lin=1 ; lin<=4 ; lin++){
        for(col=1 ; col<=4 ; col++){
            cout<<setw(4) << num;
            num++;
        }
        cout <<endl;
    }

    return 0;
}
```

Bacana, né?

## Exemplo de uso de laços aninhados

Vamos agora imprimir um triângulo de asteriscos, do tamanho que o usuário quiser.

Por exemplo, um triângulo de tamanho 5:

```
*  
**  
***  
****  
*****
```

Note que vai ter 5 linhas.

Na primeira linha, tem 1 coluna.

Na segunda linha, tem 2 colunas.

...

Na quinta linha, tem 5 colunas.

O primeiro FOR, a variável de controle *lin* vai de 1 até N, onde N é o tamanho do triângulo que o usuário quiser.

Dentro de cada FOR, vamos usar outro FOR para imprimir os asteriscos.

Eles devem imprimir de 1

até *lin*, consegue captar essa ideia?

Na primeira linha, imprime 1 asterisco.

Na segunda linha, imprime 2 asteriscos.

...

Na N-ésima linha, imprime N asteriscos.

Veja como fica nosso código:

```
#include <iostream>  
#include <iomanip>  
using namespace std;  
  
int main()  
{  
    int lin, col, N;  
  
    cout<<"Tamanho do triângulo: ";  
    cin >> N;  
  
    for(lin=1 ; lin<=N ; lin++){  
        for(col=1 ; col<=lin ; col++){  
            cout<<"*";  
        }  
    }  
}
```

```

        cout <<endl;
    }

    return 0;
}

```

Note que o FOR de fora, usa chaves, pois tem mais de uma linha de comando abaixo dele.

Já o FOR de dentro, não precisa, pois ele só tem uma linha de comando. Cuidado para não confundir as chaves! Faça sempre um espaçamento e uma indentação correta, para não errar.

## Laço dentro de laço de laço dentro de laço...

Ok, já vimos como aninhar estruturas de repetição, colocamos um FOR dentro de outro.

E que tal agora colocar mais um FOR, aninhado aos outros dois?

Vamos mostrar todos os horários possíveis de um relógio, durante o dia, no formato:

hour:min:sec

Ou seja, vai de : 00:00:00

Até o : 23:59:59

Vamos usar três variáveis: h, m e s

h vai de 0 até 23, tanto m como s vão de 0 até 59, pois representam os minutos e segundos.

Nosso código fica assim:

```

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

int main()
{

```



```

cout.fill('0');
for(int h=0 ; h < 24 ; h++)
    for(int m=0 ; m < 60 ; m++)
        for(int s=0; s<60 ; s++){
            cout<<setw(2)<<h<<":";
            cout<<setw(2)<<m<<":";
            cout<<setw(2)<<s<<endl;
        }

    return 0;
}

```

Usamos o `setw(2)` para definir 2 caracteres de espaçamento. Quando não tiver nada pra aparecer nesse espaçamento, vai aparecer 0, pois usamos `cout.fill('0')`

# Mega-Sena com C++

Neste **tutorial de C++**, vamos aprender como contar todas os possíveis palpites da Mega-Sena, bem como vamos ver como exibir todos esses números, usando a técnica de estruturas de repetição aninhadas.

## A loteria da Mega-Sena

Muito provavelmente você já jogou na Mega Sena, não é?

Ela funciona assim: você deve escolher 6 dezenas (sena), de um universo de 60 números, de 1 até 60.

No sorteio, tem um globo com 60 bolas e as moças bonitas lá vão tirando bolinha por bolinha...então, obviamente, as dezenas não se repetem, concorda?

No final, eles exibem o resultado na ordem crescente dos valores, ou seja, da dezena menor pra maior.

O 'menor' palpite é:

1 2 3 4 5 6

Já o 'maior' palpite é:

55 56 57 58 59 60

Aqui vem o segredo:

- 1.A primeira dezena vai de 1 até 55
- 2.A segunda dezena vai de 2 até 56
- 3.A terceira dezena vai de 3 até 57
- 4.A quarta dezena vai de 4 até 58
- 5.A quinta dezena vai de 5 até 59
- 6.A sexta dezena vai de 6 até 60

# Quantos palpites são possíveis na Mega Sena

Então, vamos lá.

Vamos usar 6 variáveis para as dezenas: dez1, dez2, dez3, dez4, dez5 e a dez6.

A variável acumuladora, para contar quantas iterações (consequentemente, quantos palpites são possíveis na Mega Sena), é a *sum*.

Agora basta fazer FOR aninhado com FOR e contar quantas possibilidades existem, sempre tendo cuidado com o intervalo que cada dezena pode assumir.

Outro segredo, importante, é que a variável dez1 começa do 1, e as seguintes começam a partir da dezena anterior somado de 1, pois as dezenas são maiores que as outras, já que estamos assumindo que estejam em ordem crescente.

O código:

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

int main()
{
    int dez1, dez2, dez3, dez4,
        dez5, dez6, sum=0;

    for(dez1=1; dez1<=55 ; dez1++)
        for(dez2=dez1+1; dez2<=56 ; dez2++)
            for(dez3=dez2+1; dez3<=57 ; dez3++)
                for(dez4=dez3+1; dez4<=58 ; dez4++)
                    for(dez5=dez4+1; dez5<=59 ; dez5++)
                        for(dez6=dez5+1; dez6<=60 ; dez6++)
                            sum++;
    cout << "Total : " << sum << endl;

    return 0;
}
```

E o resultado é:

```
Total : 50063860
Process returned 0 (0x0)   execution time : 0.167 s
```

Se ainda se lembrar das aulas de análise combinatória, basta calcular (60 seis a seis).

Aqui levou 0.167s pra rodar mais de 50 milhões de iterações, e aí na sua máquina?

## Exibindo todos os palpites da Mega-Sena

Agora vamos imprimir na tela todos os possíveis resultados:

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

int main()
{
    int dez1, dez2, dez3, dez4,
        dez5, dez6;

    for(dez1=1; dez1<=55 ; dez1++)
        for(dez2=dez1+1; dez2<=56 ; dez2++)
            for(dez3=dez2+1; dez3<=57 ; dez3++)
                for(dez4=dez3+1; dez4<=58 ; dez4++)
                    for(dez5=dez4+1; dez5<=59 ; dez5++)
                        for(dez6=dez5+1; dez6<=60 ; dez6++)
                            cout<<dez1<<"-"<<dez2<<"-"<<dez3<<"-"
                                <<dez4<<"-"<<dez5<<"-"<<dez6<<endl;

    return 0;
}
```

Note que agora é beeeem mais demorado, e isso se deve ao fato da função *cout* ser mais lenta, demora pra exibir as coisas na sua tela, se a máquina ficasse só fazendo os cálculos, como no exemplo anterior, seria bem mais rápido. Mas aqui temos que mostrar os resultados das iterações, então a coisa é mais morosa mesmo.

# As instruções **BREAK** e **CONTINUE** do C++

Neste tutorial, vamos aprender duas instruções, ou comandos, muito importantes: o **break** e o **continue**, em C++.

## Instrução **BREAK** em C++

Se você fez direitinho e na ordem, o **curso C++ Progressivo**, já deve conhecer este comando break, pois usamos no tutorial:

[Switch, case e break](#)

Lá, quando esse comando era executado, ele simplesmente encerrava o teste condicional SWITCH.

Aqui, ele faz a mesma coisa, mas no caso ele serve para interromper um laço, a qualquer instante.

No código abaixo, o C++ fica pedindo um número ao usuário e calculando seu quadrado.

Se em algum momento o usuário digitar 0, o IF se torna verdadeiro e o comando break é acionado, interrompendo sumariamente o laço WHILE:

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

int main()
{
    int num;

    while(true){
        cout<<"Numero: ";
        cin >>num;

        if(num==0)
            break;

        cout<<num*num<<endl;
    }

    return 0;
}
```

Ou seja, quando queremos, em algum momento, interromper um laço, usamos a instrução `BREAK`, que geralmente vem dentro de um teste condicional, dentro de algum looping.

Se tivermos um laço dentro de outro, e dentro deste estiver uma instrução `BREAK`, somente esse laço mais interno que será interrompido, ok?

O seguinte código fica pedindo notas para o usuário, para calcular a média. Caso o usuário digite uma nota que não é válida (abaixo de 0 ou acima de 10), o laço `WHILE` é encerrado e fornecemos a média dos números digitados dessa nota inválida:

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

int main()
{
    int aux=0;
    float num, sum=0;

    while(true){
        cout<<"Nota: ";
        cin >> num;

        if(num<0 || num>10)
            break;

        sum+=num;
        aux++;
    }

    cout<<"Média: "<<(sum/aux)<<endl;
    return 0;
}
```

Note que garantimos que os números fornecidos sejam corretos (entre 0 e 10), pois caso contrário o laço é encerrado.

## O comando **CONTINUE** em C++

Se o BREAK encerra o teste condicional ou estrutura de controle, o comando CONTINUE encerra **apenas a iteração**.

Ou seja, pula pra próxima iteração.

Vamos somar todos os números de 1 até 100, exceto os múltiplos de 4:

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

int main()
{
    int num, sum=0;

    for(num=1; num<=100 ; num++){
        if(num%4==0)
            continue;
        sum += num;
    }

    cout<<"Total: "<<sum<<endl;
    return 0;
}
```

Dentro do laço verificamos se o número é divisível por 4, se for, essa iteração é pulada, não rodando o código seguinte ao CONTINUE, vai pra próxima iteração normalmente.

Assim como o BREAK, o comando CONTINUE geralmente ocorre sob determinado teste condicional, quando você quer excluir uma iteração específica do looping.

Vamos refazer o código que calcula a média.

Agora, ao invés de parar a execução, ele pula a iteração, não somando aquela nota inválida na soma:

```

#include <iostream>
using namespace std;

int main()
{
    int aux=0;
    float num, sum=0;

    while(true){
        cout<<"Nota: ";
        cin >> num;

        if(num>10){
            cout<<"Nota acima de 10 são inválidas"<<endl;
            continue;
        }
        if(num<0){
            cout<<"Nota negativa, encerrando cálculos e exibindo a média: ";
            break;
        }

        sum+=num;
        aux++;
    }

    cout<<(sum/aux)<<endl;
    return 0;
}

```

Note que a média é calculada apenas para números válidos. Para encerrar o looping, você precisa digitar um valor negativo.



# Números primos em C++ - Como descobrir

Neste tutorial, vamos destrinchar os primos. Vamos aprender como verificar se um número é primo ou não, bem como exibir uma lista de quantos primos quisermos, usando laços e loopings, em C++.

## Números Primos na Matemática

Um número é dito ser primo quando pode ser dividido somente por 1 e por ele mesmo.

O 7 é primo, você pode dividir só por 1 e por 7, por qualquer outro valor vai dar um resultado quebrado.

O 12 não é primo, pois pode ser dividido por 1, 2, 3, 4, 6 e 12.

Vejamos alguns números primos: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997...

Os primos são uma classe muuuuito especial na Matemática, tendo utilidade em diversos ramos e áreas, como na criptografia.

Vale a pena pesquisar sobre eles, há todo um véu de mistério neles, pois tentam há milênios encontrar uma fórmula para gerar números primos e nada até hoje deu certo.

Será que um dia você consegue? Quem sabe...se ganhar o prêmio Nobel, não esquece de compartilhar a grana com a gente...

Simbora caçar uns primos?

## Como Descobrir se um número é primo

Para verificar se um número *num* é primo, basta verificar seus divisores, de 1 até *num*.

Por exemplo, vamos testar se o 9 é primo. Basta analisar o resto da divisão por 1, 2, 3, 4, 5, 6, 7, 8, e 9.

Se for primo, somente vai ser divisível por 1 e por ele mesmo, logo vai ter 2 divisores. Mas o resto da divisão vai dar 0 quando fizermos  $9\%3$ , logo 3 também é divisor, totalizando 3 divisores, logo não é primo.

Agora o 11.

Podemos pegar o resto da divisão por 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 e 11, que só vai dar 0 pra 1 e pra 11.

Logo, ele é primo.

Ou seja, basta fazer o resto da divisão de *num* por 1, 2, 3, 4, 5, ..., até *num*, e contar na variável *div* (inicializada com 0), quantos divisores tem.

Se for 2, é primo.

Se for mais que 2, não é primo.

Veja o código:

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

int main()
{
    int aux, num=479001599, div=0;

    for(aux=1 ; aux<=num ; aux++)
        if(num%aux==0)
            div++;

    if(div==2)
        cout<<"É primo"<<endl;
```

```

else
    cout<<"Não é primo"<<endl;
return 0;
}

```

Testamos com um número primo gigante, o 479001599.  
Aqui levou 1.831s pra verificar, e na sua máquina?

## Otimizando a busca por primos

Ora, todo número é divisível por 1.

Então não precisamos fazer o resto da divisão por 1, já é uma checagem a menos.

E também não precisamos testar até *num*.

Passou da metade, não vai ter mais nenhum divisor possível.

Dando uma enxugada no código, ele fica assim:

```

#include <iostream>
using namespace std;

int main()
{
    int aux, num=479001599, div=0;

    for(aux=2 ; aux<=num/2 ; aux++)
        if(num%aux==0)
            div++;

    if(div==0)
        cout<<"É primo"<<endl;
    else
        cout<<"Não é primo"<<endl;
    return 0;
}

```

Agora levou só 1.012s

E na sua máquina?

Podemos ir mais além e calcular até a raiz quadrada de *num*, ao invés de apenas até *num/2* (pesquise o motivo disso):

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

int main()
{
    int aux, num=479001599, div=0;

    for(aux=2 ; aux<=sqrt(num) ; aux++)
        if(num%aux==0)
            div++;

    if(div==0)
        cout<<"É primo"<<endl;
    else
        cout<<"Não é primo"<<endl;
    return 0;
}
```

0.004s ! Carai, maluco!

## Achando primos num intervalo

Agora vamos imprimir todos os primos num determinado intervalo, de 1 até um valor *Máximo*, como 100.

Primeiro, uma variável pra testar todos os números de 2 até Max, é a *aux*.

Para cada número, vamos contar todos os divisores e armazenar na variável *div*, por isso ela deve começar zerada dentro do primeiro laço.

No segundo FOR, vamos verificar cada número *aux*, fazendo o resto da divisão deles por 2 até raiz quadrada do número, para verificar se tem divisores.

Se tiver, cai no IF (IF dentro de um FOR que está dentro de outro FOR, que loucura!), que incrementa *div*.

Após toda essa verificação interna, ele é primo se *div* tiver 0 como valor, se tiver, então imprimimos o valor de *aux*, pois é um primo.

Veja como fica o código:

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

int main()
{
    int aux, coun, Max=100, div;

    for(aux=2 ; aux<=Max ; aux++){
        div=0;

        for(coun=2 ; coun<=sqrt(aux) ; coun++)
            if(aux%coun==0)
                div++;

        if(!div)
            cout<<aux<<" ";
    }

    cout<<endl;

    return 0;
}
```

Teste com mil, 10 mil, 1 milhão...só agora você tem real noção do poder e capacidade de calcular que tem sua máquina

## Exercícios de Laços e Loopings

Parabéns por ter concluído a seção de [Estruturas de Repetição em C++](#), famosos laços ou loopings, um dos assuntos mais importantes de toda e qualquer linguagem de programação.

Agora chegou a hora de colocar em prática seus conhecimentos, é agora que você vai mais evoluir, vai se tornar um programador de verdade.

Resolva os exercícios, tente, se esforce, tente de novo e de novo, antes de ver a solução, combinado?

Só ler ou só assistir vídeos não vão te tornar **nunca** um programador, mesmo o mais ruinzinho.

É na raça, tentando, se esforçando, quebrando a cabeça, varando noites e chorando em posição fetal que se forja um verdade programador do curso C++ Progressivo.

Partiu? Dê logo adeus aos amigos, namorada(o), família, redes sociais...hora de se esconder da sociedade e fazer exercícios!

Ah...e vai postando aí nos comentários suas soluções!

## Exercícios de WHILE, DO WHILE e FOR em C++

0. Faça um programa em C++ que peça um inteiro ao usuário, e exiba sua tabuada.

Resolvido previamente

1. Faça um programa que receba dois números inteiros e gere os números inteiros que estão no intervalo compreendido por eles.

2. Faça um programa que peça uma nota, entre zero e dez. Mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido.

3. Faça um programa que imprima na tela os números de 1 a 20, um abaixo do outro. Depois modifique o programa para que ele mostre os números um ao lado do outro.

4. Escreva programas que exibam os seguintes padrões na tela, de acordo com o número que o usuário fornecer, que será sempre o número de linhas:

4.1

```
*  
**  
***  
****  
*****
```

4.2

```
1  
12  
123  
1234  
12345
```

4.3

```
1  
22  
333  
4444  
55555
```

4.4

```
1  
2 3  
4 5 6  
7 8 9 10
```

4.5

```
  1  
 2 3  
4 5 6  
7 8 9 10
```

4.6

```
      *
     **
    ***
   ****
  *****
```

4.7

```
1
01
101
0101
10101
```

4.8

```
      *
     ***
    *****
   *****
  *****
 *****
*****
 *****
  *****
   *****
    *****
     ***
      *
```

4.9

```
12345
2345
345
45
5
```

4.10

```
12345
1234
123
12
1
```



4.11

5 4 3 2 1

4 3 2 1

3 2 1

2 1

1

4.12

1

21

321

4321

54321

4.13

1234567654321

12345654321

123454321

1234321

12321

121

1

5. Faça um programa que leia 5 números e informe o maior número.

6. Faça um programa que leia 5 números e informe a soma e a média dos números.

7. Faça um programa que calcule o mostre a média aritmética de N notas.

8. Faça um programa que imprima na tela apenas os números ímpares entre 1 e 50. Ao final, mostre também a soma dos números.

9. Crie um programa que pede um número ao usuário e calcule o somatório até aquele valor.

10. Crie um programa que pede um número ao usuário, e calcula seu fatorial.

Ex.:  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

Resolvido previamente

11. A série de Fibonacci é formada pela sequência 0,1,1,2,3,5,8,13,21,34,55,... Onde o próximo termo é sempre a soma dos dois anteriores. Faça um programa capaz de gerar a série até o n-ésimo termo, que o usuário deverá fornecer.  
Resolvido previamente
12. Faça um programa que peça dois números, base e expoente, calcule e mostre o primeiro número elevado ao segundo número. Não utilize a função de potência da linguagem.  
Resolvido previamente
13. Faça um programa que peça um número inteiro e determine se ele é ou não um número primo. Um número primo é aquele que é divisível somente por ele mesmo e por 1.
14. Faça um programa que mostre todos os primos entre 1 e N sendo N um número inteiro fornecido pelo usuário.
15. Faça um programa que exibe todas as combinações de jogos possíveis da Mega-Sena.
16. Programe um software que recebe um número do usuário e diga se ele é um número perfeito ou não. Pesquise no Google o que é um número perfeito.
17. Programe um software que recebe dois números inteiros do usuário, e diga qual o MDC, máximo divisor comum desses números.
18. Programe um software que recebe um número menor que 1000, e diga qual o valor da unidade, da dezena e da centena.
19. Programe um software que calcula a soma dos dígitos de um número.
20. Faça um programa que mostre os n termos da Série a seguir:  
$$S = 1/1 + 2/3 + 3/5 + 4/7 + 5/9 + \dots + n/m.$$
  
Imprima no final a soma da série.

21. Seja a série harmônica  $H = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/N$ , Faça um programa que calcule o valor de H com N termos, onde N é fornecido pelo usuário.

22. Faça um programa que mostre os n termos da Série a seguir:

$$S = 1/1 + 2/3 + 3/5 + 4/7 + 5/9 + \dots + n/m.$$

Imprima no final a soma da série.

23. O valor de PI pode ser aproximado pela seguinte sequência infinita:

$$\begin{aligned}\pi &= 4 \left( \sum_{k=1}^{\infty} \frac{(-1)^{(k+1)}}{(2k-1)} \right) \\ &= 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right)\end{aligned}$$

Crie um programa que calcula o valor dessa série com 10 termos, depois com 100 termos e por fim, usando mil termos. Que valores obteve ?

# Soluções

4.

4.1

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

int main()
{
    int lin, col, N;

    cout<<"Tamanho do triângulo: ";
    cin >> N;

    for(lin=1 ; lin<=N ; lin++){
        for(col=1 ; col<=lin ; col++)
            cout<<"*";

        cout <<endl;
    }

    return 0;
}
```

4.2

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

int main()
{
    int lin, col, N;

    cout<<"Tamanho: ";
    cin >> N;

    for(lin=1 ; lin<=N ; lin++){
        for(col=1 ; col<=lin ; col++)
            cout<<col;

        cout <<endl;
    }
}
```

```
    return 0;
}
```

4.3

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

```
int main()
{
    int lin, col, N;

    cout<<"Tamanho: ";
    cin >> N;

    for(lin=1 ; lin<=N ; lin++){
        for(col=1 ; col<=lin ; col++){
            cout<<lin;

            cout <<endl;
        }

        return 0;
    }
```

4.4

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

```
int main()
{
    int lin, col, num=1;

    for(lin=1 ; lin<=4 ; lin++){
        for(col=1 ; col<=lin ; col++){
            cout << setw(3) << num;
            num++;
        }

        cout <<endl;
```

```

    }

    return 0;
}

```

4.5

```

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

int main()
{
    int lin, col, num=1, space=3;

    for(lin=1 ; lin<=5 ; lin++){
        for(col=space ; col>=0 ; col--){
            cout<<" ";

            for(col=1 ; col<=lin ; col++){
                cout << setw(3) << num;
                num++;
            }

            space--;
        }

        cout <<endl;
    }

    return 0;
}

```

4.6

```

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

int main()
{
    int lin, col, num=1, space=3;

    for(lin=1 ; lin<=5 ; lin++){
        for(col=space ; col>=0 ; col--){

```

```

        cout<<" ";

        for(col=1 ; col<=lin ; col++){
            cout << setw(2) << "*";
        }
        space--;

        cout <<endl;
    }

    return 0;
}

```

4.7

```

#include <iostream>
using namespace std;

int main()
{
    int lin, col, num=1;

    for(lin=1 ; lin<=5 ; lin++){
        for(col=1 ; col<=lin ; col++){
            cout << num%2;
            num++;
        }

        cout <<endl;
    }

    return 0;
}

```

4.8

```

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

int main()
{
    int lin, col, space=3;

    for(lin=1 ; lin<=5 ; lin++){

```

```

    for(col=space ; col>=0 ; col--)
        cout<<" ";

    for(col=1 ; col<=lin ; col++){
        cout << setw(2) << "*";
    }
    space--;

    cout <<endl;
}

space=1;
for(lin=4 ; lin>0 ; lin--){
    for(col=space ; col>0 ; col--)
        cout<<" ";

    for(col=lin ; col>0 ; col--){
        cout << setw(2) << "*";
    }
    space++;

    cout <<endl;
}

return 0;
}

```

4.9

```

#include <iostream>
using namespace std;

```

```

int main()
{
    int lin, col;

    for(lin=1 ; lin<=5 ; lin++){
        for(col=lin ; col<=5 ; col++)
            cout<<col;
        cout<<endl;
    }

    return 0;
}

```



4.10

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

int main()
{
    int lin, col;

    for(lin=1 ; lin<=5 ; lin++){
        for(space=1; space<=lin-1 ; space++)
            cout<<" ";
        for(col=1 ; col<=6-lin ; col++)
            cout<<col;
        cout<<endl;
    }

    return 0;
}
```

4.11

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

int main()
{
    int lin, col;

    for(lin=5 ; lin>0 ; lin--){
        for(col=lin ; col>0 ; col--){
            cout<<setw(2)<<col;

            cout<<endl;
        }

        return 0;
    }
}
```

4.12

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

int main()
{
```

```

int lin, col, space;

for(lin=1; lin<=5 ; lin++){
    for(space=4-lin ; space>=0 ; space--){
        cout<<" ";
    }
    for(col=lin ; col>0 ; col--)
        cout<<col;
    cout<<endl;
}

return 0;
}

```

4.13

```

#include <iostream>
using namespace std;

int main()
{
    int lin, col, space;

    for(lin=1; lin<=7 ; lin++){
        for(space=1 ; space<=lin ; space++)
            cout<<" ";

        for(col=1 ; col<=8-lin ; col++)
            cout<<col;

        for(col=7-lin ; col>0 ; col--)
            cout<<col;

        cout<<endl;
    }

    return 0;
}

```

13. e 14.

Resolvida previamente.

15.

Resolvida previamente.

# Funções

Parabéns, você já concluiu três seções do curso C++ Progressivo:

1. Básico
2. Testes condicionais
3. Laços e Loopings

Agora, nesta seção, vamos aprender como deixar nossos programas cada vez maiores, mais organizados e mais úteis, através do uso de **funções**.

Vamos começar a guardar nossos códigos, para reutilizar futuramente em nosso percurso como programadores C++.

Quando for iniciar um novo projeto, você vai ver que já tem muita coisa pronta, pois programou e salvou diversas funções, que farão tarefas específicas em diversos sistemas que você irá programar.

Simbora, entender a dana das funções em C++!

# Função em C++ - O que é, Como funciona e Como criar e usar ?

Neste tutorial, daremos início ao estudo das [Funções em C++](#), onde iremos aprender:

- O que é uma função?
- Para que serve uma função?
- Como criar uma?
- Como usar uma função?

## Função em C++: O que é e Para que serve

Você já viu uma placa de algum *hardware*, como a placa do seu computador, ou as peças internas de uma televisão, celular...é mais ou menos assim:



Veja: não existe somente uma peça responsável por fazer tudo. Um placa é uma porção de pecinha fazendo pequenos trabalhos, bem específicos.

Como fazem coisas bem específicas, como contar ou medir a temperatura, por exemplo, podemos usar essas pecinhas em outras placas, como a de um carro ou de uma geladeira.

Ou seja, cada pecinha dessas tem uma **função**, específica, faz algo. Em programação C++, é a mesma coisa: função é um treco que faz uma coisa específica.

Explicando melhor, função é um bloco de código, com alguns comandos de C++ dentro, que faz determinada tarefa. Uma função pode receber e retornar informações para quem *chamou* a função.

É muito melhor, mais eficiente, mais rápido e mais viável de criar uma placa com diversas pecinha, fazendo coisa específicas, do que uma peça fazendo algo sozinha.

Da mesma maneira, não é nada viável ter um único bloco de código gigante, de milhares de linhas, pois seria extremamente difícil de trabalhar com ele, saber onde está havendo erros, onde está cada coisa...o ideal é ir *quebrando* seu programa em programinhas menores, que fazem coisas mais específicas e fáceis de serem entendidas.

É como se cada função fosse uma pecinha de lego. Você vai juntando as peças e vai montando coisas grandes e bacanas. Seu sistema operacional, por exemplo, funciona assim.

Vamos aprender como criar uma função?

## Como criar uma função em C++

A sintaxe do código para criar uma função é a seguinte:

```
tipo nome(lista_parametros)
{
    // código da
    // função

    return algo;
}
```

As funções podem ou não retornar alguma informação, como um inteiro ou um float.

Por exemplo:

int function1() {...} - retorna um inteiro

float function2() {...} - retorna um float

void function3() { ... } - não retorna nada

Dentro dos parêntesis, ela podem receber informações para trabalhar com esses dados, são os parâmetros:

`void function1(int num) {...}` - Envia um inteiro para a função e não retorna nada

`int function2(int num1, float num2) {...}` - Envia dois números como parâmetro, um inteiro e um float, e retorna um inteiro.

Se a função retorna algum tipo de dado, usamos o *return* dentro do bloco de código, para retornar o tipo de dado que declaramos no cabeçalho da função.

Vamos criar a função *hello()*, ela não retorna nenhum dado e não recebe nenhuma informação, apenas imprime o *hello, world!* na tela:

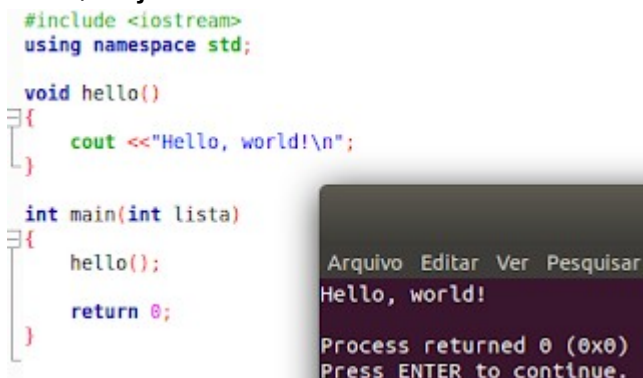
```
void hello()
{
    cout << "Hello, world!\n";
}
```

## Como chamar e usar uma função

Para chamar a função basta escrever o nome dela, seguido de parêntesis e com as informações que ela deve receber.

Para chamarmos a função criada anteriormente, fazemos simplesmente:  
`hello();`

Prontinho. O C++ vai lá buscar essa função e executar o código que tem nela, veja:



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

void hello()
{
    cout << "Hello, world!\n";
}

int main(int lista)
{
    hello();
    return 0;
}
```

Arquivo Editar Ver Pesquisar  
Hello, world!  
Process returned 0 (0x0)  
Press ENTER to continue.

Você pode invocar (chamar) a função 1, 10 ou um milhão de vezes, basta fazer: `hello();`, sem precisar repetir 1 milhão de vezes o código, basta *reutilizar*, usando funções.

## Exemplo de uso de funções em C++

Vamos criar duas funções agora.

A `menu()` exibe um menu de opções, sobre que período do dia você está.

Já na `boas_vindas()`, o usuário vai ter que digitar alguma opção, que será armazenado na variável `op`, declarada dentro da função.

Depois, ainda dentro desta função fazemos um tratamento para exibir a mensagem corretamente:

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

void menu()
{
    cout<<"Que período do dia está:\n";
    cout<<"1. Manhã\n";
    cout<<"2. Tarde\n";
    cout<<"3. Noite\n";
}

void boas_vindas()
{
    int op;
    cin>>op;

    if(op==1)
        cout<<"Bom dia!\n";
    else if(op==2)
        cout<<"Boa tarde!\n";
    else if(op==3)
        cout<<"Boa noite!\n";
    else
        cout<<"Entrada inválida\n";
}

int main()
```

```
{  
    menu();  
    boas_vindas();  
    cout<<"Encerrando...\n";  
  
    return 0;  
}
```

Note que tem três funções no código:

- 1.menu()
- 2.boas\_vindas()
- 3.main()

A main() é uma função especial do C++, ela é automaticamente executada quando compilamos e executamos nossos códigos. Dentro dela, chamamos primeiro a menu() e somente quando essa terminar de executar, chama a boas\_vindas().

Note que somente após o usuário inserir o valor na boas\_vindas() e ela te cumprimentar, é que ela vai terminar e vai voltar pra main(), que exibe um *cout* final de encerramento.

## Quando usar uma função

Sempre. O máximo que puder.

Até o momento, em nosso curso, fizemos exemplos pequenos e simples de código. Mas o normal, quando você se tornar um programador profissional, é fazer sistemas com centenas ou milhares de linhas de código, e é aí que você vai entender o quanto importante é o conceito de funções, em C++.

Você deve adquirir o hábito de criar funções pequenas, simples e de fácil entendimento e execução, pois sempre que iniciar um novo projeto, vai usar suas funções antigas e funções já existentes por padrão, do C++.

Essa técnica é chamada de dividir para conquistar, e no decorrer do nosso curso de C++ você vai ver que um programa grande e complexo nada mais é que um monte de programas pequenos e simples, conectados de maneira correta, funcionando em harmonia.



# Como Receber Informações de uma Função - O comando RETURN

Agora que você já aprendeu [o que são e para que servem as funções](#), vamos ver aprender como nos comunicar com as funções, recebendo dados e informações delas através do comando **return**.

## Trocando dados com funções em C++

No tutorial anterior, de nossa seção de [Funções em C++](#), vimos como criar e invocar uma função.

Bastou chamar pelo nome: `hello()`, que ela veio rapidinho e executou de imediato.

Poderíamos chamar 1 milhão de vezes, que ela executaria 1 milhão de vezes, sem precisar ficar repetindo código, basta invocar a função sempre que precisarmos fazer aquilo que ela se propõe a fazer.

Mas ali tem um problema...quando invocamos uma função, a execução sai do local onde houve a chamada e vai pra outro local da memória, onde estão as instruções da função, executa tudo, e volta. Nesse caso anterior, não há uma comunicação entre quem chamou e a função.

Vamos ver um código C++ onde invocamos uma função que pede o número e exibe seu dobro:

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

void doub()
{
    float num, dobro;

    cout<<"Digite um numero: ";
    cin >> num;

    dobro = 2*num;

    cout<<"Dobro: "<<dobro<<endl;
```

```
}  
  
int main()  
{  
    doub();  
    return 0;  
}
```

Legal, né? Porém, a gente não tem acesso a essa informação 'dobro', não podemos pegar ela pra usar em outro cálculo, por exemplo, ele é simplesmente exibido dentro da função e ao término dela, esse valor se perde. As variáveis internas das funções são criadas localmente, e depois excluídas.

Se tentar usar 'dobro' fora da função vai ver que dá um erro de compilação. Precisamos achar outra maneira de fazer a função se comunicar com o exterior, de enviar informações pra fora, concorda que isso é importante?

## O comando RETURN em Funções

Sempre que quisermos enviar uma informação de dentro de uma função para o local onde ela foi chamada, usamos o comando **return**.

Primeiro, no cabeçalho de declaração da função, precisamos informar o tipo de dado que a função vai retornar. No exemplo anterior é **void** porque ela não retorna nada.

Se o tipo de dado que ela vai retornar é um float, por exemplo, seu cabeçalho deve ser:

```
float doub() {...}
```

E dentro do escopo da função devemos fazer: **return info;**  
Onde *info* deve ser um float.

Pronto, a informação 'info' será retornada, devolvida para o local onde a função foi invocada.

Vamos criar uma função que pede um número, dobra ele e retorna seu dobro:

```

#include <iostream>
using namespace std;

float doub()
{
    float num, dobro;

    cout<<"Digite um numero: ";
    cin >> num;

    dobro = 2*num;

    return dobro;
}

int main()
{
    float res;
    res = doub();

    cout<<"Dobro: "<<res<<endl;
    return 0;
}

```

Agora, quando chamamos a função: `doub()`  
 Um valor é retornado para quem chamou ela, volta um *float*, no caso.

Então armazenamos o resultado desse retorno na variável `res` (obviamente, deve ser *float*):  
`res = doub();`

E prontinho, essa variável vai ter dobro do valor que o usuário digitar lá dentro da função.  
 Podemos até deixar o código mais enxuto, veja:

```

#include <iostream>
using namespace std;

float doub()
{
    float num;

```

```

    cout<<"Digite um numero: ";
    cin >> num;

    return (2*num);
}

int main()
{
    cout<<doub()<<endl;
    return 0;
}

```

Veja que podemos dar *return* numa expressão, diretamente: `return 2*num;`

## Exemplo de uso de RETURN em Funções

Vamos agora criar uma função de par ou ímpar.

Se for par, ela deve retornar 0, se for ímpar deve retornar 1.

Veja como fica nosso código C++:

```

#include <iostream>
using namespace std;

int par_impar()
{
    int num;

    cout<<"Digite um numero: ";
    cin >> num;

    if(num%2==0)
        return 0;
    else
        return 1;
}

int main()
{
    if(par_impar()==0)
        cout<<"É par!\n";
}

```

```

    else
        cout<<"É ímpar!\n";
    return 0;
}

```

Note agora que tem dois comandos *return* dentro da nossa função `par_impar()`, isso pode, mas veja que somente um ou outro pode acontecer, nunca um IF e um ELSE é executado, é sempre um deles.

Vamos dar uma enxugada nesse código?

```

#include <iostream>
using namespace std;

int par_impar()
{
    int num;

    cout<<"Digite um numero: ";
    cin >> num;

    return (num%2);
}

int main()
{
    if(!par_impar())
        cout<<"É par!\n";
    else
        cout<<"É ímpar!\n";
    return 0;
}

```

Conseguiu entender de boa?

## Boas práticas de funções

Primeiro, nomes de funções: não existe um jeito certo nem errado de usar, mas recomendamos ser consistentes com o nome delas.

De preferência crie funções com o nome no seguinte estilo: `c_progressivo()`, `projeto_progressivo()`, `van_der_graaf_generator()` ... - tudo minúsculo, separado por \_

Ou: `cProgressivo()`, `projetoProgressivo()`, `vanDerGraafGenerator()`...- primeira letra minúscula, e maiúscula para início das próximas palavras, ok?

Segundo, use muitas funções, e faça elas bem comunicativas, sempre que possível retornando e recebendo (vamos ver no próximo tutorial) dados, de maneira bem lógica e clara.

Assim como cada peça do seu carro ou cada órgão do seu corpo faz sua tarefa específica, eles também se comunicam e trocam informações entre si, isso é essencial para o bom funcionamento do sistema.

As funções de um sistema devem fazer o mesmo! Isso é uma boa prática de programação, não se esqueça! Faça suas funções serem facilmente 'acopladas' com outras funções, isso vai ser essencial para você ser capaz de criar sistemas grandes e robustos, como softwares empresariais, sistemas operacionais, jogos, etc.

# Enviando dados para Funções em C++ - Parâmetros e Argumentos

Neste tutorial, vamos aprender como enviar dados para as [funções](#), em linguagem de programação C++.

## Enviando Dados para Funções

No tutorial passado, de nosso **curso de C++**, aprendemos [como as funções retornam informações para quem as invocou, através do comando \*return\*](#).

As funções podem receber informações através do uso de *parâmetros*, que é uma variável especial, declarada no cabeçalho da função, que ficará responsável por armazenar os valores que serão passados para estas funções. Esses valores são os *argumentos*.

Para usarmos os argumentos (ou seja, passarmos algum valor para funções trabalharem), devemos informar a lista de parâmetros, dentro dos parêntesis da declaração da função.

Por exemplo, uma função que retorna um inteiro, recebe um inteiro e se chama *func*, se declara assim:

- `int func(int var) { ... }`

Para invocar, passando o número 2112, por exemplo, fazemos:

- `func(2112);`

*var* é o parâmetro, já 2112 é um argumento.

Agora uma função que não retorna nada, se chama *func2* e recebe um inteiro, um float e um booleano:

- `void func2(int var, float num, bool status) { ... }`

Se quisermos passar o inteiro 1, o float 21.12 e o booleano true, para dentro desta função, fazemos:

- `func2(1, 21.12, true);`

Lista de parâmetros: var, num e status

Argumentos (valores): 1, 21.12 e true

Veja a ordem: um inteiro, um float e um booleano! Respeite a ordem da lista de parâmetros do cabeçalho da função!

## Exemplo de parâmetros e argumentos em C++

Vamos criar uma função que recebe um número qualquer, e retorna ele ao quadrado:

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

float quadrado(float num)
{
    return num*num;
}

int main()
{
    float var;

    cout<<"Numero para elevar ao quadrado: ";
    cin >> var;

    cout<<var<<"*"<<var<<" = " << quadrado(var)<<endl;
}
```

Note que passamos a variável *var*, que é um *float*, para a função *quadrado()*, que recebe um float como argumento.

Embora tenhamos passado a variável *var*, o que acontece na verdade, por debaixo dos panos, é que passamos um número, um valor, que foi digitado pelo usuário.

Se passarmos *var = 5*, o que vai pra função é 5.

É como se tivéssemos feito: *quadrado(5)*;

Dentro da função, esse valor é armazenado no parâmetro *num*, ok?



A função não 'vê' a variável *var*, nem sabe de sua existência, só pega seu valor e copia para a variável *float num*.

## Parâmetros e Argumentos em C++

Vamos agora criar uma função que recebe um número, eleva ele ao cubo e retorna esse valor.

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

float cubo(float num)
{
    num = num*num*num;

    return num;
}

int main()
{
    float num;

    cout<<"Numero para elevar ao cubo: ";
    cin >> num;

    cout<<"Numero informado: "<<num<<endl;
    cout<<"Cubo: "<< cubo(num)<<endl;
    cout<<"Valor de num: "<<num<<endl;
}
```

Fizemos um teste nesse exemplo.

O parâmetro é *num*, dentro da função.

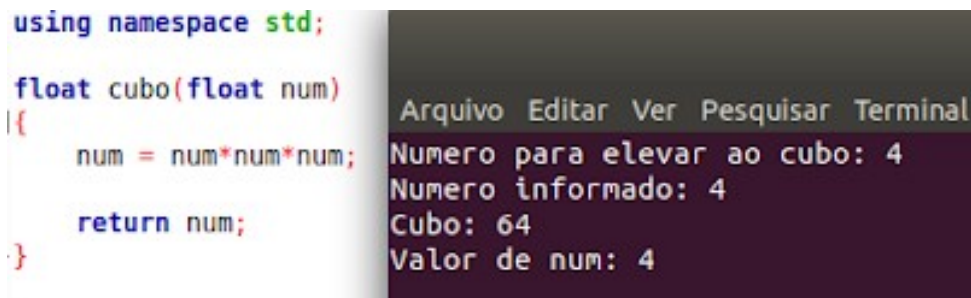
Ao invocar a função, também passar uma variável de nome *num*.

Dentro da função, alteramos o valor de *num*.

Antes, era *num*, depois passa a ser  $(num*num*num)$ , e retornamos esse novo valor de *num*.

Na *main()*, vamos exibir o número que o usuário digitar (*num*), o cubo (invocando a função *cubo()*), e depois exibimos novamente o valor de *num*.

Para num=4, o resultado é:

The image shows a code editor on the left and a terminal window on the right. The code editor contains the following C++ code:

```
using namespace std;

float cubo(float num)
{
    num = num*num*num;
    return num;
}
```

The terminal window has a menu bar with 'Arquivo', 'Editar', 'Ver', 'Pesquisar', and 'Terminal'. The output in the terminal is:

```
Numero para elevar ao cubo: 4
Numero informado: 4
Cubo: 64
Valor de num: 4
```

Note que o valor de *num* foi alterado SOMENTE dentro da função! Quando passamos um argumento pra uma função, a função faz uma cópia dele e atribui ao parâmetro específico. Dentro da função, você alterou o valor do parâmetro, a variável *num* original, não é alterada!

Ah...agora você já consegue entender um pouco mais a main():

```
int main(){
...
    return 0;
}
```

Ou seja, ela é uma função que não recebe nenhum argumento (pois não tem parâmetro) e retorna um inteiro, no caso, ela retorna 0.

Dois adendos:

1. Universalmente, quando uma função retorna 0, é porque deu tudo certo, tudo ok. Quando seu código funcionar ok, deverá retorna 0
2. Nesse caso específico, a main() não tem parâmetros, mas tem versões dela recebendo argumentos sim, veremos mais adiante em nosso curso

# Protótipo de Função - Como programar uma calculadora completa em C++

Neste tutorial de nosso **curso de C++**, vamos aprender como criar uma calculadora completa em C++, bem como entender o que são e para que servem os protótipos de funções.

## Programando uma Calculadora em C++

Vamos agora programar uma calculadora, bem funcional e útil, usando os conhecimentos que aprendemos de C++:

- [Básico](#) (operações matemáticas)
- [Teste condicional](#) (IF e ELSE)
- [Laços e loopings](#) (DO WHILE)
- [Funções](#)

As funções `sum()`, `sub()`, `mult()` e `divis()` fazem as operações de soma, subtração, multiplicação e divisão, respectivamente. Todas recebem dois dados, do tipo `float`, e retornam um resultado em `float` também.

Exceto pela função `divis()`, pois ela precisa testar se o denominador é diferente de 0.

Se for, retorna a divisão corretamente.

Se não for, retorna uma mensagem dizendo que não é possível dividir por 0, que é o correto a se fazer.

A função `menu()` é a responsável por mostrar as possíveis operações matemáticas que ele pode fazer.

O usuário digita um valor como opção.

Se essa opção for 0, o programa sai do looping DO WHILE da `menu()` e se encerra o programa.

Se digitar de qualquer outro número, pedimos os dois valores que ele vai querer calcular a operação e em seguida vai pro SWITCH, onde selecionamos a operação corretamente.

Passamos os números digitados pelo usuário pra respectiva função, ela retorna o resultado e o exibimos.

Por fim, o menu de opções é exibido novamente, para caso o usuário queira fazer outro cálculo.

Veja como fica nosso código:

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

float sum(float a, float b)
{
    return a+b;
}

float sub(float a, float b)
{
    return a-b;
}

float mult(float a, float b)
{
    return a*b;
}

float divis(float a, float b)
{
    if(b!=0)
        return a/b;
    else
        cout<<"Não pode dividir por 0\n";
}

void menu()
{
    int op;
    float a, b;
    do{
        cout<<"0. Sair\n";
        cout<<"1. Somar\n";
        cout<<"2. Subtrair\n";
        cout<<"3. Multiplicar\n";
```

```
cout<<"4. Dividir\n";
```

```
cin >> op;
```

```
if(op){
```

```
    cout<<"\nPrimeiro numero: ";
```

```
    cin >> a;
```

```
    cout<<"Segundo numero: ";
```

```
    cin >> b;
```

```
    switch(op){
```

```
        case 1:
```

```
            cout<<"Soma: " << sum(a,b) << endl;
```

```
            break;
```

```
        case 2:
```

```
            cout<<"Diferença: " << sub(a,b) << endl;
```

```
            break;
```

```
        case 3:
```

```
            cout<<"Produto: " << mult(a,b) << endl;
```

```
            break;
```

```
        case 4:
```

```
            if(b)
```

```
                cout<<"Divisão: " << divis(a,b) << endl;
```

```
            else
```

```
                divis(a,b);
```

```
            break;
```

```
        default:
```

```
            cout<<"Opção inválida\n";
```

```
    }
```

```
    }else
```

```
        cout<<"Saindo...\n";
```

```
    cout<<endl;
```

```
    }while(op);
```

```
}
```

```
int main()
```

```
{
```

```
    menu();
```

```
    return 0;
```

```
}
```

Na função `main()` nós simplesmente invocamos a função responsável por exibir o menu.

E só.

Veja, usamos funções, operações matemáticas, IF e ELSE, SWITCH e DO WHILE.

Basicamente, todo conhecimento que estudamos até aqui, em nosso **curso de C++**.

Agora faça um exercício.

## Protótipos de funções em C++

Pegue a função `main()`, recorte ela do código e cole lá em cima, antes das funções das operações matemáticas. Agora rode seu código.

Deve dar algum erro do tipo 'sum() was not declared', dizendo que a função `sum()` não foi declarada.

Isso ocorre pois o compilador lê o código de cima pra baixo.

Quando ele entra na `menu()` e esta invoca a `sum()`, ele não sabe o que fazer, pois esta função ainda não foi declarada, ela está abaixo da `menu()`. Por isso, o correto é declarar primeiro as funções de operações matemáticas, e só depois invocar elas na `menu()`.

Agora imagine num programa mais complexo e bem maior, como isso ficaria complicado.

Teríamos que ter cuidado com o que declarar antes de que, e o código antes da função principal `main()` ficaria gigantesco.

A solução pra isso é, lá em cima, declarar apenas os protótipos das funções. O protótipo nada mais é que o cabeçalho da função, com os tipos de dados que recebe e o return correto, seguido de ponto-e-vírgula, sem o par de chaves e o código da função.

Veja agora como fica o código de nossa calculadora, usando os protótipos:

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

```
void menu();
float sum(float a, float b);
float sub(float a, float b);
float mult(float a, float b);
float divis(float a, float b);
```

```
int main()
{
    menu();
    return 0;
}
```

```
void menu()
{
    int op;
    float a, b;
    do{
        cout<<"0. Sair\n";
        cout<<"1. Somar\n";
        cout<<"2. Subtrair\n";
        cout<<"3. Multiplicar\n";
        cout<<"4. Dividir\n";
        cin >> op;

        if(op){
            cout<<"\nPrimeiro numero: ";
            cin >> a;

            cout<<"Segundo numero: ";
            cin >> b;

            switch(op){
                case 1:
                    cout<<"Soma: " << sum(a,b) << endl;
                    break;
                case 2:
                    cout<<"Diferença: " << sub(a,b) << endl;
                    break;
                case 3:
                    cout<<"Produto: " << mult(a,b) << endl;
```

```

        break;
    case 4:
        if(b)
            cout<<"Divisão: " << divis(a,b) << endl;
        else
            divis(a,b);
        break;
    default:
        cout<<"Opção inválida\n";
    }
}
else
    cout<<"Saindo...\n";
cout<<endl;
}while(op);
}

float sum(float a, float b)
{
    return a+b;
}

float sub(float a, float b)
{
    return a-b;
}

float mult(float a, float b)
{
    return a*b;
}

float divis(float a, float b)
{
    if(b!=0)
        return a/b;
    else
        cout<<"Não pode dividir por 0\n";
}

```

Bem mais bonito e organizado, não acha?



Aliás, o compilador só precisa saber o retorno, o nome da função, quantos e que tipos de parâmetros a função tem, precisa nem do nome deles, você pode declarar assim também:

```
void menu();  
float sum(float, float);  
float sub(float, float);  
float mult(float, float);  
float divis(float, float);
```

Recomendamos sempre usar protótipos de funções, deixa seu código mais organizado e menos suscetível a erros e problemas

# Variável Local, Global, Constante Global e Variável estática

Neste tutorial de nossa **apostila de C++**, vamos aprender um pouco mais da relação entre funções e variáveis, conhecendo os tipos locais, global, constantes e estáticas.

## Variável Local

Quando declaramos uma variável dentro de uma função, dizemos que ela é local.

Isso se deve ao fato dela 'existir' apenas pra quem está dentro da função, ou seja, só ali dentro podem enxergar ela e seu valor.

Outros comandos, em outras funções, não podem acessar ela normalmente.

Para ilustrar isso, vamos declarar a variável *myVar* com valor 1, dentro da `main()`.

Em seguida, vamos chamar a função `imprime()`, que vai imprimir o valor de *myVar*:

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

```
void imprime()
{
    cout<<myVar;
}
```

```
int main()
{
    int myVar=1;
    imprime();
    return 0;
}
```

Esse código nem compilado é, aparece o erro:  
" 'myVar' was not declared in this scope|"

Ou seja, a variável *myVar* não está dentro do escopo da função `imprime()`, é como se ela não existisse.

Vale o contrário também:

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

```
void imprime()
{
    int myVar=1;
}
```

```
int main()
{
    cout<<myVar;
    return 0;
}
```

Ou seja, nem mesmo a função `main()` consegue visualizar o conteúdo interno da função `imprime()`.

Agora vamos declarar a *myVar* tanto dentro da `imprime()` quanto dentro da `main()`, e vamos imprimir os dois valores:

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

```
void imprime()
{
    int myVar=1;
    cout << "Na imprime() = "<<myVar<<endl;
}
```

```
int main()
{
    int myVar=2;
    cout << "Na main() = "<<myVar<<endl;
    imprime();
    return 0;
}
```

O resultado é:

```
Na main() = 2
Na imprime() = 1
```

Ou seja, podemos declarar variáveis de nomes iguais, mas em funções diferentes.

E as funções só 'enxergam' aquilo que está dentro da função, ok?

## Variável Global

Existe uma maneira de fazer com que uma mesma variável seja vista e acessada por diversas funções, basta fazê-la ser uma variável **global**.

Para isso, basta declarar ela *fora* do escopo das funções.

Por exemplo, vamos declarar a variável *pi*, e armazenar o valor do pi.

Em seguida, vamos pedir o valor do raio ao usuário, e através de outras duas funções, calculamos o perímetro e a área do círculo:

```
#include <iostream>
using namespace std;
float pi = 3.14;
```

```
float perimeter(float r)
{
    return 2*pi*r;
}
```

```
float area(float r)
{
    return pi*r*r;
}
```

```
int main()
{
    float rad;

    cout<<"Raio: ";
    cin>>rad;
```

```

cout<<"Perímetro: " << perimeter(rad)<<endl;
cout<<"Área    : " << area(rad)<<endl;

return 0;
}

```

Veja que usamos a variável *pi* nas funções, sem ter declarado dentro do escopo delas.

Se fossemos usar variáveis locais, teríamos que declarar a *pi* mais de uma vez, o que não seria muito eficiente.

Use variáveis globais quando elas forem necessárias em vários trechos diferentes de seu código.

E você pode usar o mesmo nome para uma variável local e global. Nesse caso, a variável local vai ter prioridade.

## Constantes Globais

Muitíssimo cuidado ao usar variáveis globais. Em programas complexos, é fácil 'perder o controle' de variáveis globais, visto que elas podem ser alteradas em qualquer lugar do código.

Na maioria dos casos, dê prioridade para o uso de argumentos mesmo.

Porém, caso queira usar variáveis globais que não devam ser alteradas, use a palavra-chave **const** antes declarar a variável:

- const float pi = 3.14

Vamos supor que um hacker invadiu seu sistema, e inseriu uma função chamada muda(), que vai alterar o valor do *pi* para 4, sabotando seu projeto:

```

#include <iostream>
using namespace std;
const float pi = 3.14;

void muda()
{
    pi = 4;
}

```

```
int main()
{
    muda();

    return 0;
}
```

Como a variável foi declarada com a keyword **const**, vai dar o erro:  
 "|7|error: assignment of read-only variable 'pi' | "

Ou seja, a variável é somente para leitura, não pode alterar o 'pi'. E, de fato, em nenhum programa se deve alterar o pi, logo, faz sentido ele ser global e constante, concorda?

Vamos supor que você vai criar um sistema para uma grande rede de supermercados, e vai definir o preço do desconto em 10%, faça:

- const float desconto = 0.1;

Pronto. Agora, milhares de funções podem acessar o valor do desconto. E caso você queira aumentar o desconto pra 20%?

Moleza, só fazer:

- const float desconto = 0.2;

Veja que você alterou só uma coisinha, só uma variável. Mas, automaticamente, modificou diretamente os cálculos das milhares de funções que usam essa variável.

Altera só uma vez, e a alteração ocorre em vários cantos.

Se tivesse feito isso de maneira local, teria que ir em cada função e alterar variável por variável...ia levar horas ou dias.

Mas com variável global constante, não. Altera só uma vez. E tudo muda. Sacou a utilidade de variáveis globais e constantes?

## Variável estática local

Quando declaramos uma variável dentro de uma função, ela é criada e reservada na memória do computador no início da função e é destruída, ao término da execução da função.

No exemplo de código abaixo, inicializamos a variável *myVar* com valor 1, imprimimos, incrementamos em uma unidade e terminamos a função.

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

void test()
{
    int myVar=1;

    cout<<myVar<<endl;

    myVar++;
}

int main()
{
    test();
    test();

    return 0;
}
```

Chamamos a função `test()` duas vezes, e o que aparece na tela é sempre o mesmo: o valor 1.

Cada vez que a função roda, a variável é criada, inicializada e o valor 1 é exibido. Ela é incrementada, mas depois a função termina e a variável morre, simplesmente.

Dizemos então que as variáveis locais são não-persistentes. Ou seja, elas não 'persistem', elas são criadas e destruídas, junto com a função.

Existe uma maneira de fazer com que a variável seja persistente, ou seja, que ela seja criada de início e não seja destruída, ou seja, seu endereço e valor na memória se mantém. São as variáveis estáticas.

Para declarar uma variável como sendo estática, basta usarmos a keyword *static* antes da declaração:

- `static int myVar;`

Veja:

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

```
void test()
{
    static int myVar=1;

    cout<<myVar<<endl;

    myVar++;
}
```

```
int main()
{
    test();
    test();
    test();
    test();

    return 0;
}
```

Pronto. Não importa quantas vezes você chama a função `test()`, a variável `myVar` que você vai usar é declarada e inicializada apenas uma vez. Quando a função terminar, ela ainda vai persistir e quando chamar novamente a função, ela já vai pegar o valor anterior da variável, pra imprimir e incrementar.

Assim como as variáveis globais, as estáticas locais sempre são inicializadas com valor 0, caso você não inicialize explicitamente. E caso inicialize, essa inicialização vai ocorrer somente uma vez, como você viu no exemplo de código anterior, ok?



# Argumentos Padrão e Omissão de Argumentos

Neste tutorial de C++, vamos aprender o que é um argumento padrão, para que serve e como usar, bem como omitir um argumento numa chamada de uma função.

## Argumento Padrão em C++

Já aprendemos como enviar informações para funções, através do uso de [parâmetros e argumentos](#).

Vamos supor que queiramos somar dois números, a e b, a função seria:

```
float sum2(float a, float b)
{
    return a+b;
}
```

Agora vamos supor que queiramos calcular a soma de três variáveis, teríamos que fazer uma função assim:

```
float sum3(float a, float b, float c)
{
    return a+b+c;
}
```

Note que teríamos que usar outra função, com outro nome, pro mesmo propósito: somar os argumentos. Não é algo muito inteligente, concorda? Seria legal se uma mesma função somasse 2 ou 3 argumentos, o tanto que o usuário quisesse.

Para somar dois números, seria interessante fazer: `sum(1,2);`

Para somar três números, faríamos: `sum(1,2,3);`

É aí que entra o conceito de argumento padrão.

Basta declararmos o protótipo da função assim:

```
•float sum(float a, float b, float c = 0.0);
```

E seu escopo:

```
float sum(float a, float b, float c = 0.0)
{
    return a+b+c;
}
```

O que ocorre é o seguinte:

O valor padrão de c é 0.0

Caso você faça: sum(1,2, 3), o valor de c será 3.

Caso faça: suma (1,2), você não estará definindo valor para c, logo ele vai assumir o valor padrão, que é 0. Entendeu? Argumento com valor padrão, caso você não forneça esse valor.

Teste:

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

float sum(float a, float b, float c = 0.0)
{
    return a+b+c;
}

int main()
{
    cout<<sum(1,2)<<endl;
    cout<<sum(1,2,3)<<endl;

    return 0;
}
```

Funciona pra 2 ou 3 variáveis, a gosto do freguês!

## Omissão de argumentos

O código anterior funciona para somar 2 ou 3 números.

Para somarmos 2, 3 ou 4 números, poderíamos fazer:

```
float sum(float a, float b, float c = 0.0, float d = 0.0)
{
    return a+b+c+d;
}
```

Agora você pode fazer:

```
sum(1,2);
sum(1,2,3);
sum(1,2,3,4);
```

No primeiro caso, omitimos o argumento c e o d.

No segundo exemplo, omitimos o argumento d.

No último exemplo, não omitimos nenhum argumento.

Ou seja, argumentos padrões são automaticamente substituídos, quando informamos os argumentos.

## Regras no uso de argumentos padrão

Quando informamos e passamos argumentos para uma função, eles são copiados da esquerda pra direita.

Por exemplo: `sum(1,2,3)`

O 1 vai pro 'a', o 2 pro 'b' e o valor 3 vai pro parâmetro 'c'. O valor do argumento 'd', então, é o argumento padrão, que definimos como 0.

Outra regra é que, uma vez que usemos um argumento padrão em um parâmetro, todos os outros parâmetros subsequentes deverão também ter argumentos padrão também. Por exemplo, o protótipo de função a seguir é válido:

```
•float volume(float base, float height=1, float width=1);
```

'height' é um argumento padrão, e o seguinte também.

Já o seguinte protótipo de função é inválido:

•float volume(float height=1, float base, float width=1);

Como o primeiro argumento é padrão, todos os outros devem ser, e o 'base' é um parâmetro normal, que deve ser necessariamente fornecido pela chamada da função.

Outro ponto é que os parâmetros com argumentos padrão devem ser declarados na primeira ocorrência da declaração da função. Ou seja, se você usar um protótipo de uma função e depois em outro lugar vai definir o escopo da sua função, os argumentos padrão devem ser definidos já no protótipo, que vai vir antes. Por exemplo, você pode inclusive abreviar assim:

Protótipo: double area( double = 1.0, float 2.0);

Definição da função: double area (double length, float width) { return length\*width; }

Assim, recapitulando, seja a função:

```
double area (double length = 1.0, float width = 2.0)
{
    return length*width;
}
```

Se fizermos as seguintes chamadas de função:

- 1.area() - serão usados os valores 1.0 e 2.0 para calcular a área, respectivamente, ou seja, estamos usando os dois argumentos padrão.
- 2.area(3) - serão usados os valores 3 e 2.0 para calcular a área, ou seja, o primeiro argumento padrão foi substituído por 3
- 3.area(3,6) - serão usados os valores 3 e 6 para calcular a área, ou seja, os dois argumentos padrão foram substituídos.

Vamos ver isso na prática? Teste o seguinte código:

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

void area(float = 1.0, float = 1.0);

int main()
```

```

{
    cout << "Nenhum argumento passado:"<<endl;
    area();

    cout << "\nPrimeiro argumento passado:"<<endl;
    area(2.0);

    cout << "\nAmbos argumentos passados:"<<endl;
    area(2.0, 3.0);

    return 0;
}

void area(float length, float width)
{
    cout << "Area: " << length * width << endl;
}

```

Deu tudo certo? Sim? Então vamos seguir em nossos tutoriais de C++.

# Parâmetros e Variáveis de Referência

Neste tutorial de nosso **curso de C++**, vamos aprender a passagem de argumentos por referência, em funções.

## Passagem por Valor

Já aprendemos [como enviar informações para funções, através dos argumentos e parâmetros](#).

E vimos que esta passagem é chamada [passagem por valor](#), pois só passamos o valor para a função.

Se enviamos uma variável pra uma função, e dentro dessa função essa variável é alterada, ela não é alterada fora da função. Teste o seguinte código, que eleva um número ao quadrado:

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

void square(int num);

int main()
{
    int number = 6;

    cout<<"Antes : num = "<<number<<endl;
    square(number);
    cout<<"Depois : num = "<<number<<endl;

    return 0;
}

void square(int num)
{
    num = num*num;
    cout<<"Durante: num = "<<num<<endl;
}
```

Quando invocamos a função: `square(number)`, estamos na verdade, passando uma **cópia** do valor da variável *number*.

Dentro da função, a variável *num* vai receber uma cópia do valor de *number*. Ou seja, ela não vai ter acesso a variável original *number*, somente ao seu valor! Por isso, passagem por valor.

## Parâmetro de Referência: &

Seja a declaração de variável:

```
•int num = 6;
```

O que estamos fazendo aí é alocando, reservando, um espaço na memória do seu computador.

Assim, quando usamos 'num', o C++ entende que devemos ir no endereço para onde essa variável aponta e pegar o número que está dentro daquele local da memória.

Como vimos, quando passamos essa variável normal para uma função que espera uma variável normal, ela pega apenas uma **cópia** de seu valor, e não mexe no conteúdo original.

Porém, existe outro tipo especial de variável, a *variável de referência*.

Ela é especial, pois se passarmos uma variável para uma função e a função quiser obter a *variável de referência*, através de um parâmetro de referência, ela vai ter acesso ao endereço da memória (a referência), onde esta variável está originalmente armazenada.

Ou seja: vamos ter acesso, de verdade, a variável, e não somente a sua cópia.

Para pegarmos a referência de uma variável, basta usarmos o operador & antes do nome da variável, no **parâmetro** da função! É o parâmetro de referência. Tanto no protótipo quanto na declaração da função.

Assim, o exemplo de código anterior, fica:

```

#include <iostream>
using namespace std;

void square(int &);

int main()
{
    int number = 6;

    cout<<"Antes : num = "<<number<<endl;
    square(number);
    cout<<"Depois : num = "<<number<<endl;

    return 0;
}

void square(int &num)
{
    num = num*num;
    cout<<"Durante: num = "<<num<<endl;
}

```

Veja agora o resultado:

```

Antes   : num = 6
Durante: num = 36
Depois  : num = 36

```

De fato, a função conseguiu mudar o valor da variável, pois essa passagem de argumento para o parâmetro foi via parâmetro de referência.

O que acontece aqui é que o parâmetro não vai capturar o valor do argumento, mas sim sua referência, para onde ele está apontando na memória. Falando em apontar para um endereço, estudaremos mais sobre esse assunto nas seções de ponteiros e arrays (vetores) em C++, onde estudaremos sobre passagem por referência, usando ponteiros.



## Mais sobre parâmetros de referência em C++

Alguns programadores também preferem declarar o protótipo assim:

- `square (int&);`

Você também pode usar assim no protótipo:

- `square(int &num);`
- `square(int& num);`

Também pode, o importante é que o `&` esteja tanto no protótipo quanto na declaração da função, ok?

Lembrando que se seu parâmetro é de referência, ele só pode trabalhar com variável de referência.

Seria um erro se você usar um argumento que não é uma variável, como um literal, uma expressão ou uma constante, por exemplo:

- `square(10);` // tem que usar `square(number);`
- `square(number+1);` // é uma expressão, evite

Quando fazemos: `square(int &num)`

Leia-se: "num é uma referência para um inteiro", ou seja, ele está referenciando, apontando, indicando um local da memória em que está armazenado um inteiro. Como você tem o local da variável original, você consegue mudar esse valor.

### **Exercício de passagem de valor e referência**

*Crie um programa que pede um inteiro ao usuário. Em seguida, ele deve transformar esse valor em seu cubo. Faça isso usando uma função que recebe passagem por valor e outra que usa parâmetro e variável de referência. Deixe bem claro que uma altera o valor somente dentro da função (valor) e a outra altera o valor original da variável (referência).*

# Sobrecarga de Funções: Parâmetros e Argumentos diferentes

Neste tutorial de nossa **apostila de C++**, vamos aprender o que é sobrecarga de funções, para que servem e como usar esta importante técnica de programação.

## Tamanhos de Parâmetros e Argumentos em Funções no C++

No tutorial sobre [Argumentos Padrão](#), vimos que é possível enviar uma quantidade variada de argumentos para uma função, desde que ela esteja usando parâmetros de referência.

Por exemplo, o código abaixo é de uma função que calcula uma média aritmética:

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

float average(float a, float b, float c, float d = 0.0)
{
    return (a+b+c+d)/4;
}

int main()
{
    cout<<"Media de 3 numeros: "<<average(10, 9, 7)<<endl;
    cout<<"Media de 4 numeros: "<<average(10, 9, 7, 6)<<endl;

    return 0;
}
```

Ela pode receber 3 ou 4 argumentos. Se enviar apenas 3, o 4 argumento é o padrão, de valor 0.

Porém, note um erro.

Mesmo que enviemos apenas 3 argumentos, a média aritmética é calculada como se houvessem 4 notas.

Seria interessante se, caso eu enviasse 3 argumentos, ela retornasse:  
 $(a+b+c)/3$

E caso eu enviasse 4 argumentos, ela retornasse:  
 $(a+b+c+d)/4$

E isso é possível, com sobrecarga de funções.

## Sobrecarga de Funções em C++

Sobrecarga, ou *overloading*, é a técnica que permite que tenhamos funções com mesmo nome, desde que seus parâmetros sejam diferentes.

Veja como ficaria o exemplo do programa que calcula as médias de 3 ou 4 notas:

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

float average(float a, float b, float c)
{
    return (a+b+c)/3;
}

float average(float a, float b, float c, float d)
{
    return (a+b+c+d)/4;
}

int main()
{
    cout<<"Media de 3 numeros: "<<average(10, 9, 7)<<endl;
    cout<<"Media de 4 numeros: "<<average(10, 9, 7, 6)<<endl;

    return 0;
}
```

Note que invocamos a função `average()`, a única diferença é que na primeira chamada passamos 3 argumentos, e na segunda chamada da função passamos 4 argumentos.

Como o C++ é maroto e inteligente, ele sabe qual função rodar, corretamente.

Ele é tão esperto que, mesmo que exista o mesmo número de parâmetros/argumentos, ele consegue diferenciar através do tipo de dado que estamos usando.

Veja:

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

double average(double a, double b, double c)
{
    cout<<"Media de 3 double  : ";
    return (a+b+c)/3;
}

int average(int a, int b, int c)
{
    cout<<"Media de 3 inteiros : ";
    return (a+b+c)/3;
}

int main()
{
    cout<<average(10.0, 9.0, 7.0)<<endl;
    cout<<average(10, 9, 7)<<endl;

    return 0;
}
```

Quando passamos variáveis do tipo double, ele chama a double average()

Quando passamos variáveis do tipo int, ele chama a int average()

O C++ consegue diferenciar pois as *assinaturas* de cada função são diferentes (seja no número de parâmetros ou no tipo que vai trabalhar).

Aliás, até se as funções tiverem o mesmo nome, o mesmo número de parâmetros e os mesmos tipos de dados, podemos fazer sobrecarga de funções, desde que a ordem dos parâmetros seja diferente:

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

void func(int a, double b)
{
    cout<<"Primeiro é int : "<<a<<endl;
    cout<<"Segundo é double: "<<b<<endl;
}

void func(double b, int a)
{
    cout<<"Primeiro é double: "<<b<<endl;
    cout<<"Segundo é int   : "<<a<<endl;
}

int main()
{
    func(1, 2.5);
    cout<<endl;
    func(2.5, 1);

    return 0;
}
```

No exemplo assim temos: func(int, double)  
E também: func(double, int)

Basta que a assinatura de uma função para outra seja diferente, para podermos usar sobrecarga, onde assinatura é um conjunto de dados: nome da função, tipo de dados, número de dados e ordem das informações.

No que se refere as boas práticas de programação, você deve usar sobrecarga de funções sempre que precisar usar funções com mesmo propósito e mesma lógica, mas para número e/ou tipos e/ou ordem de dados diferentes.

# Função Recursiva em C++

Neste **tutorial de C++**, vamos aprender a famosa e importante arte da recursividade, e você vai entender a piadinha de programadores: "Para saber recursão, tem que saber recursão"

## Função Recursiva

Nos programas que fizemos no decorrer de nossos tutoriais, fizemos com que, várias vezes, uma função fosse chamada dentro da outra. Recursão tem a ver com invocar uma função.

Mas ela chama uma função em especial. A função recursiva nada mais é a função que chama ela mesma.

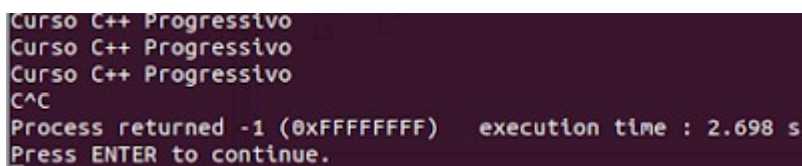
Vamos criar uma função que exibe uma mensagem na tela e depois invoca ela mesma:

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

void recur()
{
    cout<<"Progressive C++ course"<<endl;
    recur();
}

int main()
{
    recur();
    return 0;
}
```

Acontece o seguinte:

A screenshot of a terminal window with a dark background. It shows the output of the recursive program: "Curso C++ Progressivo" printed three times, followed by a cursor character "C^C". At the bottom, it displays "Process returned -1 (0xFFFFFFFF) execution time : 2.698 s" and "Press ENTER to continue.".

```
Curso C++ Progressivo
Curso C++ Progressivo
Curso C++ Progressivo
C^C
Process returned -1 (0xFFFFFFFF)   execution time : 2.698 s
Press ENTER to continue.
```

E é bem simples de entender...invocamos a recur(), ela exibe uma mensagem na tela e ... chama ela novamente, que ao ser chamada, exibe uma mensagem na tela...depois chama ela novamente, que exibe uma mensagem...e assim vai, indefinidamente, para o infinito e além.

Pra parar, tive que dar um Control+C, senão ficava rodando pra sempre.

## Como usar recursão com funções

Desse jeito, as funções recursivas não são tão úteis.

Para aprender como usar elas da maneira correta, precisamos de uma espécie de contador, assim como fizemos com os loopings.

Vamos fazer com que a recur() receba um inteiro, e só vai chamar ela mesmo se esse inteiro for maior que 0:

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

void recur(int counter)
{
    if(counter>0){
        cout<<"Curso C++ Progressivo"<<endl;
        recur(counter-1);
    }
}

int main()
{
    recur(5);
    return 0;
}
```

Se o número recebido for maior que 0, exibimos a mensagem e chamamos novamente a recur(), porém, vamos passar um argumento menor, subtraído de 1, pois essa função já executou uma vez.

Assim, se você fizer recur(5), ela vai rodar a função 5 vezes apenas:

- 1.Primeiro chamamos recur(5), exibiu a mensagem e chamou recur(4).
- 2.recur(4) exibe a mensagem e chama recur(3).



- 3.recur(3) exibe a mensagem e chama recur(2).
- 4.recur(2) exibe a mensagem e chama recur(1)
- 5.recur(1) exibe a mensagem e chama recur(0).

Quando chamamos recur(0), nada é feito e nenhuma função é mais invocada, encerrando a recursão.

```
Curso C++ Progressivo
Curso C++ Progressivo
Curso C++ Progressivo
Curso C++ Progressivo
Curso C++ Progressivo

Process returned 0 (0x0)   execution time : 0.003 s
Press ENTER to continue.
```

recur(0) é o caso base, onde ela deve parar.

Vamos praticar e ver como aplicar a técnica da recursividade com funções, em C++?

## Somatório com recursão

Vamos chamar de sum(n) o somatório do valor n.

Por exemplo:

$$\text{sum}(5) = 5 + 4 + 3 + 2 + 1$$

$$\text{Mas sum}(4) = 4 + 3 + 2 + 1$$

$$\text{Ou seja: sum}(5) = 5 + \text{sum}(4)$$

Podemos generalizar fazendo:

$$\text{sum}(n) = n + \text{sum}(n-1)$$

Veja que tem uma recursão aí. A sum() está chamando a sum(), com um argumento decrementado em 1, mas está. Quando o argumento for 1, ele deve retornar 1 (pois somatório de 1 é 1) e parar de invocar a função de somatório.

Nosso código fica:

```

#include <iostream>
using namespace std;

int sum(int num)
{
    if(num==1)
        return 1;
    else
        return num+sum(num-1);
}

int main()
{
    int num;

    cout<<"Somatório de: ";
    cin>> num;

    cout<<"Igual a: "<<sum(num)<<endl;
}

```

sum(1) é o caso base, onde resolvemos as coisas manualmente, sem usar recursão, apenas retornando 1.  
Bacana. né?

## Fatorial com recursão

Vamos chamar de fat(n) o fatorial do valor n.  

$$\text{fat}(n) = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

Ou seja:

$$\text{fat}(n) = n * \text{fat}(n-1)$$

Concorda?

Temos aí uma lógica com recursividade. A função fat() chamando a fat(). Ela deve chamar até chegar no argumento 1, e fatorial de 1 é 1.

Veja como fica nosso código:

```

#include <iostream>
using namespace std;

int fat(int num)
{
    if(num==1)
        return 1;
    else
        return num*fat(num-1);
}

int main()
{
    int num;

    cout<<"Fatorial de: ";
    cin>> num;

    cout<<"Igual a: "<<fat(num)<<endl;
}

```

fat(1) é o caso base, onde não invocamos a recursividade, e sim retornamos diretamente um valor.

## Sequência de Fibonacci com recursividade

Por fim, nosso bom e velho Fibonacci.

Relembre aqui o que é a [série de Fibonacci](#).

Basicamente, os dois primeiros termos são 0 e 1. São nossos casos base. O terceiro termo, em diante, é a soma dos dois anteriores. Ou seja:

$$F(1) = 0$$

$$F(2) = 1$$

$$F(3) = F(2) + F(1) = 1 + 0 = 1$$

$$F(4) = F(3) + F(2) = 1 + 1 = 2$$

$$F(5) = F(4) + F(3) = 2 + 1 = 3$$

$$F(6) = F(5) + F(4) = 3 + 2 = 5$$

$$F(7) = F(6) + F(5) = 5 + 3 = 8$$

...

Ou seja, a fórmula para achar o termo 'n' é:

$$F(n) = F(n-1) + F(n-2)$$

Você concorda ?

Quando o argumento de nossa função receber o valor 2, ou seja, quando quisermos saber o valor de  $F(2)$ , a função deve retornar 1. E quando quisermos saber o valor de  $F(1)$ , a função deve retornar 1.

Não devemos calcular usando a fórmula da recursividade, pois seria calculado:

$$F(2) = F(1) + F(0)$$

$$F(1) = F(0) + F(-1)$$

E não existem se deve calcular  $F(0)$  quando mais  $F(1)$ .

Assim, nosso código fica:

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

int fibo(int num)
{
    if(num==1)
        return 0;
    else
        if(num==2)
            return 1;
        else
            return fibo(num-1)+fibo(num-2);
}

int main()
{
    int num;

    cout<<"Termo: ";
    cin>> num;

    cout<<"Igual a: "<<fibo(num)<<endl;
}
```

Teste. Calcule o termo 19 da sequência, ele é 2584. Deu certo?

## Recursão x Iteração

Recursão é uma ferramenta que pode salvar nossa vida, como programador, muitas vezes, pois a ideia por trás do 'uma função chamar ela mesma' é bem simples, útil e pode ser empregada nos mais diversos tipos de problemas que sejam repetitivos, que sigam determinados padrões, como os exemplos acima vistos.

Mas é importante salientar que tudo que é feito com recursão, é possível fazer sem.

Mais especificamente, tudo que é possível fazer com recursão, você pode usar iteração, com [loopings](#).

Quando invocamos uma função, muita coisa ocorre por debaixo dos panos. Muita memória é alocada para os parâmetros e argumentos, o endereço e o código da função são alocados, armazenados e acessados em locais pré-definidos na memória, o local onde ela foi invocada é salvo também, para voltar pra lá quando terminar a execução, e por aí vai.

Isso resulta em uma coisa: recursão é mais lento que iteração. de um modo geral.

Então, por que usar recursão? Simples, alguns problemas, de lógica repetitiva, são beeeem mais simples de serem resolvidos usando recursão.

Embora seja mais lenta e menos eficiente, você pode compensar isso resolvendo/criando um algoritmo em menos tempo, usando recursão. Você pode facilmente descobrir a solução de um problema aplicando recursividade, e as vezes iteração pode ser mais complicado de programar.

O segredo da recursividade é que ela transforma um problema em um problema menor, mas similar.

Em vez de resolver  $\text{func}(n)$ , você só precisa resolver para  $\text{func}(n-1)$ , depois  $\text{func}(n-2)$ .... $\text{func}(3)$ ... $\text{func}(2)$ ...e geralmente pra resolver coisas com argumentos pequenos, nós sabemos resolver facilmente. Recursividade faz isso: transforma um problema em um probleminha, com a exata mesma lógica.

Voltaremos a usar recursão para calcular, por exemplo, caracteres na seção de strings e também com listas.

## Exercício de C++

Crie uma função para calcular o fatorial de um número, usando recursão. Escolha um número bem grande, que leve alguns segundos de sua máquina. Veja quanto tempo demorou pra fazer a operação. Agora crie outra função para calcular o mesmo fatorial, mas usando iteração ([Fatorial com laços em C++](#)), calcule o mesmo número. Anote o tempo.

Que número calculou e quanto tempo cada função levou pra calcular?

# MDC em C++ - Como calcular o Máximo Divisor Comum

Vamos aprender como calcular o MDC de um número, ou seja, seu máximo divisor comum, usando recursão em C++. Primeiro, vamos entender o que é MDC.

## O que é MDC ?

Todo número natural pode ser dividido por outros, resultando em números naturais.

Por exemplo, o 8 pode ser dividido por:

$$8/1 = 8$$

$$8/2 = 4$$

$$8/4 = 2$$

$$8/8 = 1$$

O 11, que é um primo, pode ser dividido por:

$$11/1 = 11$$

$$11/11 = 1$$

Todo número natural tem, pelo menos, dois divisores: o 1 e ele mesmo. Então, dois números naturais quaisquer, tem sempre algum divisor em comum, nem que seja apenas o 1.

É aí que entra o MDC, o **máximo**, queremos calcular o maior divisor comum, o maior divisor desses dois números.

## Como calcular o MDC

Vamos calcular os divisores de 18:

$$18/1 = 18$$

$$18/2 = 9$$

$$18/3 = 6$$

$$18/6 = 3$$

$$18/9 = 2$$

$$18/18 = 1$$

Agora os divisores de 15:

$$15/1 = 15$$

$$15/3 = 5$$

$$15/5 = 3$$

$$15/15 = 1$$

1 e 3 são os números que se repetem, os divisores em comum. E qual o maior deles?

Pimba, é o 3!

## Como calcular o MDC com C++ e recursão

O que vamos fazer é simples...vamos pegar o número maior e calcular o resto da divisão pelo menor.

Se der 0, acabou, o menor é o MDC.

Se não der 0, continuamos com mais alguns cálculos, até o resto da divisão do primeiro pelo segundo ser 0, mas pegamos duas coisas: o menor número e o resto da divisão do maior pelo menor.

Então, fazemos a recursividade, agora passando como argumentos o menor número e o resto da divisão do maior pelo menor.

Nosso código fica assim:

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

int mdc(int num1, int num2)
{
    if(num1%num2 == 0)
        return num2;
    else
        return mdc(num2, num1%num2);
}

int main()
{
    int num1, num2;
```



```
cout<<"Numeros (maior e menor): ";  
cin>> num1 >> num2;  
  
cout<<"MDC: "<<mdc(num1, num2)<<endl;  
}
```

Para entender, matematicamente, a lógica anterior, você deve estudar o [Algoritmo de Euclides](#).

## A função `exit()` do C++

Neste tutorial de nosso **curso C++ Progressivo**, vamos conhecer e aprender a usar a função `exit()`.

### A função `exit()`

De todos os programas que desenvolvemos ao longo de nosso curso, todos eles terminavam após o comando **`return`** da função `main()`.

Quando invocamos uma função, ela salva em memória o local de onde ela foi invocada, para voltar para aquele estágio, após terminar sua execução. Assim, sempre os programas voltavam pra função `main()`, mesmo a gente tendo invocado milhões de outras funções.

Porém, seria interessa se pudéssemos sair, interromper ou terminar a qualquer momento, nossos programas, não concorda?

Isso é possível, basta chamar a função `exit()`. Ela é bem simples.

Primeiro, inclua a biblioteca: *`cstdlib`*

A função está definida dentro dessa biblioteca, por isso precisamos adicionar pra usar.

Depois, basta passar um inteiro como argumento e invocar a função, veja o código:

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

int main()
{
    cout<<"A main() começou..."<<endl;
    cout<<"Chamando a exit()..."<<endl;
    exit(0);
    cout<<"Depois da exit()..."<<endl;
```

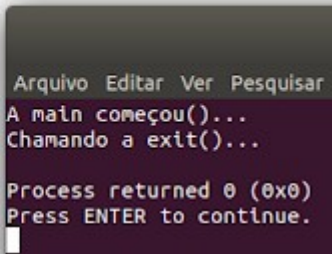
```
    return 0;
}
```

Note que o último cout nem aparece na tela:

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

int main()
{
    cout<<"A main começou()..."<<endl;
    cout<<"Chamando a exit()..."<<endl;
    exit(0);
    cout<<"Depois da exit()..."<<endl;

    return 0;
}
```



Você pode colocar esse exit(0) em qualquer local do programa, mesmo dentro de outra função qualquer, vai interromper e terminar na hora o programa, naquele momento e naquele local.

## Argumentos da função exit()

A função exit tem como parâmetro um número inteiro, apenas.

O argumento que devemos fornecer é o que vai ser retornado para o sistema operacional que executou o seu programa em C++.

Esse argumento é muito importante para avisar para quem chamou determinado programa se ele foi executado e encerrou corretamente. No futuro, você vai programar softwares que chamam outros programas, e precisa se comunicar com eles.

Uma dessas maneiras é através do argumento da exit().

Por exemplo, o 0 é o símbolo universal para: "Tudo ok, terminou tudo certo!"

Por isso que sempre usamos: return 0;, na função main(), entendeu agora? Quando alguém te chamar de um zero, não se ofenda, em programação isso é um elogio.

Existem até duas constantes bastante usadas, para simbolizar:

1. sucesso: EXIT\_SUCCESS
2. problema: EXIT\_FAILURE

Use assim: `exit(EXIT_SUCCESS)` ou `exit(EXIT_FAILURE)`, caso o programa tenha se encerrado da maneira correta ou com algum problema.

Se seu programa jogo terminar por problema na memória, faça: `exit(1)`, então 1 simboliza problema na memória.

Se a internet tiver caído e terminado a partida, o programa se encerra com `exit(2)`, onde 2 simboliza problemas com internet.

E por aí vai...você vai decidir o que cada número do erro significa, para poder saber o que está ocorrendo com determinada aplicação. Deixe o número 0 para simbolizar sucesso, que tudo deu certo na execução do programa.

O número 2112 é o número supremo do universo, respeite e não use ele. Devemos apenas reverenciar o valor 2112.

# Exercícios de Funções

Chegamos ao final de mais uma seção de nosso **curso de C++**, a de funções.

Nos exercícios abaixo, você deve usar todos seus conhecimentos aprendidos até aqui, como o básico, de laços, loopings, testes condicionais e, claro, de função.

Use e abuse das funções. Crie muitas funções.

Cada função deve fazer uma única coisa, da maneira mais simples e direta possível.

Faça com que elas sejam facilmente acopladas, ou seja, deve receber e retornar informações, sempre que possível. Isso permite que você possa usar os códigos de suas funções posteriormente.

Lembre-se: um programa grande nada mais é que uma série de programinhas pequenos.

Suas funções devem ser esses programinhas pequenos, fazendo funcionalidades específicas, ok?

01. Crie um programa que recebe dois lados menores de um triângulo retângulo e uma função retorna o valor da hipotenusa.

02. Crie um programa que recebe os três lados de um triângulo, passa esses valores para uma função que diz se esse triângulo existe ou não (pela condição da existência do triângulo, cada lado deve ser maior que o módulo da subtração dos outros dois lados e deve ser menor que a soma dos outros dois lados)

03. Faça um programa que peça um número inteiro positivo 'n' para o usuário e imprima um quadrado de lado 'n' preenchido de hashtags. Por exemplo, para n=4, deve aparecer na tela:

```
####  
####  
####  
####
```

04. Programe um software que recebe três números, para para uma função e ela retorna o maior deles. Faça outra função que recebe os mesmos números e retorna o menor deles.

05. Um número é dito ser perfeito quando ele é igual a soma de seus divisores. Por exemplo, o seis é perfeito, pois:  $6 = 1 + 2 + 3$   
Programa um software que pede um número ao usuário e diga se ele é perfeito ou não.
06. Crie um software que recebe um número do usuário, passa esse valor para uma função e ela retorna esse número escrito ao inverso. Por exemplo, você deu o valor 1234, então ele vai retornar 4321. Dica: primeiro, crie uma função que conta quantos dígitos tem um número.
07. Faça um programa para lançar uma moeda. Quando chamamos uma função, ela deve retorna cara ou coroa. Em outra função, faça 'n' lançamentos de moedas, 'n' é o valor que o usuário quiser, e mostre a porcentagem de vezes que deu cara e coroa. Se você jogar a moeda 10, 100, 1000, um milhão de vezes...o que tende a acontecer?
08. Crie um dado em C++. Role o dado: ou seja, uma função deve sortear um número aleatório de 1 até 6. Agora, faça com que o dado anterior seja lançado 100 vezes, mil vezes e 1 milhão de vezes. A cada vez que ele rodar, você deve armazenar o valor que ele forneceu, ao final, você mostra quantas vezes cada número foi sorteado. Bate com os resultados da estatística ?
09. Crie um jogo de par ou ímpar. Você deve escolher 0 para par ou 1 para ímpar, em seguida fornece um número. O computador gera um número de 0 até 10, soma os valores e diz quem ganhou, além de mostrar o placar e perguntar se quer jogar mais uma rodada.
10. Aos moldes do jogo par ou ímpar, crie o jogo do Pedra, Papel ou Tesoura, em C++.
11. Crie um jogo onde o computador sorteia um número de 1 até 10, e você tenta adivinhar qual é.
12. Vamos incrementar o jogo anterior? Faça com que o programa diga dizer quantas tentativas você levou para acertar. Faça com o que o computador sorteie um número de 1 até 100. A cada vez que você chutar, ele deve dizer se você chutou abaixo do valor real, acima ou se acertou. Ao final, diz o número de tentativas que você teve e se bateu o record ou não. Ah, ao final de cada rodada, o programa pergunta se você quer jogar novamente ou não, exibindo o record atual.

## Soluções

09.

### Par ou Ímpar em C++: Como programar

Neste tutorial, vamos aprender como programar o jogo do Par ou Ímpar, em C++, onde o usuário vai jogar contra o computador.

### Par ou Ímpar em C++: Código comentado do Jogo

As variáveis 'jogador' e 'computador' vão armazenar, respectivamente, o número de vitórias suas e da máquina, respectivamente. A função placar() serve somente para exibir o placar, usando essas variáveis.

A função par\_impair() serve para perguntar ao jogador se ele quer escolher par (deve digitar 0) ou ímpar (digitar 1), e retorna esse valor.

Na função jogada\_humano(), o usuário vai escolher um número para jogar. Na função jogada\_computador(), a sua máquina vai sortear um número de 0 até 10, usando geração de números aleatórios.

Por fim, a função jogada() vai pegar qual opção o jogador escolheu (par ou ímpar), o número que o jogador lançou e o número que a máquina jogou. Ele vai somar esses números escolhidos e testar se é par (resto da soma deve ser 0) ou ímpar (resto da soma deve ser 1).

Ele avisa o resultado que deu, quem ganhou e incrementa a variável 'jogador' ou 'computador' corretamente, para exibição no placar.

Na main(), tem a variável 'continuar', para decidir se o jogo deve rodar novamente ou não.

A variável 'parimpar' recebe a opção do usuário, se quer PAR ou ÍMPAR.

O número que o jogador escolheu fica na variável 'num\_jogador' e que a máquina sortearou fica na 'num\_comp'.

Agora é só fazer a jogada e exibir o placar.

Para perguntar se o jogador deseja jogar mais uma vez e para garantir que ocorra pelo menos uma jogada, usamos o laço DO WHILE.

## Código do jogo Par ou Ímpar em C++

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

int jogador=0,
    computador=0;

void placar();           // Exibe o placar
int par_impar();         // Jogador diz se quer par ou ímpar
int jogada_humano();     // Humano escolhe numero
int jogada_computador();// Computador escolhe numero
void jogada(int parimpar, int jog, int comp); // Verifica quem ganhou

int main()
{
    int continuar=1,
        parimpar,
        num_jogador,
        num_comp;

    do{
        parimpar = par_impar();
        num_jogador = jogada_humano();
        num_comp = jogada_computador();
        jogada(parimpar, num_jogador, num_comp);

        placar();
        cout<<"\nJogar mais uma vez?"<<endl;
        cout<<"0. Sair"<<endl;
        cout<<"1. Jogar de novo"<<endl;
        cin >> continuar;

    }while(continuar);
    return 0;
}
```



```

int par_impar()
{
    int num;

    cout<<"\nPar ou ímpar? Digite:"<<endl;
    cout<<"0 para par"<<endl;
    cout<<"1 para ímpar"<<endl;
    cin >> num;

    return num;
}

int jogada_humano()
{
    int num;
    cout<<"\nDigite um número de 0 até 10:"<<endl;
    cin >> num;
    return num;
}

int jogada_computador()
{
    unsigned seed = time(0);
    srand(seed);

    return rand()%11;
}

void jogada(int parimpar, int jog, int comp)
{
    cout<<"\nJOGADA: "<<endl;
    cout<<"Humano   = "<<jog<<endl;
    cout<<"Máquina  = "<<comp<<endl;
    cout<<"Soma     = "<<(jog+comp)<<endl;
    cout<<"Resultado = ";

    if( (jog+comp)%2 == 0 )
        cout<<"PAR\n";
    else
        cout<<"ÍMPAR\n";

    if( (jog+comp)%2 == parimpar){

```

}

ponto fraco, que você notou?

## Game em C++: Adivinhe o número sorteado

Crie um jogo onde o computador sorteie um número de 1 até 100. A cada vez que você chutar um número, ele deve dizer se você chutou abaixo do valor real, acima ou se acertou. Ao final, diz o número de tentativas que você teve e se bateu o record ou não. Ah, ao final de cada rodada, o programa pergunta se você quer jogar novamente ou não, exibindo o record atual.

### Código comentado do jogo em C++

Bem, vamos lá, vamos com calma e cuidado, que hoje vamos passar das 100 linhas de código.

Mas não se assuste, um grande programa nada mais é que vários pequenos programinhas divididos, entre outras coisas, entre funções que fazem coisas específicas.

Inicialmente, declaramos a variável *record*, que vai armazenar o número record de tentativas que o usuário acertou em uma rodada. Ela inicia em 0. Já já você vai entender o motivo.

Em seguida, criamos a função *limpa()*, que vai limpar a tela a cada rodada.

A função *gerar()* faz com que o computador gere um número aleatório, de 1 até 100, e retorna ele.

A função *palpite()* vai servir para receber o palpite do usuário. Ela recebe um inteiro, o número de tentativas que o usuário já realizou, pra exibir: "tentativa 1", "tentativa 2", "tentativa 3"...

A função *checa()* vai comparar o número que você chutou com o número gerado pelo computador.

Se você acertar, ele retorna 1, se errar retorna 0.

A função *dica()* vai receber o seu palpite e o número gerado, e vai te dar a dica, se o seu palpite foi maior, menor ou igual ao gerado pelo computador.

A função `encerra()` exibe as informações finais do jogo.

Ela pega o número de tentativas que você realizou, e diz se você bateu o record ou não. Inicialmente, se a variável `record` for igual 0, essa variável vai pegar o número de tentativas que você levou pra acertar, no seu primeiro jogo. Depois, ela avalia se suas tentativas foram menores que o valor de `'record'`, se for, ela diz que você bateu o record e esse é o novo valor da variável `record`. Se não, mostra o record.

Por fim, a função `continuar()` pergunta se você deseja jogar de novo ou não.

Funções explicadas, vamos pra lógica do jogo, na `main()`.

Inicialmente, declaramos 3 variáveis, a que vai armazenar o número de tentativas em cada rodada, a que vai armazenar o número gerado em cada rodada e a que vai armazenar os palpites do usuário.

No primeiro `DO WHILE`, vamos controlar as rodadas que serão jogadas. Ao fim de cada iteração, chamamos a função `continuar()` dentro do `WHILE`, pra saber se devemos começar outra rodada ou não.

Dentro de cada rodada, primeiro limpamos a tela.

Em seguida, zeramos o número de tentativas, afinal, é uma nova rodada.

Exibimos o record atual, através da variável `'record'`. E aviamos que geramos o número a ser adivinhado.

Agora, vamos ficar em um looping: pergunta um chute, diz se errou ou acertou, dá a dica...pede outro chute, avisa se acertou ou não, dá dica...pede outro número...até a pessoa acertar.

Para isso, vamos usar outro laço `DO WHILE`. Esse vai rodar sempre que a pessoa errar. Errou? Roda de novo. Quem controla isso é a função `checa()`.

Dentro desse `DO WHILE` interno, primeiro incrementamos o número de tentativas.

Pegamos o palpite do usuário, então damos a dica (que também avisa quando ganha).

Por fim, vai pra função `checa()`, que retorna 1 se a pessoa acerta (termina o looping) ou 0 se ela erra (roda de novo).

Encerrado esse looping interno, chamamos a função `encerra()`, que vai dizer em quantas tentativas você acertou, se bateu o record ou não.

Depois disso, vai no while externo, que vai perguntar se você deseja jogar de novo ou não.

Simples, não?

## Código do Jogo em C++

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

int record=0;

void limpa();    // Limpa a tela
int gerar();     // Gera o número aleatório
int palpite(int); // Pega o palpite do usuário
int checa(int, int); // Checa se errou ou ganhou
void dica(int, int); // Da a dica se é maior ou menor
void encerra(int); // Exibe informações da partida
int continuar(); // Pergunta se quer jogar de novo

int main()
{
    int tentativas,
        gerado,
        palp;

    do{
        limpa();
        tentativas=0;
        cout<<"Record até o momento: "<< record <<" tentativas!"<<endl;
        gerado = gerar();
        cout<<"\nNúmero sorteado! "<<endl;

        do{
            tentativas++;
```

```

        palp = palpite(tentativas);
        dica(gerado, palp);
    }while(checa(gerado, palp) != 1);

    encerra(tentativas);
}while( continuar() );
return 0;
}
void limpa()
{
    if(system("CLS")) system("clear");
    //int n;
    //for(n=1 ; n<10 ; n++)
    // cout<<endl;
}
int gerar()
{
    unsigned seed = time(0);
    srand(seed);

    return 1+rand()%100;
}

int palpite(int tent)
{
    int palp;
    cout<<"\nTente adivinhar o número de 1 até 100."<<endl;
    cout<<"Tentativa " << tent << ":" << endl;
    cin >> palp;

    if(palp>0 && palp<=100)
        return palp;
    else
        cout<<"Só vale números de 1 até 100"<<endl;
}

int checa(int generado, int palp)
{
    if(gerado == palp)
        return 1;
    else
        return 0;
}

```

```

void dica(int gerado, int palp)
{
    if(palp<gerado)
        cout<<"ERROU! Seu palpite foi MENOR que o número
sorteado!"<<endl;
    else
        if(palp>gerado)
            cout<<"ERROU! Seu palpite foi MAIOR que o número
sorteado!"<<endl;
        else
            cout<<"Ahhhhh muleeeeeque!"<<endl;
}

void encerra(int tent)
{
    cout<<"\nVocê acertou em "<<tent<<" tentativa(s)!"<<endl;

    if(record==0)
        record = tent;

    if(tent<=record){
        cout<<"Parabéns! O record é seu!"<<endl;
        record = tent;
    }
    else
        cout<<"O record ainda é de "<<record<<" tentativa(s)!"<<endl;
}

int continuar()
{
    int cont=1;

    cout<<"\nJogar de novo?"<<endl;
    cout<<"0. Sair"<<endl;
    cout<<"1. De novo!"<<endl;
    cin >> cont;

    return cont;
}

```

# Arrays / Vetores

Nesta seção, iremos dar início ao estudo das estruturas de dados.

Estrutura de dados é um grupo, ou coleção, de informações, com algumas características em comum, cujo objetivo é lidar com uma quantidade qualquer de informações de uma maneira bem simples, fácil e automatizada.

Iniciaremos nossos estudos pelos famosos vetores, também conhecidos como arrays. São idênticos aos usados na linguagem C, baseado em ponteiros, assunto da próxima seção de nossa apostila de C++.

Bons estudos.



# Vetor/Array em C++: O que são? Para que servem?

Neste tutorial introdutório de nossa seção de Vetores (ou Arrays), vamos dar início aos estudos desse importante assunto e ferramenta, que você usará muito, em seus programas de C++.

## O que é um Array (ou vetor) ?

O array, também chamado de vetor, é tipo de estrutura de dados (coleção de dados), do mesmo tipo. Ou seja, é um modo existente de trabalharmos com várias quantidades de variáveis.

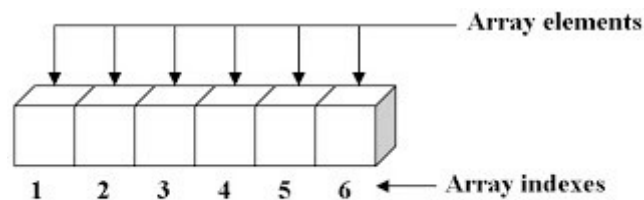
Até o momento, estudamos apenas variáveis únicas, que recebem e armazenam uma única informação, como um número. É um bloquinho de memória reservado para guardar algum dado.

No caso do array, ele é um grupo de variáveis do mesmo tipo. Quando declaramos um vetor, o C++ vai lá e reserva um bloco de memória bem grande, suficiente para caber várias variáveis, todos os endereços são vizinhos.

Assim como as structs e classes, os arrays são estáticos, ou seja, durante a execução possuem o mesmo tamanho sempre.

Inicialmente, vamos estudar os arrays ao estilo da linguagem C, baseado em ponteiros (que iremos estudar em breve). Mais na frente no nosso curso, na seção de STL (Standard Template Library), vamos conhecer os *vectors*, arrays como objetos completos e bem mais versáteis.

Mas antes, precisamos entender um pouco mais sobre os arrays.



One-dimensional array with six elements

Um array de uma dimensão, com 6 elementos (variáveis do mesmo tipo)

## Para que serve um vetor (ou array) ?

Imagine que você quer calcular a média de dois alunos da sua turma. Basta declarar duas variáveis: a e b, e fazer  $(a+b)/2$   
Simples, né?

E caso queira calcular a média de três alunos?  
Hora, é óbvio:  $(a+b+c)/3$

Bacana...e se quiser calcular a média de uma turma de 30 alunos?  
Vai declarar 30 variáveis e fazer:  $(a+b+c+d+e+f+g...)/30$  ?  
Nem tem 30 letras no alfabeto.

É aí que entra o conceito de array.

Vamos simplesmente declarar um vetor de floats, por exemplo, de tamanho 30.

Ou seja, 30 variáveis do tipo float, vizinhas de memória, serão alocadas em sua máquina, de uma vez só, ao criarmos esse array.

Basicamente é para isso que serve um array: vamos aprender a declarar, inicializar, usar e manusear grandes blocos de informações de uma só vez, mexendo com 10, 10, mil ou 1 milhão de variáveis de uma vez só, de maneira bem automatizada e simples, através de arrays.

# Arrays em C++ : Como Declarar, Inicializar, Acessar e Usar

Agora que já aprendemos o que são arrays, para que servem e onde são usados, vamos aprender a trabalhar com eles, declarando, acessando e usando das mais variadas maneiras.

## Como declarar um Array em C++

Assim como as variáveis que usamos até o momento, em nossa apostila, para declarar um vetor, você primeiramente precisa declarar o tipo de dado dele (char, int, double, float etc). Em seguida, precisa dar um nome também.

Agora vem a parte diferente, em array: você precisa dizer o tamanho do seu array, ou seja, quantas variáveis únicas aquele vetor vai conter, e isso é informado entre colchetes [ ].

Vamos declarar algumas variáveis?

- int RG[10]: array com 10 inteiros pra armazenar números de RG's
- float notas[5]: array com 5 números floats, pra armazenar notas
- char nome[100]: array com espaços para 100 caracteres, para armazenar um nome ou texto (também chamado de strings, que estudaremos futuramente nosso curso)

Só isso: tipo, nome do array e quantos elementos ele deve ter.

## Como inicializar um Array

Ao declarar, você já pode de cara inicializar os elementos do array. Para isso, colocamos os valores entre colchetes { }.

Por exemplo, um array de 2 elementos inteiros de valores 21 e 12:

- int rush[2] = { 21, 12 };

Se você quiser inicializar um array de mil elementos, todos com valor 0, basta fazer:

- int num[1000] = {};

Automaticamente o C++ vai preencher todas as variáveis com valor inicial nulo.

Você não precisa nem declarar o número de elementos se for inicializar diretamente, o C++ vai contar quantos elementos você informou e declarar o array com tamanho exato daquilo que está usando, basta deixar os colchetes vazios.

Por exemplo, inicializando uma string (array de caracteres) com o nome do nosso curso e exibir uma mensagem de boas vindas:

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

int main()
{
    char site[] = {"C++ Progressivo"};

    cout << "Bem vindos ao " << site << endl;
}
```

## Como acessar os elementos de um Array

Vamos supor que tenhamos um array de 3 inteiros:

```
int num[3];
```

A primeira variável é: num[0]

A segunda variável é: num[1]

A terceira variável é : num[2]

Note uma coisa essencial: **o primeiro elemento tem SEMPRE índice 0**

O primeiro é num[0] e não num[1].

Você pode declarar um vetor de 10, 100, mil, milhão...o primeiro elemento é acessado usando o índice 0:

Primeiro elemento: num[0]

Segundo elemento: num[1]

n-ésimo elemento : num[n-1]

Ou seja, se seu array tem tamanho X, você acessa os elementos a partir do índice 0, até o índice (X-1), ok ?

## Como usar Arrays em C++

Vamos colocar a mão na massa e ver como se usa arrays em C++? De verdade?

Como vamos tratar com uma grande quantidade de variáveis, praticamente sempre que usamos arrays, usamos também laços para acessar mais rapidamente e de maneira mais dinâmica e flexível, os vetores. Vamos ver na prática.

### ▪ Exemplo 1

Coloque os números 0, 1, 2, 3, ..., 9 em um array de 10 elementos.

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

int main()
{
    int num[10], cont;

    for(cont=0 ; cont<10 ; cont++)
        num[cont] = cont;
}
```

Nossa variável auxiliar é 'cont', e ela recebe valores de 0 até 9.

A variável de índice 0 recebe o valor 0.

A variável de índice 1 recebe o valor 1.

...

A variável de índice 9 recebe o valor 9.

Isso é feito com a linha de comando: num[cont] = cont;

### • Exemplo 2

Coloque os números 1, 2, 3, ..., 10 em um array de 10 elementos, em seguida exiba eles no formato 'Elemento 1', 'Elemento 2', ..., 'Elemento 10'.

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

```

int main()
{
    int num[10], cont;

    for(cont=0 ; cont<10 ; cont++)
        num[cont] = cont+1;

    for(cont=0 ; cont<10 ; cont++)
        cout << "Elemento " << cont+1 << ": " << num[cont] << endl;
}

```

Agora fizemos com um detalhe de diferença:

A variável de índice 0 recebe o valor 1.

A variável de índice 1 recebe o valor 2.

...

A variável de índice 9 recebe o valor 10.

Isso é feito com a linha de comando: num[cont] = cont+1;

Veja bem, nós programadores, contamos a partir do 0. Mas as pessoas normais (?????!), contam a partir do 1. Então, temos que exibir 'elemento 1' como o primeiro, e não elemento 0. Por isso, adicionamos 1 à variável cont,

- **Exemplo 3**

Crie um programa que peça a nota de 5 alunos, armazene essas notas num array, depois exiba elas, bem como a sua média.

```

#include <iostream>
using namespace std;

int main()
{
    float grade[5], sum=0;
    int cont;

    for(cont=0 ; cont<5 ; cont++){
        cout<<"Insira a nota " << cont+1 << ": ";
        cin >> grade[cont];
        sum += grade[cont];
    }
}

```

```

for(cont=0 ; cont<5 ; cont++)
    cout<<"Nota " << cont+1 << ": " << grade[cont] << endl;

sum /= 5;

cout<<"Media: " << sum << endl;

}

```

Vamos armazenar as notas no array 'grade', e a soma das notas na variável 'sum'. Nosso contador é o 'cont'.

Primeiro, usamos um laço FOR para pedir as 5 notas ao usuário. Cada vez que ele digita uma nota, ela é atribuída a um elemento do array, em seguida é somada a variável sum (que deve inicializar em 0).

Depois, exibimos todas as notas digitadas. Por fim, basta dividir a variável 'sum' por 5 e teremos a média das notas digitadas.

Note que o tamanho do código seria o mesmo pra 1 milhão de notas, bastaria mudar o número 5 pra 1000000. Viu como os arrays deixam nossas possibilidades de criar programas bem mais flexíveis?

- **Exemplo 4**

Crie um array de 101 elementos. Em cada elemento do array, armazene o valor do dobro do índice. Ou seja, a variável num[x] deve ter armazenado o valor 2x. Depois, exibe a lista dos 100 números primeiros números pares.

```

#include <iostream>
using namespace std;

int main()
{
    const int ARRAY_SIZE = 101;
    int num[ARRAY_SIZE], cont;

    for(cont=0 ; cont<ARRAY_SIZE ; cont++)
        num[cont] = 2*cont;
}

```

```

for(cont=1 ; cont<ARRAY_SIZE ; cont++)
    cout<<"Dobro de "<<cont<<": "<<num[cont]<<endl;

}

```

Os arrays sempre devem ser inicializados com um valor literal (um número diretamente) ou por uma variável, de preferência constante (const). Aliás, é recomendável você declarar uma constante no começo do programa e ficar usando variável durante o código, fica mais fácil para futuras alterações.

- **Exemplo 5**

Peça para que 6 funcionários de uma empresa digitem seus salários. Em seguida, seu programa deve dizer quanto de imposto de renda cada um deles deve pagar por mês. A taxa é de 15%.

```

#include <iostream>
using namespace std;

int main()
{
    const int ARRAY_SIZE = 6;
    float func[ARRAY_SIZE];
    int cont;

    for(cont=0 ; cont<ARRAY_SIZE ; cont++){
        cout<<"Funcionario "<<cont+1<<": ";
        cin >> func[cont];
    }

    cout<<"Imposto a pagar: "<<endl;
    for(cont=0 ; cont<ARRAY_SIZE ; cont++)
        cout<<"Funcionario "<<cont+1<<": R$"<<func[cont]*0.15<<endl;

}

```



# Probabilidade e Estatística em C++: Jogando Dados

Agora que já aprendemos os conceitos básicos dos [Arrays em C++](#), vamos ver um uso na prática deles, fazendo lançamento de dados e usando conceitos de Probabilidade e Estatística.

## Lançar Dados em C++

O que vamos fazer é bem simples: vamos jogar dados, ou seja, sortear números de 1 até 6.

Vamos fazer isso 100 vezes, 1000 e 100 mil vezes.

Inicialmente, declaramos um array de nome *dice*, de 6 posições, do tipo inteiro, e inicializamos com valor nulo. Vamos usar um inteiro *num* para armazenar o número aleatório gerado.

Em seguida, para gerar os números, vamos utilizar a função `gen()`:

```
int gen()
{
    std::random_device rd;
    std::mt19937 gen_numb(rd());
    std::uniform_int_distribution<> dis(1, 6);

    return dis(gen_numb);
}
```

Usamos uma solução mais sofisticada, pois vamos gerar milhares de números aleatórios, num curtíssimo intervalo de tempo, e para isso a `rand()/srand()` não servem (gerar números seguidamente).

Vamos usar a biblioteca *raindon* e vai retornar um inteiro entre 1 e 6, simulando um dado.

(Referência:

[https://en.cppreference.com/w/cpp/numeric/random/uniform\\_int\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution))

Se sair 1, armazenamos na posição dice[0].

Se sair 2, armazenamos na posição dice[1].

...

Se sair 6, armazenamos na posição dice[5].

Agora, basta fazer: num = gen();

Depois: dice[num-1]++ (pois o array vai de 0 até 5)

E incrementar o índice 'num' do array em uma unidade.

Fazemos os loopings rodar, primeiro gerando os aleatórios, depois exibindo quantas vezes cada número saiu (o número 'num' saiu dice[num] vezes). Não esqueçamos de zerar os valores do array, antes de cada looping de geração de números.

Veja como fica nosso código:

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

int gen()
{
    std::random_device rd;
    std::mt19937 gen_numb(rd());
    std::uniform_int_distribution<> dis(1, 6);

    return dis(gen_numb);
}

int main()
{
    int dice[6] = {},
        num=0, aux;

    cout <<"100 lançamentos de dados:"<<endl;
    for(aux=0 ; aux<100 ; aux++){
        num = gen();
        dice[num-1]++;
    }
    for(aux=0 ; aux<6 ; aux++){
```

```

    cout<<"Numero "<<aux+1<<" saiu: "<<dice[aux]
    <<" ("<<fixed<<setprecision(2)<<100.0*dice[aux]/100<<" %)"<<endl;
}

cout <<"\n1000 lançamentos de dados:"<<endl;
for(aux=0 ; aux<1000 ; aux++){
    num = gen();
    dice[num-1]++;
}
for(aux=0 ; aux<6 ; aux++){
    cout<<"Numero "<<aux+1<<" saiu: "<<dice[aux]
    <<" ("<<fixed<<setprecision(2)<<100.0*dice[aux]/1000<<" %)"<<endl;
}

cout <<"\n10000 lançamentos de dados:"<<endl;
for(aux=0 ; aux<100000 ; aux++){
    num = gen();
    dice[num-1]++;
}
for(aux=0 ; aux<6 ; aux++){
    cout<<"Numero "<<aux+1<<" saiu: "<<dice[aux]
    <<" ("<<fixed<<setprecision(2)<<100.0*dice[aux]/100000<<"
%)<<endl;
}

}

```

Poderíamos ter feito direto com o código: `dice[ gen()-1]++` .E o resultado:

```

100 lançamentos de dados:
Numero 1 saiu: 15 (15.00 %)
Numero 2 saiu: 11 (11.00 %)
Numero 3 saiu: 15 (15.00 %)
Numero 4 saiu: 18 (18.00 %)
Numero 5 saiu: 23 (23.00 %)
Numero 6 saiu: 18 (18.00 %)

1000 lançamentos de dados:
Numero 1 saiu: 208 (20.80 %)
Numero 2 saiu: 204 (20.40 %)
Numero 3 saiu: 164 (16.40 %)
Numero 4 saiu: 184 (18.40 %)
Numero 5 saiu: 188 (18.80 %)
Numero 6 saiu: 152 (15.20 %)

100000 lançamentos de dados:
Numero 1 saiu: 166878 (16.69 %)
Numero 2 saiu: 167335 (16.73 %)
Numero 3 saiu: 166717 (16.67 %)
Numero 4 saiu: 166631 (16.66 %)
Numero 5 saiu: 166282 (16.63 %)
Numero 6 saiu: 167257 (16.73 %)

```

Segundo a teoria das probabilidades, a chance de sair algum número em um dado é de 1 em 6, ou  $1/6$  ou 16,67% de chances.

Note que em 100 lançamentos, o resultado é um pouco flutuante.

A medida que aumentamos o número de sorteios, as estatísticas reais vão se aproximando do resultado da teoria, por volta dos 16,67% , logo nosso método de jogar dados é cada vez mais realista e mais aleatório à medida que aumentamos o número de lançamentos.

# Como achar o Maior e o Menor elemento de um Array

Neste tutorial, vamos fazer um exercício interessante, que envolve arrays e busca. Vamos gerar um array de 1 milhão elementos, com valores aleatórios, e vamos fazer a máquina fazer uma varredura nesse array gigante, em busca do maior e do menor elemento do vetor.

## Fazendo buscas em Arrays

Primeiro, definimos o tamanho de nosso array, que será SIZE, será uma constante de valor 1 milhão.

Em seguida, basta declarar nosso array, de nome 'numb': `numb[SIZE];`

Vamos declarar uma variável auxiliar, a 'count', e outras duas que vão armazenar o valor do maior (highest) e menor valor contido naquele array (lowest).

Vamos primeiro sair em busca do maior elemento.

A lógica é a seguinte: de início, vamos supor que o primeiro elemento do array, o de índice 0, seja o maior. Então, fazemos:

```
highest = numb[0];
```

O que temos que fazer é percorrer, todo o array, a partir do índice 1 e comparar todos os outros elementos com o valor armazenado em 'highest'.

Vamos comparar se o elemento 'numb[1]' é maior que 'highest'.

Vamos comparar se o elemento 'numb[2]' é maior que 'highest'.

Vamos comparar se o elemento 'numb[3]' é maior que 'highest'.

...

Vamos comparar se o elemento 'numb[999999]' é maior que 'highest'.

Essas comparações vamos fazer com um simples teste condicional IF, dentro de um looping FOR que vai percorrer todos os elementos do array gigante:

```
if (numb[count] > highest)
```

Ora, se algum elemento for maior, devemos alterar o valor armazenado em 'highest' para esse novo valor, concorda? Resumindo, basta fazer:  
highest = numb[count];

Nosso código fica assim:

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

int main()
{
    const int SIZE = 1000000;
    int numb[SIZE], count, highest, lowest;
    srand( time(0) );

    for (count=0; count<SIZE; count++ )
        numb[count] = rand();

    highest=numb[0];
    for (count=1; count<SIZE; count++){
        if (numb[count] > highest)
            highest = numb[count] ;
    }

    lowest=numb[0];
    for (count=1; count<SIZE; count++){
        if (numb[count] < lowest)
            lowest = numb[count] ;
    }

    cout<<"Maior: "<<highest<<endl;
    cout<<"Menor: "<<lowest<<endl;
}
```

E para achar o menor valor?  
A lógica é a mesma.

Fazemos com que, de início, o menor valor armazene o valor do primeiro elemento do array:

```
lowest = numb[0];
```

Depois comparamos todos os outros elementos do array com esse 'lowest', checando se os outros elementos do array são MENORES, que lowest, se forem, atualizamos o novo valor de lowest.

Aliás, dá até pra gente fundir esses dois laços FOR com IFs aninhados, em um só laço, veja:

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

int main()
{
    const int SIZE = 1000000;
    int numb[SIZE], count, highest, lowest;
    srand( time(0) );

    for (count=0; count<SIZE; count++ )
        numb[count] = rand();

    highest=numb[0];
    lowest=numb[0];

    for (count=1; count<SIZE; count++){
        if (numb[count] > highest)
            highest = numb[count];

        if (numb[count] < lowest)
            lowest = numb[count];
    }

    cout<<"Maior: "<<highest<<endl;
    cout<<"Menor: "<<lowest<<endl;
}
```

1 milhão de elementos, e achamos o menor e o maior elemento de maneira praticamente instantânea. Poderoso esse esquema de buscar com arrays e laços, não acha? Fantástico o C++.

# Arrays em Funções

Neste tutorial de nossa seção de **Arrays em C++**, vamos aprender como trabalhar com arrays e funções, aprendendo como passar um array como argumento, como retornar e receber um array, como copiar, comparar e alterar arrays, usando funções.

## Arrays como argumentos para Funções

O que diferencia, basicamente, uma variável inteira de um array de inteiros? Vejamos as declarações:

- `int num;`
- `int num[2112];`

Hora, é o par de colchetes, com um número dentro. Concorda? Quando passamos um inteiro para uma função, seu protótipo fica:

- `show(int num);` ou `show(int);`

Então, como você pode suspeitar, para passarmos um array como argumento, você pode fazer, no protótipo:

- `show(int num[]);` ou `show(int []);`

Veja bem, você passa só com o par de colchetes, ok? Sem número dentro. Já no código da função, o cabeçalho é com: `show(int num[]);`

Vamos dar um exemplo de código de um array que declaramos na `main()`, de inteiros, passamos pra função `show()` e ela exibe esses elementos:

Olhe como invocamos a função: `show(num);`  
O nome do array é 'num', ok? Não faça: `show(num[])` pra invocar a função.

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

```
void show(int []);
```

```
int main()
```



```

{
    int num[]={10, 20, 30};

    show(num);

    return 0;
}

void show(int num[])
{
    int count;

    for(count=0 ; count<3 ; count++)
        cout<<"Elemento "<<count+1<<": "<<num[count]<<endl;
}

```

Pegou? Porém, aí tem um problema...a função 'já sabia' de antemão que o array tinha 3 elementos. Mas, ora, as funções não tem bola de cristal. Por isso, é comum e de praxe, passarmos junto com o array, o seu tamanho também, para as funções:

```

#include <iostream>
using namespace std;

void show(int [], int);

int main()
{
    int num[]={10, 20, 30, 40, 50};
    int size=5;
    show(num, size);

    return 0;
}

void show(int num[], int size)
{
    int count;

    for(count=0 ; count<size ; count++)
        cout<<"Elemento "<<count+1<<": "<<num[count]<<endl;
}

```

Quando estudarmos a classe *vector*, em STL, vamos ver tipos mais dinâmicos e poderosos, de arrays, e não precisaremos nos preocupar de informar o tamanho do array, pois será uma informação própria da estrutura que vamos usar.

## Retornar Array: Passagem por referência

Vamos criar um array chamado `num[]`, inserir alguns números. Em seguida, vamos passar para a função `doub()`, que vai dobrar cada elemento do array, e imprimir ele dobrado.

Em seguida, imprimimos, agora na `main()`, novamente o array `num`:

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

void doub(int [], int);

int main()
{
    int num[]={10, 20, 30, 40, 50};
    int size=5, count;

    doub(num, size);

    for(count=0 ; count<size ; count++)
        cout<<num[count]<<" ";

    return 0;
}

void doub(int num[], int size)
{
    int count;

    for(count=0 ; count<size ; count++)
        num[count] *= 2;

    for(count=0 ; count<size ; count++)
        cout<<num[count]<<" ";
```

```
    cout<<endl;  
}
```

O resultado é:

20 40 60 80 100

20 40 60 80 100

Ora, veja só! Você passou um array pra função. Dentro dela, ela dobrou. Quando voltou da função, pra main(), e você imprimiu novamente o array, ele estava dobrado.

Ou seja: sua função mexeu no array.

Embora você tenha passado somente o nome do array, ela alterou o array original. Ou seja: quando passamos um array pra uma função, essa passagem é por **referência**. Quando passamos o nome do array pra função, o C++ passa o endereço real do array. Assim, a função altera aquela posição da memória, diretamente, e não uma cópia do valor (como ocorre na passagem por valor).

Isso ocorre por questões de eficiência, visto que levaria muito tempo pra fazer uma cópia dos arrays passados para as funções (no dia-a-dia, trabalhamos com arrays muuuito grandes mesmo).

Ok? C++ passa o array por referência, não se esqueça!

## Como copiar Arrays

Muitas vezes, queremos fazer algumas coisas com arrays, mas sem alterá-los.

Por exemplo, vamos supor que tenhamos um array 'num' de inteiros, e queiramos um outro array onde cada elemento seja o triplo do valor de cada elemento do array 'num'.

O que podemos fazer é primeiro criar uma cópia de 'num', vamos chamar de 'copy', e pimba, mandamos a 'copy' para uma função que triplica os elementos. Como a passagem é por referência, ela vai alterar os valores da 'copy' e nem vai ficar sabendo da existência de 'num', que fica inalterado.

Para fazer uma função 'copyArray' que faz uma cópia de um array, precisamos passar os dois arrays como argumento para a função, bem como o tamanho deles, que deve ser necessariamente o mesmo. Depois, basta copiar elemento por elemento, com um laço.

Veja como fica nosso código:

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

void copyArray(int [], int [], int);
void triple(int [], int);

int main()
{
    int num[]={10, 20, 30, 40, 50}, copy[5];
    int size=5, count;

    copyArray(num, copy, size);
    triple(copy, size);

    cout<<"Array original: "<<endl;
    for(count=0 ; count<size ; count++)
        cout<<num[count]<<" ";

    cout<<"\nArray triplicado: "<<endl;
    for(count=0 ; count<size ; count++)
        cout<<copy[count]<<" ";

    return 0;
}

void copyArray(int num[], int copy[], int size)
{
    int count;

    for(count=0 ; count<size ; count++)
        copy[count] = num[count];
}

void triple(int copy[], int size)
{

```

```
int count;  
  
for(count=0 ; count<size ; count++)  
    copy[count] *= 3;  
  
}
```

Resultado:

Array original:

10 20 30 40 50

Array triplicado:

30 60 90 120 150

Se você **realmente** quiser preservar o array original 'num' e quiser mesmo garantir que ele não seja alterado, pode declará-lo e usá-lo como sendo do tipo **const**:

No cabeçalho: void copyArray(const int [], int [], int);

Na declaração da função: void copyArray(const int num[], int copy[], int size)  
{...}

No declaração do array: const int num[]={10, 20, 30, 40, 50};

# Ordenar elementos de um Array em C++

Neste tutorial de nosso **curso de C++**, vamos aprender como ordenar os elementos de um Array.

## Ordenar elementos (sorting)

Ordenar, isto é, colocar elementos (como números) em uma determinada ordem (como crescente ou decrescente, por exemplo), é um dos assuntos mais importantes e estudados em computação, devido sua importância.

Na sua escola ou universidade, a lista de alunos é ordenada em ordem alfabética.

As seções eleitorais onde você vai votar, também usam algum tipo de ordem. As contas bancárias, também organizam tudo de acordo com números, bem como os números de uma cartela de loteria.

Quando você for um programador profissional em C++, sua empresa vai pedir para você organizar diversas coisas, em algum tipo de ordem, como nomes, salários, identificações, cargos, tamanho ou peso de produtos, etc etc.

Ou seja, é um assunto **absurdamente** usado, importante demais mesmo, que você vai usar muito mesmo, várias vezes, em sua carreira, e vamos introduzir agora neste tutorial, um pouco do assunto.

## Como ordenar um Array em C++

Vamos usar o [algoritmo Selection Sort](#), para ordenar os elementos do Array. Primeiro, criamos a função `gen()`, que vai gerar números aleatórios entre 1 até 100.

Vamos criar um array de tamanho `size=10` e nome `'num'`.

Vamos preencher, em seguida, cada posição desse array com um número aleatório, e depois imprimimos esse array de valores aleatórios. Próximo passo agora é ordenar esse array,

A lógica é a seguinte: pegamos o primeiro elemento do array, e comparamos com todos os outros. Ou seja, comparamos o elemento 0 com o de índice 1, depois o 0 com o de índice 2...até chegar no 0 com o de índice 9.

Se esses outros elementos forem menores que o de índice 0, invertemos os valores desses dois elementos. Pronto, ao término dessa operação, o menor valor do array estará na posição 0.

Agora vamos colocar o segundo menor valor na posição de índice 1. Para isso, vamos comparar o elemento 1 com o 2, depois o 3, depois o 4...até compararmos o elemento 1 com o elemento 9, o último. Novamente, se esse outro elemento for menor que o de índice 1, invertemos a posição deles.

Ao final desta etapa, o segundo menor valor estará na posição 1. Agora basta fazer isso para os elementos 2, 3,...7 e 8. Na última 'rodada' de comparações, comparamos o elemento 8 com o 9 para decidir qual é maior e qual é menor.

O primeiro looping, usando um laço for, é o que controla o primeiro elemento, e dentro dele vamos comparar com todos os outros índices.

Vamos usar a variável 'prev' para receber o primeiro índice. Ela vai de 0 até 'size-2' (de 0 até 8, ou seja: `for(prev=0; prev < size -1 ; prev++)` ).

O segundo looping, um laço for aninhado, vai comparar o índice 'prev' com todos os outros elementos do array.

O segundo índice vai ser armazenado na variável 'next', ele começa sempre do valor 'prev+1' (quando o primeiro elemento é 0, ela começa do 1...quando o primeiro elemento é o de índice 1, ele começa no índice 2, etc etc...até o primeiro elemento ser o de índice 8 e ela será o índice 9), até 'size-1' (ou seja, `for (next=prev+1 ; next < size ; next++)` ).

Dentro desse laço interno, usamos um teste condicional IF pra saber se o segundo elemento é menor que o primeiro, se for, invertemos os valores desses elementos.

Veja como fica nosso código:

```

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

int gen()
{
    std::random_device rd;
    std::mt19937 gen_numb(rd());
    std::uniform_int_distribution<> dis(1, 100);

    return dis(gen_numb);
}

int main()
{
    int size=10, prev, next, aux;
    int num[size];

    for(aux=0 ; aux<size ; aux++){
        num[aux] = gen();
    }

    cout<<"Array original: "<<endl;
    for(aux=0 ; aux<size ; aux++)
        cout<<num[aux]<<" ";
    cout<<endl;

    // Selection sort algorithm
    for(prev=0 ; prev<size-1 ; prev++)
        for(next=prev+1 ; next<size ; next++){
            aux=num[prev];

            if(num[next]<num[prev]){
                num[prev]=num[next];
                num[next] = aux;
            }
        }

    cout<<"Array Ordenado: "<<endl;
    for(aux=0 ; aux<size ; aux++)
        cout<<num[aux]<<" ";
    cout<<endl;
}

```



```
    return 0;
}
```

## Exercícios de Arrays

01. Faça um algoritmo que ordena os elementos de um array do maior pro menor, ou seja, ordem decrescente.

02. Coloque o código anterior todo dentro de funções: função que preenche o array, função que inverte dois valores, função que faz o selection sort e função que exibe arrays. Use protótipos de função, pra deixar seu código bem profissional.

Solução:

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

int gen();
void fillArray(int [], int);
void showArray(int [], int);
void invert(int &, int &);
void selectionSort(int [], int);

int main()
{
    int size=10;
    int num[size];

    fillArray(num, size);

    cout<<"Array original: "<<endl;
    showArray(num, size);

    selectionSort(num, size);

    cout<<"\n\nArray Ordenado: "<<endl;
    showArray(num, size);

    return 0;
}
```

```

int gen()
{
    std::random_device rd;
    std::mt19937 gen_numb(rd());
    std::uniform_int_distribution<> dis(1, 100);

    return dis(gen_numb);
}

void fillArray(int num[], int size)
{
    for(int aux=0 ; aux<size ; aux++){
        num[aux] = gen();
    }
}

void showArray(int num[], int size)
{
    for(int aux=0 ; aux<size ; aux++)
        cout<<num[aux]<<" ";
}

void invert(int &a, int &b)
{
    int aux;
    aux=a;
    a=b;
    b=aux;
}

void selectionSort(int num[], int size)
{
    int prev, next;

    for(prev=0 ; prev<size-1 ; prev++)
        for(next=prev+1 ; next<size ; next++)
            if(num[next]>num[prev])
                invert(num[prev],num[next]);
}

```

# Matriz em C++ : Array de Arrays

Neste tutorial de nosso **Curso de C++**, vamos aprender o importante conceito de matriz em C++.

## Array de Arrays - O que é? Para que serve?

Até o momento, em nossa seção de *arrays*, criamos arrays de inteiros, float, doubles, char etc.

Ou seja, criamos arrays que armazenam números ou caracteres.

Porém, também é possível armazenar outras coisas em arrays.

Uma coisa curiosa de se armazenar em arrays, são outros arrays.

Por exemplo, imagine que você foi contratado por uma escola para fazer um programa em C++ que vai, dentre outras coisas, armazenar as notas dos alunos. Cada aluno tem, por exemplo, 5 notas diferentes.

Você pode, antes de tudo, raciocinar assim: criar um array para armazenar os alunos.

Por exemplo: `alunos[10]`

Porém, cada aluno tem 5 notas. Então cada aluno desses vai ter um array de notas:

`notas[5]`

Veja bem: temos um array de alunos, cada bloco representa um aluno. E dentro de cada bloco, ou seja, de cada aluno, tem uma espécie de array interno, com as notas de cada aluno. Cada aluno tem seu array de notas, e cada aluno faz parte do array de alunos.

Vamos formalizar isso?

## Como declarar uma Matriz em C++

Chamamos pelo nome de matriz, um array de arrays, ou arrays multidimensionais.

Até o momento, trabalhamos apenas com arrays de uma dimensão, só uma 'linha' de bloquinhos, que formam um array.

Para aumentarmos o número de dimensões de um array, basta irmos acrescentando pares de colchetes, por exemplo:

- Para declarar um array de uma dimensão, fazemos: `float grade[5];`
- Para declarar um array de duas dimensões, fazemos: `float students[10][5];`

Vamos lá. Quando fazemos '`float grade[5]`' queremos dizer: 5 bloquinhos de float.

Quando fazemos: '`float students[10][5]`', queremos dizer: 10 bloquinhos, onde cada bloquinho desses tem um array dentro, de 5 floats.

Dizemos que essa é uma matriz 10x5 (10 linhas e 5 colunas, onde cada linha representa um aluno, e cada coluna representa uma nota diferente).

Para facilitar, vamos imaginar uma matriz 3x3, declaramos assim:

- `int matrix[3][3];`

A expressão '`matrix[3]`' quer dizer: um array de 3 elementos.

O que contém cada elemento? Um inteiro? Um float? Um char? Não, cada elemento é um outro array, de tamanho [3].

Veja como fica a representação dessa matriz:

|         | Coluna 0                  | Coluna 1                  | Coluna 2                  |
|---------|---------------------------|---------------------------|---------------------------|
| Linha 0 | <code>Matrix[0][0]</code> | <code>Matrix[0][1]</code> | <code>Matrix[0][2]</code> |
| Linha 1 | <code>Matrix[1][0]</code> | <code>Matrix[1][1]</code> | <code>Matrix[1][2]</code> |
| Linha 2 | <code>Matrix[2][0]</code> | <code>Matrix[2][1]</code> | <code>Matrix[2][2]</code> |

O primeiro elemento de nossa matriz é: `matrix[0]`

Ele é um array, de 3 elementos, que são representados por:

Primeiro elemento: `matrix[0][0]`

Segundo elemento: `matrix[0][1]`

Terceiro elemento : `matrix[0][2]`

O segundo elemento de nossa matriz é: `matrix[1]`

Ele é um array, de 3 elementos, que são representados por:

Primeiro elemento: `matrix[1][0]`

Segundo elemento: `matrix[1][1]`

Terceiro elemento : `matrix[1][2]`

O terceiro elemento de nossa matriz é: `matrix[2]`

Ele é um array, de 3 elementos, que são representados por:

Primeiro elemento: `matrix[2][0]`

Segundo elemento: `matrix[2][1]`

Terceiro elemento : `matrix[2][2]`

Ou seja, para declaramos uma matriz de 'i' linhas e 'j' colunas, fazemos:

- `tipo matrix[i][j];`

Se você já estudou matriz e determinantes, na escola, deve se lembrar de como se usa uma matriz, como sinalizar cada elemento etc. Aqui, a única diferença é que a contagem começa do índice 0, ao invés de 1.

## Como inicializar uma Matriz em C++

Vamos criar uma matriz para armazenar as notas de 3 alunos, onde cada aluno tem duas notas.

Ou seja, teremos uma matriz de 3 linhas (uma para cada aluno) e 2 colunas (cada coluna representa uma nota). Declaramos essa matriz, então:

- `float grade[3][2];`

Vamos agora inicializar uma matriz 3x3, por exemplo, com as notas:

```
float grade[3][2] = { {9.5, 9.1},  
                     {8.8, 8.4},  
                     {10.0, 10.0} };
```

O array de notas do primeiro aluno é: {9.5, 9.1}

O array de notas do segundo aluno é: {8.8, 8.4}

O array de notas do terceiro aluno é : {10.0, 10.0}

Por exemplo, qual a primeira nota do segundo aluno? Segundo aluno é o aluno 1, primeira nota é a de índice 0, então essa nota está armazenada em: `grade[1][0] = 8.8`

Já a segunda nota do segundo aluno é: `grade[1][1] = 8.4`

Simples, não?

# Matrizes em Funções

Neste tutorial de nossa **apostila de C++**, vamos aprender como trabalhar com matrizes e funções, aprendendo a declarar, invocar e usar estes dois importantes tópicos, conjuntamente.

## Como Passar Matriz para Função

No estudo de arrays, vimos que existem algumas maneiras de declarar o cabeçalho de funções com arrays como parâmetros, da seguinte maneira:

- tipo func(tipo \*array);
- tipo func(tipo array[tamanho]);
- tipo func(tipo array[]);

Ou seja, basta passar o ponteiro (\*array - estudaremos mais na frente) ou o só com o par de colchetes em aberto (array[]).

No caso de arrays bidimensionais, precisamos especificar o número de **colunas** da matriz que estamos enviando:

- void func(int arr[][COLUMN]);

Veja bem, o número de linhas não é obrigatório (até podemos passar), mas o número de colunas, sim.

Vamos declarar e inicializar uma matriz 2x2 e em seguida enviara para a função show(), que vai simplesmente exibir ela na forma de uma tabela, veja como fica nosso código:

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

```
void show(int arr[][2], int row)
{
    for (int i=0 ; i<row ; i++){
        for(int j=0 ; j<2 ; j++)
            cout<<arr[i][j]<<" ";
        cout<<endl;
    }
}
```

```
int main()
{
    int arr[][2]={ {1,2}, {3,4} };
    show(arr, 2);
    return 0;
}
```

Note algumas coisas importantes, no código acima.

Primeiro, passamos a matriz `arr[][2]` para a função, com o número de colunas.

Mas e o número de linhas? Como a função vai saber o tanto de linhas, para imprimir a tabela ?

Ela não sabe, por isso passamos outro parâmetro na função, o inteiro 'row'.

## Matriz em C++: Passagem por referência

Toda passagem de dados para funções que envolvem arrays, é por referência. Sempre.

Ou seja, passou um array pra uma função? Ela vai acessar diretamente o array e seus dados, direto na memória. Não é uma cópia que vai pra função, não é passagem por valor, ok?

Isso quer dizer uma coisa: cuidado! As funções podem modificar seus arrays, lembre-se sempre disso.

O código abaixo introduz a função 'init', que vai receber um array e inicializar cada elemento dele, perguntando pro usuário:

```
#include <iostream>
using namespace std;
const int COLS = 3;
const int ROWS = 3;

void init(int arr[][COLS], int ROWS)
{
    for(int i=0 ; i<ROWS ; i++)
        for(int j=0 ; j<COLS ; j++){
            cout << "matrix["<<i+1<<"["<<j+1<<"]: ";
            cin >> arr[i][j];
        }
}
```



```

    }
}

void show(int arr[][COLS], int ROWS)
{
    for (int i=0 ; i<ROWS ; i++){
        for(int j=0 ; j<COLS ; j++){
            cout<<arr[i][j]<<" ";
        }
        cout<<endl;
    }
}

int main()
{
    int arr[ROWS][COLS];
    init(arr, ROWS);
    show(arr, ROWS);
    return 0;
}

```

Para melhorar a organização, já definimos as linhas (ROWS) e colunas (COLS) como variáveis globais do tipo constantes, para não ter perigo de ninguém, em algum local do código, alterar seus valores. Isso deixa o código mais claro e seguro, para manutenção.

```

matrix[1][1]: 1
matrix[1][2]: 2
matrix[1][3]: 3
matrix[2][1]: 4
matrix[2][2]: 5
matrix[2][3]: 6
matrix[3][1]: 7
matrix[3][2]: 8
matrix[3][3]: 9
1 2 3
4 5 6
7 8 9

```

## Exercício de Matriz em C++

Crie uma matriz 4x4, onde cada linha representa as notas de um aluno, e cada coluna é uma matéria diferente. Você deve criar uma função que vai preenchendo as notas dos alunos, uma por uma, assinalando qual é a matéria e qual é o aluno.

Em seguida, seu programa deve exibir, de maneira organizada, as notas de cada aluno, bem como a média de cada um, a média da turma para cada matéria, e a média geral, de todos alunos de todas as notas.

## Exercícios de Arrays em C++

Usando seus conhecimentos até o momento (básico, laços, testes condicionais, funções e principalmente arrays), resolva os seguintes exercícios em linguagem C++ e coloque nos comentários seu código.

### Questões de Arrays em C++

01. Escreva um programa que vai receber e armazenar 10 números em um array. Em seguida, você deve criar duas funções que vão achar o maior e o menor valor desse array.
02. Você foi contratado por uma emissora de TV para fazer um software em C++ pra trabalhar com os dados da chuva em sua cidade. Seu vetor de 7 posições deve armazenar quanto choveu em cada dia. Em seguida, deve dizer que dia choveu mais, o dia que choveu menos, o tanto que choveu na semana e a média, para exibir na TV.
03. Crie um vetor de 10 posições, para armazenar a média de 10 alunos. Seu programa, em seguida, deve ir em cada uma das notas e verificar se o aluno foi aprovado (nota maior que 7), foi reprovado (nota menor que 5) ou vai ficar pra recuperação (nota entre 5 e 7).
04. Crie um programa gerador de números da Mega-Sena. Ou seja, crie um array de 6 posições e em cada posição sorteie um número de 1 até 60. Ah, seus números não podem ser repetidos.
05. Faça um programa em que vamos fornecer 10 números para um array, incluindo valores repetidos. Seu código deve vasculhar esse array para criar um novo array que não tenha números repetidos.
06. Programe o jogo da velha, em C++. Dois jogadores, no seu computador, devem poder jogar. Cada jogador diz uma linha e um número, no tabuleiro 3x3, para jogar (só pode jogar em locais livres do tabuleiro). Ao final de cada partida, ele deve dizer quem ganhou ou se deu empate, bem como o placar do jogo, e se querem jogar mais uma partida ou encerrar o programa.

## Jogo da velha em C++

07. Em um tabuleiro NxN (você escolhe N), o computador deve sortear N posições nesse tabuleiro, sem que o usuário saiba. Essas posições são as dos navios. Agora você deve dar chutes (tiros na água) pra tentar acertar os navios. A cada tiro que você der, o programa deve dizer se você acertou ou se errou, e se errou, deve dizer quantos navios tem ali ao redor daquele ponto.

Mais exercícios, com soluções, sobre arrays em C++:

<https://www.w3resource.com/cpp-exercises/array/index.php>

# Jogo da Velha em C++

Neste tutorial, vamos te ensinar do mais absoluto zero, como programar o jogo da velha em C++, um game super bacana que você pode usar para jogar com algum amigo.

```
Jogador 1 ganhou!  
  
X |  | O  
---  
  | X |  
---  
O |  | X  
  
Placar:  
1 x 0
```

## Lógica do Jogo da Velha em C++

Vamos usar uma matriz, 3x3, de inteiros, para representar as jogadas. Se a casa tiver o número 0, ela está vazia. Se tiver o número 1 é porque o jogador 1 jogou lá, e se tiver -1, é porque o jogador 2 jogou naquela posição.

Vamos explicar, por partes, as funções.

A função **init()**, inicializa o tabuleiro, colocando 0 em todas as casas.

A função **show()** vai exibir o tabuleiro, imprimir o tabuleiro bonitinho, com barras, underlines, espaçamentos corretos e, se tem X ou O em cada casa. Essa última parte quem faz é a função **printBlock()**.

A função **printBlock()** vai em cada casa do tabuleiro, se tiver 0 lá, ela retorna um espaço vazio ' '.

Se tiver o número 1 lá, ela imprime X. Se tiver -1 lá, é porque o jogador 2 jogou nessa casa, e lá vai ter um O.

A função **playMove()** realiza uma jogada, ou seja, pede a linha e a coluna que o jogador vai jogar.

Lembrando que ele vai digitar valores de 1 até 3, e o array vai de 0 até 2, por isso temos que subtrair uma unidade dos dados digitados pelo usuário (row-- e col--). Também devemos checar se ela está ocupada (board[row][col]), e se o usuário digitou valores corretos (entre 1 e 3). Se tudo estiver ok, ela preenche com 1 ou -1 a casa que o jogador escolheu.

A função **checkContinue()** vai verificar se ainda tem espaço em branco, ou seja, se ainda é possível realizar alguma jogada. Se tiver espaço em branco, retorna 1, se não tiver, retorna 0.

A função **checkWin()** vai checar se alguém ganhou. Para isso, basta somar cada linha, cada coluna e cada diagonal, se alguma dessas fileiras tiver soma 3, o jogador 1 ganhou e retornamos 1. Se a soma for -3, o jogador 2 ganhou, e retornamos -1. Se ninguém tiver ganhado, até aquele momento, retornamos 0.

Agora vamos pra função **main()**. Ela cria o tabuleiro 3x3, a variável *cont* (que vai perguntar se o jogador vai querer jogar de novo ou não, as variáveis *player1* e *player2* (que vão armazenar o placar do jogo), e a variável *result*, que armazena a informação de quem ganhou o jogo.

O jogo vai ocorrer dentro de um DO WHILE.  
Primeiro, inicializamos o tabuleiro com valores 0.

Agora vamos fazer uma partida, para isso, chamamos a função **game()**.

A função **game()** é onde ocorre a magia, o desenrolar do jogo.  
Primeiro, declaramos a variável *turn*, que vai dizer se é o jogador 1 ou 2 que vai jogar. A cada jogada, devemos incrementar ela em uma unidade (turn++), para mudar a vez para o próximo jogador.

Após dizer de quem é a vez, vamos fazer a jogada, através da função *playMove()*. Note que passamos a informação de quem é o jogador, através da expressão *turn%2*. Se ela der 0, é a vez do jogador 1 (e coloca o número 1, representado pelo X no tabuleiro), se der 1, é a vez do jogador (e preenchemos a matriz com o número -1, que vai ser representado pelo caractere O no tabuleiro).

Em seguida, devemos checar se o tabuleiro ainda tem posições vazias, pra jogar. Se tiver, armazenamos o número 1 na variável *cont*, senão, armazenamos o valor 0.

Também devemos checar se após essa jogada, alguém ganhou. Se o jogador 1 tiver ganhado, armazenamos 1 na variável *win*. Se o jogador 2 tiver ganhado, armazenamos o valor -1. Se ninguém tiver ganhado, armazenamos o valor 0.

Pronto. Uma jogada terminou. Vai ter uma próxima jogada? Isso depende. Se ainda tiver espaço vazio (*cont*=1) e ninguém tiver ganhado (*win*=0, o mesmo que *!win*), vai ter outra jogada.

Se não tiver mais espaço vazio (*cont*=0) ou alguém tiver ganhado (*win*=1 ou *win*=-1), não vai ter próxima jogada, e sai do laço DO WHILE.

Após o laço, vamos verificar o que ocorreu. Se *win*=1, o jogador 1 ganhou e a função *game* retorna 1.

Se *win*=-1, o jogador 2 ganhou e retornamos 2. Se nenhum tiver ganhado, é porque o tabuleiro tá cheio (*cont*=0) e deu empate, então retornamos 0.

Esse retorno, vai lá pra função *main()*, na variável *result*. Ele pega o resultado, e manda pra função ***scoreboard()***, junto com os valores de *player1* e *player2*, ela é responsável por controlar o placar.

Após a exibição do placar, perguntamos se vai querer jogar novamente o jogo ou sair.

Se digitar 1 (continuar), a variável *cont* fica com valor 1, o laço WHILE dá verdadeiro, e outra partida será iniciada. Senão, se for 0, o laço termina junto com o jogo.

Bacana, não é?

Pessoal, o código abaixo tem quase 200 linhas. Podem existir erros, sem dúvidas.

Se encontrarem algum, ou alguma melhoria, por favor escreva nos comentários, ok?

## Código do jogo da Velha em C++

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

void init(int board[][3]);      // Initializes the board with 0's
char printBlock(int block);    // Prints each square of the board
void show(int board[][3]);     // Show the board
void playMove(int board[][3], int); // Play one move
int checkContinue(int *board[3]); // Check if there is still white space
int checkWin(int *board[3]);    // Check if anyone won
int game(int board[][3]);      // PLayer an entire game
void scoreboard(int, int &, int &); // Show the scoreboard

int main()
{
    int board[3][3];

    int cont=0, player1=0, player2=0, result;
    do{
        init(board);
        result = game(board);
        show(board);
        scoreboard(result, player1, player2);

        cout<<"\n Outra partida?"<<endl;
        cout<<"0. Sair"<<endl;
        cout<<"1. Jogar de novo"<<endl;
        cin >> cont;
    }while(cont);

    return 0;
}

void init(int board[][3])
{
    for(int i=0; i<3; i++)
        for(int j=0; j<3; j++)
            board[i][j]=0;
```



```
}
```

```
char printBlock(int block)
```

```
{
```

```
    if(block==0)
```

```
        return ' ';
```

```
    else if(block==1)
```

```
        return 'X';
```

```
    else
```

```
        return 'O';
```

```
}
```

```
void show(int board[][3])
```

```
{
```

```
    cout<<endl;
```

```
    for(int row=0 ; row<3 ; row++){
```

```
        cout<<" "<< printBlock(board[row][0]) <<" |";
```

```
        cout<<" "<< printBlock(board[row][1]) <<" |";
```

```
        cout<<" "<< printBlock(board[row][2]) <<endl;
```

```
        if(row!=2){
```

```
            cout<<" ____ _\n"<<endl;
```

```
        }
```

```
    }
```

```
}
```

```
void playMove(int board[][3], int player)
```

```
{
```

```
    int row, col, check;
```

```
    do{
```

```
        cout<<"Linha: ";
```

```
        cin >>row;
```

```
        cout<<"Coluna: ";
```

```
        cin >> col;
```

```
        row--; col--;
```

```
        check = board[row][col] || row<0 || row>2 || col<0 || col>2;
```

```
        if(check)
```

```
            cout<<"Essa casa não está vazia ou fora do intervalo 3x3"<<endl;
```

```
    }while(check);
```

```

    if(player==0)
        board[row][col]=1;
    else
        board[row][col]=-1;
}

int checkContinue(int board[][3])
{
    for(int i=0 ; i<3 ; i++)
        for(int j=0 ; j<3 ; j++)
            if(board[i][j]==0)
                return 1;
    return 0;
}

int checkWin(int board[][3])
{
    int row, col, sum;

    // Adding the lines
    for(row=0 ; row<3 ; row++){
        sum=0;

        for(col=0 ; col<3 ; col++)
            sum += board[row][col];

        if(sum==3)
            return 1;
        if(sum==-3)
            return -1;
    }

    // Adding the columns
    for(col=0 ; col<3 ; col++){
        sum=0;

        for(row=0 ; row<3 ; row++)
            sum += board[row][col];

        if(sum==3)
            return 1;
        if(sum==-3)
            return -1;
    }
}

```

```

}

// Adding the diagonals
sum=0;
for(row=0 ; row<3 ; row++)
    sum += board[row][row];
if(sum==3)
    return 1;
if(sum==-3)
    return -1;

sum=board[0][2]+board[1][1]+board[2][0];
if(sum==3)
    return 1;
if(sum==-3)
    return -1;

return 0;
}

int game(int board[][3])
{
    int turn=0, cont, win;

    do{
        show(board);
        cout<<"Jogador "<<1+turn%2<<endl;
        playMove(board, turn%2);
        turn++;

        cont=checkContinue(board);
        win = checkWin(board);
    }while(cont && !win);

    if(win==1){
        cout<<"Jogador 1 ganhou!\n"<<endl;
        return 1;
    }else
        if(win==-1){
            cout<<"Jogador 2 ganhou!\n"<<endl;
            return 2;
        }else
            cout<<"Empate\n"<<endl;
}

```

```

    return 0;
}

void scoreboard(int result, int &player1, int &player2)
{
    if(result==1)
        player1++;
    if(result==2)
        player2++;

    cout<<"Placar: "<<endl;
    cout<<player1<<" x "<<player2<<endl;
}

```

## Desafio em C++

Crie uma versão modificada deste jogo, que pergunta ao usuário se ele deseja jogar com um amigo ou contra o computador. Se ele escolher jogar contra a máquina, faça com que seu código escolha alguma casa aleatória, quando for a vez da máquina.

# Ponteiros

Nesta seção, vamos aprender a dominar uma das ferramentas que mais fazem a linguagem C++ ser tão absurdamente poderosa, os ponteiros.

Com eles, vamos ter total domínio sobre cada kilobyte de memória de nossa máquina, de uma maneira bem simples e versátil.

# Ponteiros em C++ - O que são e Endereços de Memória (operador &)

Neste tutorial introdutório de nossa seção sobre **ponteiros em C++**, vamos aprender o básico deste importante conceito.

## Ponteiros - O que são? Para que servem? Onde vivem? De que se alimentam ?

Ponteiro nada mais é que uma variável, assim como **int** é, como **float**, **double**, **char** etc.

E como toda variável, ela serve para armazenar um tipo especial de dado. No caso dos ponteiros, essas variáveis servem para armazenar endereços de memória.

Vamos usar essas variáveis para apontar para determinados locais de memória, e trabalhar em cima disso.

Por exemplo, se fazemos:

- `int count = 1;`

A variável **count** está armazenada em algum local da memória, e lá existe o valor 1.

Poderíamos usar um ponteiro para *apontar* para o endereço desta variável. Dentro do ponteiro não tem o valor 1, e sim um endereço de memória, que aponta para um local onde o 1 está armazenado.

Quando fazemos uma declaração como esta, estamos separando uma determinada quantidade de kiloBytes na memória para armazenar aquele valor.

## Endereços de memória: operador &

Ainda usando a variável de exemplo, onde usamos **count**, o C++ substitui por seu valor, 1.

Por exemplo:

```
cout << count+count;
```

Ele vai imprimir 1+1, ou seja, 2.

Porém, é possível trabalhar com o endereço, o espaço na memória, onde esse número 1 está alocado, ao invés de se trabalhar com o valor que ele armazena, para isso basta usar o símbolo & antes da variável:

- &count

Por exemplo, o código abaixo declara uma variável, armazena um valor. Depois, imprime o valor da variável, o tanto de memória que foi reservada (através da função **sizeof**, que retorna o número de kBytes usados) e por fim, mostra seu endereço, usando o operador &:

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

int main()
{
    int count=1;

    cout<<"Valor da variável : "<< count << endl;
    cout<<"Tamanho de kb      : "<< sizeof(count) << endl;
    cout<<"Endereço de memória: "<< &count <<endl;

    return 0;
}
```

Veja, em nossa máquina, a variável **int** usa 4 KB e foi armazenada no endereço:

```
Valor da variável : 1
Tamanho de kb      : 4
Endereço de memória: 0x7fffb13b95e4
```

E na máquina de vocês?

Agora um desafio: quantos KB são usados para armazenar um endereço de memória ?

Escrevam nos comentários.

E o que esse número esquisito, de endereço de memória, tem a ver com ponteiro?

Muito simples: do mesmo jeito que as variáveis **int** armazenam inteiros, as **char** armazenam caracteres etc, os ponteiros são variáveis que armazenam esse troço aí: endereços de memória.

Ou seja, o ponteiro vai apontar para um local da memória, e é possível fazer coisas incríveis com essa ideia super simples, como alocação dinâmica de memória, que veremos mais adiante em nosso **curso de C++**.



# Ponteiros: Como declarar, Inicializar e Usar

Agora que já aprendemos [o que são ponteiros e endereços de memória](#), vamos colocar a mão na massa, criando e usando ponteiros em C++.

## Como Declarar Ponteiros em C++: \*

Assim como qualquer outro tipo de variável, para usar um ponteiro, precisamos declarar o tipo de dado ponteiro.

E ele é feito da seguinte maneira:

- tipo \*nome;

Por exemplo:

- int \*number;

Isso quer dizer que criamos um ponteiro (pois colocamos o asterisco na declaração), que vai apontar para um endereço de memória que armazena valores do tipo int.

Cuidado para não confundir: não é porque tem um **int** escrito que *number* é um inteiro. É um ponteiro pra um **int**, ok?

Para evitar confusão, muitos programadores preferem escrever:

- float\* value;

Ou seja, não é uma variável do tipo float, é um ponteiro para um float (é *float\** e não *float*).

Rode o exemplo de código abaixo:

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

```
int main()
{
    int *number;
```

```
cout << number << endl;

return 0;
}
```

O resultado é 0, que é o símbolo de NULL ou ponteiro vazio, em C++. Aliás, é até uma boa prática inicializar com 0 ou NULL, ponteiros.

Vamos aprender como apontar o ponteiro para um local mais específico.

## Como inicializar Ponteiros em C++

Bom, vamos lá.

Primeiro, vamos criar e inicializar uma variável do tipo **double**:

- double pi = 3.14;

Agora, vamos declarar um ponteiro que aponta para uma variável do tipo double:

- double \*pointer;

Para inicializar um ponteiro precisamos fornecer um endereço de memória. Que tal o endereço de memória da variável *pi*? Ele é obtido assim: &pi

Então, declarando e inicializando o ponteiro:

- double \*pointer = &pi;

Prontinho, nosso ponteiro aponta pra uma variável.

Vejam o resultado do código:

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

int main()
{
    double pi = 3.14;
    double *pointer = &pi;

    cout << "Valor da variavel pi  : " << pi << endl;
    cout << "Endereço da variavel pi: " << pointer << endl;
```

```
    return 0;
}
```

## Conteúdo de um ponteiro: **\*pointer**

Dizemos que a variável *pi* referencia diretamente o valor 3.14

Já a variável *pointer* referencia indiretamente o valor 3.14, pois aponta para o local que ele está armazenado.

Se usarmos a variável *pointer*, estamos trabalhando com endereço de memória.

Para lidar com o valor que está armazenado naquele endereço, usamos:

*\*pointer*

Ou seja, um asterisco antes do nome do ponteiro.

Veja:

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

int main()
{
    double pi = 3.14;
    double *pointer;
    pointer = &pi;

    cout << "Valor da variavel pi : " << *pointer << endl;
    cout << "Endereço da variavel pi: " << pointer << endl;

    return 0;
}
```

Ou seja, tanto faz fazer isso:

```
cout << pi;
```

Ou isso:

```
cout << *pointer;
```

É a mesma coisa, pois: `pointer = &pi`

Ou seja, dá no mesmo, pois o ponteiro aponta pra variável.

## Como usar ponteiros em C++

E qual o propósito disso?

Simple: podemos também alterar o valor das variáveis, através de seus ponteiros.

Veja no exemplo abaixo, inicialmente a variável tem valor 1.

Em seguida, através de um ponteiro que aponta para ela, alteramos para valor 2:

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

int main()
{
    int number = 1;
    int *pointer;
    pointer = &number;

    cout << "Valor inicial de number: " << number << endl;
    *pointer = 2;
    cout << "Novo valor de number : " << number << endl;

    return 0;
}
```

No exemplo anterior, tanto faz trabalhar com 'number' diretamente ou com \*pointer.

O asterisco, nesse caso, é chamado de operador indireto.

No exemplo abaixo, fazemos com que um mesmo ponteiro aponte (em momentos diferentes), para variáveis diferentes e altere seus valores:

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

int main()
{
    int a=1, b=1, c=1;
    int *pointer;

    cout << "a+b+c: " << (a+b+c) << endl;
```

```
cout <<"Incrementando a, b e c"<<endl;
```

```
pointer = &a;  
(*pointer)++;
```

```
pointer = &b;  
(*pointer)++;
```

```
pointer = &c;  
(*pointer)++;
```

```
cout <<"a+b+c: " << (a+b+c) << endl;
```

```
return 0;
```

```
}
```

Vejam só, alteramos o valor de 3 variáveis, sem atuar diretamente nelas, apenas com um ponteiro.

Note que se fizemos: `*pointer++`, vai dar erro, pois o operador `++` tem precedência sobre o asterisco, ou seja, o `++` vai agir antes na variável 'pointer', que o operador indireto `*`.

# Ponteiros em Funções em C++

Neste tutorial de nossa **apostila de C++**, vamos aprender como trabalhar com ponteiros em funções.

Antes, vamos relembrar um pouco do que aprendemos na seção de Funções.

## Passagem por Valor e Variável de Referência

Rode o seguinte programa, que eleva ao quadrado, um número digitado pelo usuário.

Ele mostra o número digitado, ele ao quadrado e depois o valor original, novamente:

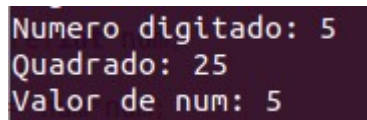
```
#include <iostream>
using namespace std;

int square(int num)
{
    num = num*num;
    cout << "Quadrado: " << num << endl;
}

int main()
{
    int num;
    cout << "Digite um número: ";
    cin >> num;

    cout << "Numero digitado: " << num << endl;
    square(num);
    cout << "Valor de num: " << num << endl;

    return 0;
}
```



```
Numero digitado: 5
Quadrado: 25
Valor de num: 5
```

Veja que passamos a variável 'num' para a função, e lá dentro ela é elevada ao quadrado. Mostramos esse valor dentro da função, e depois mostramos novamente quando a chamada da função é terminada.

Ela não mudou o valor armazenado em 'num'.

Isso ocorre porque quando passamos uma variável pra uma função, a função faz uma cópia de seu valor. Não vai usar a variável diretamente, e sim o seu valor apenas, por isso a variável original não é alterada.

Para dar acesso diretamente à variável, podemos usar as variáveis de referência.

Que é basicamente passar o nome da variável, com o símbolo & na frente, veja como fica:

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

int square(int &num)
{
    num = num*num;
    cout << "Quadrado: " << num << endl;
}

int main()
{
    int num;
    cout << "Digite um número: ";
    cin >> num;

    cout << "Numero digitado: " << num << endl;
    square(num);
    cout << "Valor de num: " << num << endl;

    return 0;
}
```

Agora veja que o valor de 'num' foi alterado dentro da função square():

```
Numero digitado: 5
Quadrado: 25
Valor de num: 25
```

Isso aconteceu porque usamos a variável de referência &num ao invés de apenas num.

E agora que você já estudou ponteiros e endereços de memória, sabe que ao usar o &, estamos lidando diretamente com endereços de memória. Ou seja, a função agora vai ter acesso ao endereço de memória da variável num e vai alterar esse bloco de memória diretamente.

## Como usar Ponteiros em Funções no C++

Outra maneira de alterar o valor de uma variável, passando pra uma função, é usando ponteiros. De fato, é importante saber usar esta técnica, pois ela é bem útil em funções que lidam com Strings e em várias bibliotecas do C++, por exemplo.

Vamos criar uma função, chamada doub(), que vai dobrar o valor que ela receber, usando ponteiros:

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

int doub(int*);

int main()
{
    int num;
    cout << "Digite um número: ";
    cin >> num;

    cout << "Numero digitado: " << num << endl;
    doub(&num);
    cout << "Valor de num: " << num << endl;

    return 0;
}

int doub(int *ptr_num)
{
    (*ptr_num) = (*ptr_num) * 2;
    cout << "Dobro: " << *ptr_num << endl;
}
```



}

O parâmetro dela é: `int*`

Ou seja, ela espera um ponteiro do tipo `int`. Se ela espera um ponteiro, temos que passar como argumento um...endereço de memória! Por isso, fizemos:

```
doub(&num);
```

Lembrem-se: ponteiro é uma variável que armazena um endereço de memória!

Dentro da função, `'ptr_num'` é um ponteiro. Ponteiro para o endereço de memória da variável `'num'` lá da função `main()`.

Queremos dobrar o valor para qual ele aponta. Ora, como se acessa o valor armazenado no local para qual o ponteiro aponta? Usando asterisco:

```
*ptr_num
```

Por isso, para dobrar o valor da variável `'num'`, através de ponteiro `'ptr_num'` que aponta para ela, basta fazer:

```
*ptr_num = *ptr_num * 2;
```

Para não se confundir com os operadores, pode fazer assim que fica mais organizado:

```
(*ptr_num) = (*ptr_num) * 2;
```

## O poder dos ponteiros

Uma grande vantagem de trabalhar com ponteiros, é que eles podem lidar de maneira mais 'global', digamos assim, com as variáveis para qual apontam.

Por exemplo, note que sempre pedimos um número ao usuário dentro da função `main()`, e a partir daí mandamos esse valor ou seu endereço para as mais diversas funções.

Usando ponteiros, podemos pegar valores do usuário dentro de uma função, como a `getNum()`, veja:

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

```

void getNum(int *);
int doub(int*);

int main()
{
    int num;
    getNum(&num);
    cout <<"Numero digitado: " << num << endl;

    doub(&num);

    cout <<"Valor de num: " << num << endl;

    return 0;
}

void getNum(int *ptr_num)
{
    cout <<"Digite um número: ";
    cin >> *ptr_num;
}

int doub(int *ptr_num)
{
    (*ptr_num) = (*ptr_num) * 2;
    cout << "Dobro: " << *ptr_num <<endl;
}

```

Declaramos a variável 'num', e passamos seu endereço para a variável doub(), esse endereço vai ficar armazenado dentro do ponteiro ptr\_num. Agora, queremos alterar o valor de ptr\_num, então basta usar: \*ptr\_num

E prontinho, magicamente, a variável que foi declarada lá na função main() foi alterada dentro da função getNum().

Poderosos, esses ponteiros, não?

# Ponteiros, Arrays e Aritmética

Dando continuidade ao estudo dos [ponteiros](#), vamos ver a importante relação deles com os [arrays](#), bem como aprender a manipular aritmeticamente os ponteiros, com operações matemáticas.

## Arrays e Ponteiros, Ponteiros e Arrays

Por curiosidade, vamos declarar um array de inteiros, de nome 'numbers', inicializar e depois simplesmente imprimir o valor '\*numbers':

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

int main()
{
    int numbers[]={1, 2, 3, 2112};

    cout << *numbers << endl;

    return 0;
}
```

Olha que interessante o resultado:

A screenshot of a code editor showing the same C++ code as above. The code is color-coded: #include is green, using namespace std; is blue, int main() is blue, { is purple, int numbers[]={1, 2, 3, 2112}; is purple, cout << \*numbers << endl; is green, return 0; is blue, and } is purple. A dark overlay is visible on the right side of the code editor, showing the text 'Arquivo Edit' and the number '1'.

Ou seja: o nome do array funciona como um ponteiro.  
E para onde ele aponta? Para o primeiro elemento do array.

Assim, sempre que tivermos um array de nome: arr  
Se usarmos o nome da variável do array, ela vai se comportar como um array que aponta para: arr[0]

E para apontar para os demais membros do array?

Lembre-se da seguinte regra:

- $\text{arr}[\text{indice}] = *(\text{arr} + \text{indice})$

Ou seja:

$\text{arr}[0]$  pode ser referenciado por  $*(\text{arr} + 0)$

$\text{arr}[1]$  pode ser referenciado por  $*(\text{arr} + 1)$

$\text{arr}[2]$  pode ser referenciado por  $*(\text{arr} + 2)$

...

$\text{arr}[n]$  pode ser referenciado por  $*(\text{arr} + n)$

Podemos fazer um ponteiro de nome 'ptr' apontar para o primeiro elemento de um array das seguintes formas:

1. `int *ptr = numbers;`

2. `int *ptr = &numbers[0];`

Vamos imprimir todo um array, usando apenas um ponteiro:

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int numbers[6]={1, 2, 3, 4, 5, 2112},
```

```
    *ptr = numbers;
```

```
    for(int aux=0 ; aux<6 ; aux++)
```

```
        cout << *(ptr+aux) << endl;
```

```
    return 0;
```

```
}
```

## Aritmética dos Ponteiros

No exemplo de código C++ anterior, você viu que fizemos várias vezes uma operação de adição com ponteiros: `ptr+aux`, onde `aux` é uma variável inteira que foi de 0 até 5, para percorrer o array.

Poderíamos ter feito o mesmo programa com o operador `++`, veja:

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

int main()
{
    int numbers[6]={1, 2, 3, 4, 5, 2112},
        *ptr = numbers;

    for(int aux=0 ; aux<6 ; aux++){
        cout << *ptr << endl;
        ptr++;
    }

    return 0;
}
```

Ou seja, cada vez que adicionamos uma unidade ao ponteiro (`++`), ele aponta para o próximo elemento do array. Podemos fazer o inverso, apontar para o último elemento do array e irmos decrementando:

```
#include <iostream>
using namespace std;
int main()
{
    int numbers[6]={1, 2, 3, 4, 5, 2112},
        *ptr = &numbers[5];

    for(int aux=5 ; aux>=0 ; aux--){
        cout << *ptr << endl;
        ptr--;
    }

    return 0;
}
```

Poderíamos usar também os operadores -= ou +=

A lógica é a seguinte.

Quando apontamos o ponteiro para o array, ele vai apontar para o endereço de memória do primeiro elemento. Vamos supor que seja o bloco de memória 1000.

Quando fazemos: ptr++

Ele não incrementa o endereço de memória em uma unidade, ele não vai pra 1. Ele vai para: 1000 + sizeof(int)

Como o array é de inteiros, cada bloco do array ocupa 4 Kbytes, logo ptr vai apontar para o endereço 1004. Depois para o bloco 1008, depois 1012... Se o array fosse de double, ele ia apontar para os endereços: 1000, 1008, 1016, 1024... a cada vez que incrementássemos, pois a variável double ocupa 8 Kbytes.

Veja como fica o código do programa que imprime apenas os elementos de índice par, do array (0, 2 e 4):

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

int main()
{
    int numbers[6]={1, 2, 3, 4, 5, 2112},
        *ptr = numbers;

    for(int aux=0 ; aux<3 ; aux++){
        cout << *ptr << endl;
        ptr += 2;
    }

    return 0;
}
```

Essa é a lógica e matemática dos ponteiros. Por isso é tão importante definir o tipo de ponteiro (int, char, double...), pois os ponteiros apontam para blocos inteiros de memória, o tamanho desses blocos variam de acordo com o tipo de dado.

# Comparação de Ponteiros e Ponteiro para Constantes

Neste tutorial, vamos aprender como fazer comparação entre ponteiros bem como usar variáveis constantes com eles.

## Comparação entre ponteiros em C++

Do mesmo jeito que podemos comparar duas variáveis quaisquer (como `int` e `double`), também podemos comparar ponteiros.

E usando os mesmo operadores: `>`, `>=`, `<`, `<=`, `==` e `!=`

Dizemos, por exemplo, que um ponteiro é maior que outro, quando, por exemplo, seu endereço de memória em um array aponta para uma variável cujo índice é maior que outro.

Por exemplo:

```
int *ptr1 = array[0];  
int *ptr2 = array[1];
```

Então, a comparação: `ptr2 > ptr1` vai resultar em um resultado verdadeiro.  
Já: `ptr1 == ptr2` vai resultar em um resultado falso, pois apontam para endereços de memória diferente.

Ou seja, ponteiros são variáveis que armazenam endereços de memória. Então, ao comparar dois ponteiros, estamos comparando dois endereços de memória, e não os valores para qual eles apontam, ok ?

## Ponteiros e *const* C++

Sempre que usamos a palavra-chave *const*, estamos dizendo ao compilador que o valor armazenado naquela variável **não** deve ser alterado.

Muitas vezes, queremos passar uma variável como informação, mas não queremos que ela seja modificada de maneira alguma, nesses casos, é importante usar a keyword *const*.

Até o momento, usamos ponteiros não-constantes que apontam para variáveis não-constantes, ou seja, podemos mudar até o valor da variável via ponteiro que aponta para ela.

Você não pode, por exemplo, passar um ponteiro, que não é constante, para uma função que espera uma variável constante como argumento. Veja:

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

int main()
{
    const double pi = 3.14;
    const double *ptr = &pi;

    cout << *ptr << endl;

    return 0;
}
```

Para apontarmos para uma variável constante (pi), tivemos que definir um ponteiro constante (const double \*).

Tente tirar o 'const' da declaração do ponteiro, e veja o que acontece.

Poderemos, porém, ter um ponteiro constante que aponta para uma variável que não é constante:

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

int main()
{
    double pi = 3.14;
    double * const ptr = &pi;

    cout << *ptr << endl;

    return 0;
}
```



A diferença é que esse ponteiro vai apontar SEMPRE para o mesmo endereço de memória. Você pode até alterar o valor para qual ele aponta. Mas jamais vai alterar o endereço para qual ele aponta. Ponteiros do tipo constante devem ser inicializados ao serem declarados.

Por fim, poderemos ter um ponteiro constante, que aponta para uma variável constante:

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

int main()
{
    double pi = 3.14;
    const double *const ptr = &pi;

    cout << *ptr << endl;

    return 0;
}
```

Note que, neste caso, não podemos alterar o endereço do ponteiro (não podemos fazer `ptr = &outra_variavel`) e nem podemos alterar o valor armazenado no local que o ponteiro aponta (`*ptr = 21.12`)

Esses casos de constante e ponteiros, bem como de comparação entre ponteiros, é muito usado no estudo de strings, por exemplo.

# Alocação Dinâmica de Memória e o Operador **new** e **delete**

Neste tutorial, vamos aprender uma nova maneira de declarar variáveis, usando o operador **new** e ponteiros, em C++.

## Alocação Dinâmica de Memória

Vamos supor que você vai fazer um software para a multinacional que você trabalha.

Ela tem 1000 empregados.

Então, você vai declarar, dentre várias coisas, mil variáveis do tipo `int` para armazenar a idade de cada pessoa, mil variáveis do tipo `double` para armazenar o salário de cada um, mil strings para armazenar o nome de cada um, etc, etc.

Porém, sua empresa cresceu, e agora tem 2 mil funcionários. E agora, volta lá no código e muda tudo?

Até o momento, em nosso **curso de C++**, declaramos valores fixos, valores exatos de blocos de memória, mas isso nem sempre é interessante, no mundo da computação. Muitas vezes você não sabe quantas variáveis vai precisar usar, logo não sabe quantas variáveis vai precisar declarar.

Por exemplo, se você for criar uma nova rede social, quanto espaço de memória vai precisar para armazenar os dados dos usuários? Ué, não sabe...afinal, pode ter mil ou 1 milhão de usuários, quem sabe.

O que você tem que fazer é, então, alocar memória **dinamicamente**. Ou seja, vai alocar a medida que for precisando.

## Ponteiros e o Operador **new**

Vamos supor que você queira alocar espaço para uma variável do tipo `double`.

Você vai pedir, via código C++, para o computador que ele ache um bloco de memória na máquina que seja possível alocar para você, que esteja livre.

Ele então vai lá vasculhar no seu sistema, acha um bloco e te devolve o endereço inicial desse bloco, o endereço do primeiro byte. Ora, se ele te devolve um endereço, o que você precisa para guardar esse tipo de informação?

Sim, nosso amado tipo de ponteiro.

Esse pedido de alocação é feito com o operador **new**.

Veja um exemplo de código que alocamos um bloco de memória para uma variável do tipo `double`:

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

```
int main()
{
    double *ptr;
    ptr = new double;
    *ptr = 3.14;

    cout << *ptr << endl;

    return 0;
}
```

Quando digitamos 'new double', a máquina devolve um endereço de memória, o primeiro do bloco reservado para o `double`, e armazena esse endereço no ponteiro 'ptr', que obviamente deve ser do tipo `double` também (ou seja, aponta para o endereço inicial de um bloco de 8 Bytes, espaço necessário para um `double`).

Veja, não definimos nenhuma variável puramente do tipo double. Temos apenas um ponteiro que aponta para um bloco que tem um valor do tipo double.

Podemos, inclusive alterar esse valor armazenado, usando \*ptr:

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

int main()
{
    double *ptr;
    ptr = new double;
    cin >> *ptr;

    cout << *ptr << endl;

    return 0;
}
```

Tudo que poderíamos fazer com uma variável do tipo double, podemos fazer com esse ponteiro.

Podemos também alocar arrays inteiros, com o operador new, veja:

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

int main()
{
    int *ptr;
    ptr = new int[10];

    for(int aux=0 ; aux<10 ; aux++){
        ptr[aux] = aux+1;
    }

    for(int aux=0 ; aux<10 ; aux++){
        cout << ptr[aux] << endl;
    }
}
```

```
    return 0;  
}
```

No código acima, alocamos um array de 10 inteiros, e preenchemos com os números de 1 até 10.

Ou seja, no primeiro exemplo, ptr 'virou' um double. Nesse exemplo de cima, ptr 'virou' um array de inteiros.

Flexíveis esses ponteiros, não são?

## Liberando espaços de memória - Operador **delete**

Outra vantagem de usar alocação de memória dinamicamente, é a possibilidade de liberar também essa memória que foi alocada, bastando usar o operador **delete**.

No primeiro exemplo, desalocamos o bloco de memória assim:

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    double *ptr;  
    ptr = new double;  
    cin >> *ptr;  
  
    cout << *ptr << endl;  
  
    delete ptr;  
  
    return 0;  
}
```

No caso dele ser um array, colocamos o par de colchetes [ ] antes do nome do ponteiro:

```

#include <iostream>
using namespace std;

int main()
{
    int *ptr;
    ptr = new int[10];

    for(int aux=0 ; aux<10 ; aux++){
        ptr[aux] = aux+1;
    }

    for(int aux=0 ; aux<10 ; aux++){
        cout << ptr[aux] << endl;
    }

    delete [] ptr;

    return 0;
}

```

Isso pode parecer bobagem hoje em dia, se imaginar seu PC com vários e vários Terabytes, não é preciso ficar liberando memória alocada, durante a execução de um programa.

Porém, esse conhecimento e prática é essencial em sistemas críticos que consomem muita memória, como em um jogo de altíssimo nível ou em um sistema operacional, super eficiente, não pode deixar tudo alocado pra sempre, tem que ir liberando aos poucos.

Já em sistemas com pouco espaço de memória (como seu relógio digital, o painel de uma geladeira ou o timer de seu microondas), é vital ir desalocando e liberando memória, pois esse recurso é super escasso.

Estudaremos com mais detalhes o operador new quando estudarmos Classes e Objetos, em Programação Orientada à Objetos.

# Programação Orientada a Objetos

Neste seção, estudaremos o que faz o C++ ser diferente da linguagem C: sua capacidade de usar a programação orientada a objetos!

# Orientação a Objetos em C++: Introdução

Antes de entrarmos em detalhes sobre um novo paradigma de programação (a orientação a objetos), vamos entender um pouco o que vínhamos fazer e o que vamos fazer de diferente.

## Programação Funcional

Até o momento, usamos apenas um estilo de programação: a funcional, também chamada de procedimental. Como o nome diz, é um método que usa procedimentos, ou como chamamos: funções.

Basicamente, ele é um roteiro, um script de procedimentos e comandos. Ou seja, a gente só usa variáveis e funções. Vamos pegar um exemplo.

Vamos supor que você queira calcular a média aritmética de dois números, em C++. Necessariamente, você vai ter que ter duas variáveis pra armazenar os valores e um cálculo da média. Pronto, tem aí um 'roteiro' do que deve ser feito.

O que você pode fazer de diferente, é colocar o cálculo da média em uma função, para que ela possa ser invocada indefinidamente:

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

float average(float num1, float num2)
{
    return (num1+num2)/2;
}

int main()
{
    float num1, num2;

    cout<<"Numero 1: ";
    cin >> num1;

    cout<<"Numero 2: ";
    cin >> num2;

    cout<<"Media: "<< average(num1,num2) << endl;
```



```
    return 0;  
}
```

Veja como este programa é apenas um procedimento, começa rodando do começo da main() e vai até o final dele. Sempre. Tudo que fizemos foi sempre assim. Tem começo (geralmente declaração de variáveis), meio (chamando funções para fazer diversas coisas) e um fim (exibindo resultados).

O objetivo da programação funcional é criar funções que façam coisas. Pode parecer simples e bobo né? Mas coisas incríveis foram feitas usando isso. O Kernel do Linux, por exemplo, não usa C++, apenas C, ou seja, não tem orientação a objetos, apenas programação procedural.

Com o passar dos anos e décadas, os softwares foram ficando cada vez mais e mais e mais, e mais um pouco, complexos. E alguns problemas foram surgindo.

## Programação Orientada a Objetos

Até o momento, em nossos programas, qualquer função poderia trabalhar em cima de qualquer dado. Isso com o tempo virou um problema de segurança. Seria ideal se determinadas funções pudessem atuar somente em cima de alguns dados.

Por exemplo, em programação funcional, as funções que trabalham com os dados da tesouraria de uma empresa, poderiam trabalhar com quaisquer dados, como os dos funcionários. Mas seria mais bacana se a tesouraria tivesse suas próprias funções e para trabalhar com os dados dos funcionários, tivessem funções próprias também. E por questão de segurança, nenhum pudesse mexer nas coisas dos outros.

Outro problema: você designa uma função pra receber um inteiro. Sem querer, alguém usa essa função e manda um float. Se fizer esse teste, vai dar erro, vai sair resposta errada e pode até simplesmente fechar o programa rodando. Imagina 'fechar' o programa de um avião, em pleno voo? Não dá né.

Daí que veio a bendita e linda POO: Programação Orientada a Objetos. Ela resolve esse problemas, e de uma maneira incrivelmente simples e fácil. Seu segredo é: ela passa a trabalhar com um treco chamado **objeto**.

Cada objeto vai ter suas próprias variáveis e somente algumas funções podem ver atuar e ver ele.

Se tem um objeto do tipo Som no seu jogo, ele vai ter variáveis, características e funções específicas atuando nele. Os objetos do tipo Personagem, vão ter variáveis, características e funções específicas pra eles. Uma função que mexe com Som não pode atuar num objeto do tipo Personagem. E, nossa, isso evita muuuuuitos bugs e potenciais problemas.

Isso de cada objeto ter seus dados e procedimentos, é o chamado encapsulamento, base da lógica da POO. É como se existisse código específico para cada 'coisa'. Uma função só vê os dados daquela 'coisa' e só pode atuar naquela 'coisa'. Você pode, inclusive, esconder informações, dizer claramente: 'Ei, essa variável aqui, que armazena a senha, só pode ser vista aqui dentro do servidor, ela é inacessível para usuários de fora'. E isso traz uma segurança incrível.

Se você está desenvolvendo um game em C++ e usar orientação a objetos, você vai criar dados e procedimentos que só vão atuar e fazer sentido na parte da Lógica. Vai criar informações e funções que só vão ser visíveis e só vão atuar nos Graphics, vai criar coisas específicas para o Cenário (que nem são visíveis fora desse escopo). Você encapsula, você divide, você organiza as coisas...tá captando a ideia da POO ?

Não é mais aquele emaranhado de funções e variáveis que todo mundo pode ver e usar. Uma variável declarada, geralmente só deve ser usada pela função X(). Mas a função Y() pode sim ver e atuar nessa variável, isso é um erro, um problema, e seria interessante se isso fosse naturalmente impossível, se a própria linguagem fizesse essa separação. E é essa separação que a Orientação a Objetos faz.

Se você faz um sistema Web, você quer que apenas algumas funções sejam disponíveis pros usuários, como a função Exibe(), que vai mostrar as notas de um aluno de EAD. Mas se tiver usado programação funcional, o danado do aluno pode criar um código que chama a função Altera(), pra mudar suas notas, hackear o sistema. Ué, ele pode muito bem 'chutar' o nome das funções, e vai que acerta...

Com programação orientada a objetos, podemos deixar bem claro: "Ei sistema, somente essa função e essas variáveis podem ser acessadas pelos alunos". Quando ele tentar invadir o sistema, chamando outras funções que não fazem parte daquele 'objeto' (as notas dele, por exemplo), ele vai ser sumariamente bloqueado.

Bacana essa orientação a objetos, né? Mas vamos deixar de lero-lero, ir pro próximo tutorial e começar a aprender de vez como usar essa bruxaria.

## **Fontes de estudo**

[Paradigmas de Programação](#)

[Programação Procedural](#)

# O que são Classes e Objetos em C++

Neste tutorial de nosso **curso de C++**, vamos aprender de uma maneira bem coloquial e simples, o que são classes e objetos, a base da programação orientada a objetos.

## O que é uma Classe

Classe nada mais é que uma planta, um molde, um diagrama.

Por exemplo, vamos imaginar uma Classe de um carro.

Essa classe vai descrever todos os detalhes desse carro: qual seu tamanho, número de portas, potência do motor, se é manual ou automático, se tem teto solar ou não, seus utilitários etc e etc.

É basicamente isso, é uma 'forma', que explica como são os...objetos.

Pra fazer um carro, precisamos dos detalhes da Classe carro para criar um carro. Esse carro vai ser um objeto.

Sim, pra explicar classe, temos que falar de objeto...

## O que é um Objeto

...e pra falar de Objeto, precisamos falar de Classe. Não dá pra separar nem explicar bem uma das duas coisas, fica algo 'vago', e odeio essas explicações técnicas e vagas.

Tudo que você conhece é um objeto. Você é um objeto, você mora em um objeto, você come objetos, se locomove em objetos, está acessando esta página/apostila através de um objeto, aliás, esse curso é um Objeto.

Tudo é um objeto.

Entenda objeto como as coisas reais, do dia-a-dia, que existem de fato. E classe, como algo abstrato.

# Classe e seus Objetos

Uma boa maneira de explicar algo, é através de exemplos reais. Então vamos lá.

Vamos supor que você concluiu o curso **C++ Progressivo** e virou um baita programador C++, com um excelente salário, e decidiu comprar um carro.

Você não vai chegar na concessionária e falar: "Ei, moço, quero um carro".

E o vendedor não vai falar: "Ok, aqui está um carro".

Nem você vai responder: "Obrigado, agora tenho um carro"

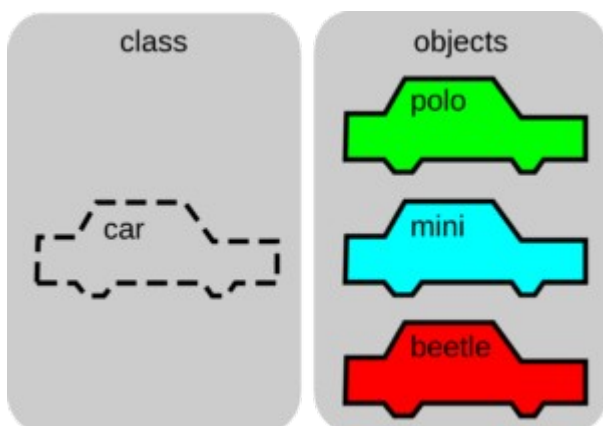
Quando você vai comprar um carro, você vai querer um Gol, um Uno, um Corolla ou uma BMW (caso seja programador C++). Ou seja, você vai comprar algo específico. Esse carro específico é um objeto. Um objeto da classe Carro.

A classe Carro vai dizer: "Esses objetos tem x portas, motor tal, câmbio, potência tal, cor tal...", ou seja, as características comuns que todos carros tem. Todos os carros tem portas e motor, por exemplo.

Um fusquinha tem duas portas e motor fraquinho. Seu Audi tem 4 portas e uma baita potência. Mas ambos tem determinado número de portas e um valor exato de potência.

Ou seja, um Fusca, um Palio, uma Mercedes...são todos objetos da classe Carro.

Captou?



## Atributos e Funções de Classes e Objetos

Vamos para mais um exemplo. Você não conhece e nem lida com um 'humano'. Você lida com seu pai, com sua mãe, com seus amigos...ou seja, com humanos 'específicos'.

Vamos criar a classe Human. Quais as características de um humano? Ué, ele tem nome, idade, altura, peso, coração, pulmão etc etc.

Chamamos esses detalhes de atributos, seus detalhes, suas informações.

Essa classe também vai 'fazer' algumas coisas, como: respirar, bater o coração, andar, dormir...ou seja, a classe Human, além de ter características (valores, números), também vai ter algumas ações...essas ações são funções.

Funções específicas que só existem na classe Human. Afinal, não tem uma função Respirar() na classe Car, nem tem uma função PassarMarcha() na classe Human.

Ou seja, cada Classe tem suas características (atributos) e ações específicas que ocorrem lá (funções). Isso que define uma classe: atributos e funções.

O bacana da programação orientada a objetos, é que as funções do Car só podem atuar nos objetos do tipo Car, e as ações da classe Human só vão poder atuar em cima de objetos da classe Human.

## Instanciar (criar) Objetos a partir de uma Classe

Quando você vai criar algo, esse algo foi criado a partir de uma classe. Ou seja, dizemos que o objeto foi instanciado a partir de uma classe.

Por exemplo, vamos supor que você foi contratado por uma empresa pra trabalhar no setor de RH. De início, você vai criar uma classe chamada Funcionario.

Quais as características de um Funcionario ? Ué, ele tem nome, cargo, salário, número de identificação...

Quando alguém novo for contratado, você precisa criar um objeto específico daquele novo funcionário. Dizemos que você vai *instanciar* um objeto da classe Funcionario.

Esse objeto tem nome "Francisco Moreira", idade (32), salário (R\$ 10.000,00 - afinal é um programador C++), número de identificação (2112)...

Ou seja, objetos são trechos reais, baseados em trechos abstratos (classes).

A partir do próximo tutorial, vamos aprender de verdade como programar classes, como instanciar objetos, como definir suas características e ações.

# Como criar uma Classe e Objetos em C++

Agora que já temos uma boa noção sobre o que é o paradigma de programação Orientação a Objetos, bem como os conceitos de Classe e Objeto (de maneira mais abstrata), vamos colocar a mão na massa e fazer código! Criar as tais classes e objetos de verdade.

## Como criar uma Classe em C++

Inicialmente, o conceito de codificação de classe lembra muito o de *structs*. No caso, vamos usar a palavra-chave **class** para criar uma classe, da seguinte maneira:

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

class MinhaClasse
{
    //Informações
    //da sua
    //Classe
};

int main()
{
    return 0;
}
```

Ou seja, escrevemos *class*, escolhemos um nome pra nossa classe, abrimos chaves e lá dentro colocamos todas as informações sobre ela. Não esqueça de colocar o ponto-e-vírgula ao final do escopo da classe!

Prontinho, a classe *MinhaClasse* foi criada! Note também que ela não está dentro da função *main()*, e sim no mesmo nível dela.



## Definindo membros de uma Classe: Variáveis e Funções

Vamos criar uma classe de verdade? Vamos criar uma classe que vai criar quadrados. Nossa classe se chama Quadrado.

Qual um dado importante de um quadrado? Ué, seu lado. Logo, ele vai ter uma variável que armazena seu lado.

Qual outra 'coisa' importante de um quadrado? Sua área. Vamos criar uma função que calcula a área desse quadrado. Nossa classe fica assim:

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

```
class Quadrado
{
    double lado;
    double area();
};
```

```
double Quadrado::area()
{
    return lado*lado;
}
```

```
int main()
{
    return 0;
}
```

Pronto, ali tem a declaração da variável 'lado' bem como o cabeçalho da função area(). E essa função, onde vamos declarar? Ela foi declarada após o escopo da classe.

Até o momento, em nosso curso, se quiséssemos declarar uma função area() em nossos programas, faríamos:

```
double area()
{
    return lado*lado;
}
```

E essa função é visível e acessível a todo nosso programa, correto?  
E se quisermos criar uma função que é de uma Classe e deve ser visível e acessível somente por aquela classe?

Fazemos:

```
double Quadrado::area()
{
    return lado*lado;
}
```

Entendeu? Coloca "NomeDaClasse::" antes do nome da função, isso diz que aquela função é da classe NomeDaClasse, e somente ela tem acesso e pode usar tal função!

## Como criar e instanciar Objetos em C++

Ok, nossa classe tá bonitinha e feita! Agora vamos criar objetos a partir dela!

Se quiser criar uma informação do tipo inteira, você faz:

```
int num;
int idade;
int RG;
```

Se quiser criar uma informação do tipo string, você faz:

```
string nome;
string endereco;
```

E por ai vai. Se quiser criar uma informação do tipo 'MinhaClasse', você faz:  
MinhaClasse exemplo;

Por exemplo, para criar um objeto da classe Quadrado, você deve fazer:

```
Quadrado q1;
Quadrado quadradin;
```

Vamos chamar de 'q' o nosso objeto, criamos ele assim:

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

class Quadrado
{
    double lado;
```

```

    double area();
};

double Quadrado::area()
{
    return lado*lado;
}

int main()
{
    Quadrado q;

    return 0;
}

```

Prontinho. Agora existe um quadrado de verdade, de nome 'q' do tipo Quadrado, que tem membros 'lado' e 'area()'. Bacana, né?

Quando criamos um objeto, dizemos que estamos **instanciando** um objeto q da classe Quadrado. É uma nova 'instância' da classe.

Você pode inclusive criar outro objeto, ou milhões, basta dar nomes diferentes a eles: q1, q2, ..., q2112...cada um terá suas características: seus valores de 'lado' e suas áreas. Embora sejam do mesmo 'tipo' Quadrado (ou seja, todos tem lado e area() em comum), cada um tem seus valores específicos.

É que nem nós, pessoas...todo mundo tem nome, coração, capacidade pulmonar, idade...mas cada um tem seus valores específicos (nome específico, idade específica, condicionamento físico específico).

Compreendeu melhor agora, o conceito de Classe e Objeto, em C++ ?

No próximo tutorial vamos aprender como acessar, alterar e usar esses membros da classe Quadrado.

## Acessando membros de Classes e Objetos: **private** e **public**

No tutorial passado de nosso curso, aprendemos [como criar classes e objetos em C++](#). Agora, vamos ver como usar esses conhecimentos, acessando os membros (variáveis e funções) de um objeto/classe.

### Acessando membros de Classes e Objetos: .

Ao final de nosso tutorial anterior, tínhamos o código que criava uma classe Quadrado e o objeto 'q'. Faça o seguinte, rode esse código. Obviamente, não aconteceu nada. Definimos uma classe, instanciamos um objeto, e ficou por isso mesmo.

Vamos aprender agora como acessar esses membros dos objetos, através do operador ponto: .

Veja que temos o objeto 'q'. Ele tem a variável 'lado' e a função 'area', que podem ser acessadas:

- q.lado;
- q.area();

Basta colocar o nome do objeto, seguido do operador ponto e o nome do membro, definido na classe, igual fazemos com as structs. Vamos criar um programa que define um valor para o lado, e em seguida chama a função area() para imprimir o valor da área desse quadrado:

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

```
class Quadrado
{
    double lado;
    double area();
};
```

```
double Quadrado::area()
{
    return lado*lado;
}
```

```
int main()
```

```

{
    Quadrado q;
    q.lado = 2;
    cout<<"Area: " << q.area() << endl;

    return 0;
}

```

Rode esse código e veja o resultado. Vai aparecer o erro:

'double Quadrado::lado' is private within this context|

Ou seja, tá dizendo que a variável 'lado' é privada! O que é isso? E agora, José?

## Especificadores de acesso: public e private

Lembra que, ao falarmos de classe e objetos, dissemos que ele tem um poder especial: de deixar algumas coisas sendo públicas para todo o código e outras coisas privadas, que podem ser acessadas somente por alguns elementos? Pois é, é isso.

Como não definimos como vai ser o acesso desses membros, o C++ colocou eles, por padrão, como *private* (ou seja, privados).

Como tá tudo privado, somente as membros internos dos objetos podem acessar esses dados. Ou seja, somente uma função do objeto pode acessar e modificar o valor da variável desse objeto.

Mas, queremos acessar essa variável na função main(), uma função de fora, nada a ver com o objeto/classe. Então, precisamos definir esses membros como públicos.

Para isto, basta digitar "public: " antes daquilo que desejamos definir como público, veja como fica nossa classe:

```

class Quadrado
{
    public:
        double lado;
        double area();
};

```

Faça essa pequena alteração no seu código, e rode ele. O resultado vai ser:

Ora, se eu defini o lado como tendo valor 2, a área vai ser  $2 \times 2 = 4$ , está correto!

Veja que maravilha essa orientação a objeto...eu criei um objeto de nome 'q', e ele automaticamente veio com a variável 'lado' e a função 'area()'.

### Segurança da informação em Classes e Objetos

Bacana...muito bonito...mas tem um problema aí...essa variável 'lado', ela é visível e acessível a qualquer parte do programa. Imagina se ela faz parte do código da Tesouraria da sua empresa, acharia interessante que alguém de outro setor tivesse acesso a essa variável?

É uma falha de segurança, concorda?

Que tal se somente uma função da classe Quadrado tivesse acesso a variável 'lado'. Faria mais sentido, não é?

Também faria sentido a variável 'lado' ser privada, ninguém poderia mexer nela fora do escopo do Objeto.

Vamos criar então uma função chamada 'define(double)' que recebe uma variável e seta esse valor pra variável 'lado'.

Nosso código, mais seguro e profissional, vai ficar assim:

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

class Quadrado
{
    public:
        void define(double);
        double area();
    private:
        double lado;
};

void Quadrado::define(double l)
{
    lado = l;
}

double Quadrado::area()
```

```

{
    return lado*lado;
}

int main()
{
    Quadrado q;
    q.define(2);
    cout<<"Area: " << q.area() << endl;

    return 0;
}

```

Ou seja, agora definimos o valor do lado do quadrado através da função `define()`, e não mais tendo acesso diretamente a variável `'lado'`. Aliás, você pode criar 1 bilhão de objetos agora, não vai ser permitido modificar nem acessar diretamente a variável `'lado'` de nenhum deles!

Quer alterar o valor do lado? Pede pra função `Quadrado.define()`

Quer saber a área daquele quadrado? Pede pra função `Quadrado.area()`

A comunicação dos objetos com o mundo externo é somente via funções, e isso é muito interessante pois podemos ter um controle maior sobre algumas informações.

## Exercício de Orientação a Objetos

1. Você precisa saber o lado de um objeto do tipo `Quadrado`, crie uma função que retorna o valor do lado desse quadrado, incrementando o código anterior.
2. Ao definir o lado do quadrado, obviamente ele não pode ser 0 ou menos, faça com que a função `define()` só aceite valores positivos de lado para o quadrado.

# Funções Get e Set, Const e Segurança da Informação com Orientação a Objetos

Neste tutorial de nosso **curso de C++**, vamos usar mais um exemplo de Classe e Objeto, dessa vez para entender melhor o uso de funções *setters* e *getters*, bem como deixar nosso código mais seguro, através da keyword *const*.

## Funções Get e Set em Orientação a Objetos

Vamos fazer outro exemplo de classe! Agora vamos trabalhar com um retângulo, que diferente do quadrado que só tem um lado de valor igual, o retângulo precisa do valor de dois lados. Vamos chamar de length (comprimento) e width (largura).

A função que pega o valor da área, vamos chamar de `getArea()` e a que retorna o valor do perímetro de `getPerimeter()`...de `get`, que aqui significa pegar, obter, acessar...

Como nossas variáveis são privadas, afinal é uma questão de segurança que ninguém de fora mexa nem veja elas, vamos precisar de funções para definir o valor dessas variáveis, serão as `setLength()` e a `setWidth()`...de `set`...falamos 'setar', que significa definir, mudar.

Veja como fica nosso código:

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

class Rect
{
    private:
        double length, width;

    public:
        double getArea();
        double getPerimeter();
        void setLength(double);
        void setWidth(double);
};

double Rect::getArea()
```



```

{
    return length * width;
}
double Rect::getPerimeter()
{
    return 2*(length+width);
}
void Rect::setLength(double l)
{
    length = l;
}
void Rect::setWidth(double w)
{
    width = w;
}

int main()
{
    Rect r;
    double var;

    cout<<"Comprimento: ";
    cin >> var;
    r.setLength(var);

    cout<<"Largura: ";
    cin >> var;
    r.setWidth(var);

    cout<<"Area: " << r.getArea() << endl;
    cout<<"Perimetro: " << r.getPerimeter() << endl;

    return 0;
}

```

Na main(), instanciamos um objeto 'r' do tipo Rect, e declaramos uma variável 'var' que vai receber os valores que o usuário vai digitar, pra os valores do comprimento e largura.

Primeiro ele digita o valor para o comprimento, e setamos esse valor através da setLength().

Depois ele digita o valor para a largura, e setamos esse valor através da setWidth().

Depois é só exibir o valor da área e do perímetro, através das `getArea()` e `getPerimeter()`.

Ou seja, usamos getters e setters para acessar e mudar as variáveis. Veja novamente: ninguém no mundo externo, nem o melhor hacker do planeta, vai ter acesso direto a essas variáveis, somente através das funções de set e get.

## Segurança da Informação com Orientação a Objetos

"Ué, a gente não tem acesso as variáveis diretamente, mas temos através das funções de Set e Get, que é a mesma coisa".

Bom, quase, mas tem um fundo de razão. Você ainda precisaria adivinhar o nome dessas funções, pra ter acesso a elas, mas não discordo de você, essa segurança toda ainda não ficou à prova.

Um exemplo: um hacker danadinho poderia muito bem definir valores negativos para os lados do retângulo, o que é um absurdo! Os lados de um retângulo devem ser positivos!

É aí que entra a mágica da orientação a objetos! Vamos alterar nossas funções de `setLength()` e `setWidth()`, vão ficar assim:

```
void Rect::setLength(double l)
{
    while(l<=0){
        cout << "Comprimento negativo! Digite um valor válido: ";
        cin >> l;
    }
    length = l;
}
void Rect::setWidth(double w)
{
    while(w<=0){
        cout << "Largura negativa! Digite um valor válido: ";
        cin >> w;
    }
    width = w;
}
```

Amigos, enquanto valor de l ou w for negativo, vai cair dentro do while e só sai desse while quando o valor digitado for maior que 0, e não tem papo, pode ser o melhor hacker do mundo, ele não vai driblar isso.

## A palavra-chave **const** em Orientação a Objetos

Como dissemos, o objetivo das funções *getters* é simplesmente pegar valores, acessar essas informações, jamais, nunca, alterá-la.

Você pode trabalhar na tesouraria de uma empresa e disponibilizar a função `getSalario()` pro restante da empresa, para cada um visualizar o valor do salário...mas em hipótese alguma essas pessoas devem poder alterar o valor apresentado (no caso, o salário...imagina que maravilha seria poder alterar seu salário?)

Uma maneira de garantir que essas funções são apenas de acesso, é usar a keyword **const**, tanto no cabeçalho dentro da definição de classe como na declaração da função.

Veja agora como fica nosso código todo completo, seguro, lindo e maravilhoso:

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

class Rect
{
    private:
        double length, width;

    public:
        double getArea() const;
        double getPerimeter() const;
        void setLength(double);
        void setWidth(double);
};

double Rect::getArea() const
{
    return length * width;
}
double Rect::getPerimeter() const
```

```

{
    return 2*(length+width);
}
void Rect::setLength(double l)
{
    while(l<=0){
        cout << "Comprimento negativo! Digite um valor válido: ";
        cin >> l;
    }
    length = l;
}
void Rect::setWidth(double w)
{
    while(w<=0){
        cout << "Largura negativa! Digite um valor válido: ";
        cin >> w;
    }
    width = w;
}

int main()
{
    Rect r;
    double var;

    cout<<"Comprimento: ";
    cin >> var;
    r.setLength(var);

    cout<<"Largura: ";
    cin >> var;
    r.setWidth(var);

    cout<<"Area: " << r.getArea() << endl;
    cout<<"Perimetro: " << r.getPerimeter() << endl;

    return 0;
}

```

## Exercício de Orientação a Objetos

Dê uma incrementada no exemplo anterior. Faça que ele exiba também os lados do retângulo, para isso defina e use as funções `getLength()` e `getWidth()`, use a palavra-chave `const` para deixar essas funções mais seguras e à prova de hackers.

# Alocação dinâmica de objetos com ponteiros: O operador new e delete

Neste tutorial de nosso **Curso de Orientação a Objetos em C++**, vamos aprender a usar ponteiros com objetos, e ver uma nova maneira de alocar memória.

## Ponteiros de Objetos

Inicialmente, vamos criar um classe, chamada Student, que vai ter duas variáveis (math e english), que vai armazenar as notas do aluno. Também vai ter a get and set, e uma chamada getAverage() para calcular a média de cada aluno.

Nossa classe, toda completinha e bonitinha fica assim:

```
class Student
{
    private:
        double math, english;

    public:
        double getAverage() const;
        void setMath(double);
        void setEnglish(double);
        double getMath() const;
        double getEnglish() const;
};

double Student::getAverage() const
{
    return (getMath() + getEnglish())/2;
}

void Student::setMath(double m)
{
    math = m;
}

void Student::setEnglish(double e)
{
    english = e;
}
```

```

}
double Student::getMath() const
{
    return math;
}
double Student::getEnglish() const
{
    return english;
}

```

Vamos agora declarar um ponteiro de nome 'ptr' do tipo Student:

- Student \*ptrStudent

Agora vamos declarar de fato um objeto da classe Student:

- Student s1;

Podemos associar o ponteiro com o objeto da seguinte maneira:

- ptrStudent = &s1  
(não esqueça o &, lembre-se que ponteiros armazenam endereços de memória, e ao colocarmos o & antes de uma variável, ele fornece o endereço daquela variável)

## Trabalhando com ponteiros com Classes e Objetos: ->

Como já estudando ponteiros, vimos que eles podem fazer mil maravilhas. Pode exemplo, no exemplo anterior, para o ponteiro acessar os elementos de um objeto, devemos usar o operador ->

Vamos definir duas notas para um aluno:

```

ptrStudent->setMath(8);
ptrStudent->setEnglish(10);

```

E para sabermos a média, usando ponteiro, basta fazer:

```

ptrStudent->getAverage()

```

Nosso código completo fica:

```

#include <iostream>
using namespace std;

class Student
{
    private:
        double math, english;

    public:
        double getAverage() const;
        void setMath(double);
        void setEnglish(double);
        double getMath() const;
        double getEnglish() const;
};

double Student::getAverage() const
{
    return (getMath() + getEnglish())/2;
}

void Student::setMath(double m)
{
    math = m;
}

void Student::setEnglish(double e)
{
    english = e;
}

double Student::getMath() const
{
    return math;
}

double Student::getEnglish() const
{
    return english;
}

int main()
{
    Student *ptrStudent;
    Student s1;

```



```

ptrStudent = &s1;

ptrStudent->setMath(8);
ptrStudent->setEnglish(10);

cout << ptrStudent->getAverage()<<endl;

return 0;
}

```

Note que usamos apenas um aluno, poderíamos ter criado mil alunos e ir fazendo o ponteiro apontar para cada um desses alunos, e o código seria sempre o mesmo, por exemplo:

Student s1, s2, s3, s4;

Faz: ptrStudent = &s1

Depois: ptrStudent = &s2

Depois: ptrStudent = &s3

Depois ptrStudent = &s4

E prontinho, depois é só usar

ptrStudent->setMath();

ptrStudent->setEnglish();

ptrStudent->getAverage()

Isso evita muito escrever código à toa.

## Alocando espaço para objetos: Operador **new**

Existe uma nova maneira de instanciar objetos, é a partir do operador **new**. Primeiro você declara seu ponteiro:

- MyClasse \*ptr;

Nesse ponto, temos um ponteiro que serve para apontar para um objeto? Que objeto é esse? Vamos criar ele agora, da seguinte maneira:

- ptr = new MyClasse

## Deletando objetos - **delete**

Quando criamos um objeto, via instanciamento normal, ele vai sempre existir e ocupar um espaço na memória. Uma das vantagens de fazer a alocação dinâmica de memória, através do **new**, é que podemos destruir esse ponteiro, destruindo esse objeto e liberando espaço na memória, para isto, basta usar o operador **delete**:

- `delete ptr;`

E prontinho, temos um espaço livre na máquina. Isso pode parecer bobeira agora, mas quando você criar sistemas super complexos que exigem o máximo de eficiência e o menor uso possível de memória (como para programar um relógio de pulso ou microondas, que tem pouca memória), é uma técnica importantíssima.

Veja o código abaixo, apenas com um ponteiro, preenchamos a nota de dois alunos e calculamos sua média, e tudo isso apenas como um objeto/ponteiro da classe Student. Veja como ficou o código:

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

class Student
{
    private:
        double math, english;

    public:
        double getAverage() const;
        void setMath(double);
        void setEnglish(double);
        double getMath() const;
        double getEnglish() const;
};

double Student::getAverage() const
{
    return (getMath() + getEnglish())/2;
}
```

```

void Student::setMath(double m)
{
    math = m;
}
void Student::setEnglish(double e)
{
    english = e;
}
double Student::getMath() const
{
    return math;
}
double Student::getEnglish() const
{
    return english;
}

int main()
{
    Student *ptrStudent;
    ptrStudent = new Student;

    ptrStudent->setMath(8);
    ptrStudent->setEnglish(10);
    cout << "Media do Estudante 1: " << ptrStudent->getAverage() << endl;

    ptrStudent->setMath(6);
    ptrStudent->setEnglish(7);
    cout << "Media do Estudante 2: " << ptrStudent->getAverage() << endl;

    delete ptrStudent;

    return 0;
}

```

# Como criar um projeto com Classes e Objetos em C++: Cabeçalhos e Implementações

Neste tutorial de nossa **apostila de C++**, vamos aprender a organizar nossas classes e implementações de funções, criando um projeto bem profissional.

## Como criar um projeto em C++

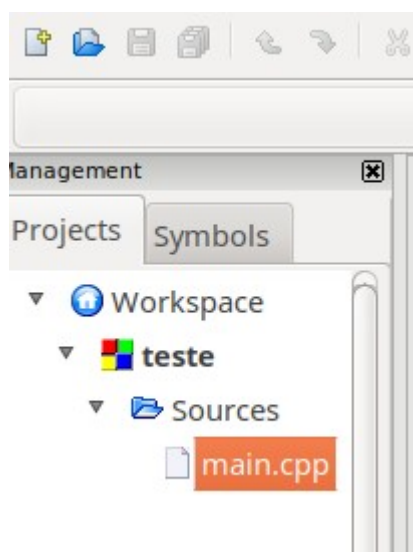
Abra seu Code::Blocks (ou outra IDE que esteja usando).

Clique em File, depois New e escolha Project.

Escolha Console application e marque a opção C++.

Dê um nome para seu projeto, escolha a pasta que quer salvar e clique em Finish,

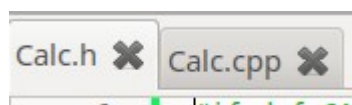
Vai ficar assim:



Note que já veio o arquivo 'main.cpp', vamos já usar ele.

Agora, clique naquele arquivo branco lá em cima, de 'New file', e clique em Class. Dê o nome 'Calc' para essa classe, pois vamos criar uma calculadora.

Agora veja que legal:



Ele criou dois arquivos: Calc.h e o Calc.cpp

## Implementando uma Classe em um Projeto C++:

**.h**

Ok, vamos lá, primeiro vamos trabalhar no arquivo Calc.h, ele é chamado de cabeçalho.

Vamos ao esboço de nossa classe.

Ela vai ter dois números: num1 e num2, obviamente private.

Vai ter funções de soma (sum), subtração (sub), multiplicação (prod) e divisão (div), todas public e const (elas não vão poder alterar as variáveis).

Somente as funções setNum1 e setNum2 vão poder alterar as variáveis.

E prontinho, nosso cabeçalho da classe Calc.h fica assim:

```
#ifndef CALC_H
#define CALC_H

class Calc
{
    private:
        double num1, num2;

    public
        double sum() const;
        double sub() const;
        double prod() const;
        double div() const;
        void setNum1(double);
        void setNum2(double);
};

#endif // CALC_H
```

Veja que tem uns #ifndef, #define, #endif...não mexa neles, são super importantes, deixem eles aí que depois explicamos seus significados.

## Implementando Funções de uma Classe de um Projeto: **.cpp**

É nesse arquivo que vamos implementar as funções de nossa classe Calc.h

Note que bem no começo, tem um "#include Calc.h", assim esse arquivo .cpp fica diretamente relacionado ao arquivo .h , entende a mágica da coisa?

Neste arquivo, vamos implementar todas as funções getters e setters que foram previamente apresentadas no cabeçalho da classe. O arquivo .cpp fica assim:

Note que não tem nenhum 'cout', se tivesse, teríamos que incluir 'include <iostream>':

```
#include "Calc.h"
```

```
double Calc::getSum() const
{
    return num1+num2;
}
```

```
double Calc::getSub() const
{
    return num1-num2;
}
```

```
double Calc::getProd() const
{
    return num1*num2;
}
```

```
double Calc::getDiv() const
{
    return num1/num2;
}
```

```
void Calc::setNum1(double n1)
{
    num1=n1;
}
```

```
void Calc::setNum2(double n2)
{
    num2=n2;
}
```

```
}
```

## Rodando nosso projeto C++: **main.cpp**

Agora, na main.cpp, a única coisa que você tem que fazer é dar um #include "Calc.h" lá em cima no arquivo, e prontinho, veja como ficou a main:

```
#include <iostream>
#include "Calc.h"
```

```
using namespace std;
```

```
int main()
{
    Calc c;
    double num;

    cout << "Primeiro numero: ";
    cin >> num;
    c.setNum1(num);

    cout << "Segundo numero: ";
    cin >> num;
    c.setNum2(num);

    cout << "\nResultados: "<<endl;
    cout << "Soma: " << c.getSum()<<endl;
    cout << "Subtração: " << c.getSub()<<endl;
    cout << "Multiplicação: " << c.getProd()<<endl;
    cout << "Divisão: " << c.getDiv()<<endl;

}
```

Agora imagine por um instante se tivéssemos declarado essa classe e as implementações das funções tudo na main.cpp, a bagunça que seria e o código gigantesco.

Dessa maneira que fizemos, separando por cabeçalhos (.h) e implementações de funções (.cpp), ficou tudo mais bonitinho, organizado e profissional, é assim que fazem e usam em softwares profissionais.

Habitue-se a fazer seus projetos com essa organização, ok?

# Construtores em C++- Orientação a Objetos

Neste tutorial, vamos aprender sobre um tipo especial de função, a função Construtora, que desempenha um papel muito importante na orientação a objetos.

## A função Construtora do C++

Existe uma função muitíssimo importante e utilizada em orientação a objetos, é a função construtora.

E ela possui algumas características únicas e especiais.

A primeira delas é que seu nome é o mesmo nome da classe. Ou seja, se sua classe se chama "MyClass", a função construtora tem que se chamar "MyClass()", obrigatoriamente, ok?

Outra característica dela é que, assim que instanciamos um objeto, a função construtora é sempre automaticamente invocada!

Por fim, uma última característica, ela não retorna nenhum valor! Não escreva void, int, double etc, nada em seu cabeçalho e implementação.

Vamos ver um teste de uma função construtora funcionando? Vamos criar a classe "MyClass" que tem apenas um membro, a função "MyClass()", que simplesmente vai exibir uma mensagem, ela fica assim:

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

class MyClass
{
    public:
        MyClass();
};

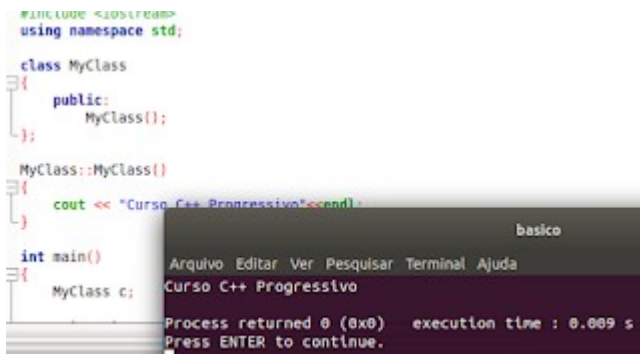
MyClass::MyClass()
{
    cout << "Curso C++ Progressivo"<<endl;
}

int main()
{
    MyClass c;
```



```
    return 0;
}
```

Prontinho! Apenas criamos o objeto 'c' e veja o que apareceu na tela:

The image shows a C++ IDE with a code editor on the left and a terminal on the right. The code in the editor defines a class 'MyClass' with a public constructor 'MyClass()' and a member function 'MyClass::MyClass()' that prints 'Curso C++ Progressivo' to the console. The 'main' function creates an object 'c' of type 'MyClass'. The terminal on the right shows the output 'Curso C++ Progressivo' and the message 'Process returned 0 (0x0) execution time : 0.009 s Press ENTER to continue.'

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

class MyClass
{
public:
    MyClass();
};

MyClass::MyClass()
{
    cout << "Curso C++ Progressivo" << endl;
}

int main()
{
    MyClass c;
}
```

## Função Construtora: Para que serve?

Ok, já vimos como fazer, como funciona e o resultado...mas pra que serve isso? Só pra mostrar umas mensagens bobinhas na tela?

Claro que não, jovem!

Como o próprio nome diz, ele serve para 'construir' coisas, no caso, ele constrói coisas automaticamente. Quando criamos um objeto, é interessante que ele já faça algumas operações internas de imediato, sem precisarmos sempre fazer tudo manualmente.

A este tipo de construtor, chamamos ele de *default*, pois ele não recebe nenhum argumento. Porém, a maneira mais comum e útil de usar uma função construtora é com uma lista de parâmetros, que são informações que vem de fora pra dentro do objeto, e vão desempenhar algum papel importante lá dentro.

## Função Construtora: Como usar?

Por exemplo, vamos relembrar nossa classe `Rect`, que criava retângulos. Lembra que sempre tínhamos que usar as funções setters para definir os valores de comprimento e largura. Que tal se o construtor fizesse isso automaticamente?

Veja como fica:

```

#include <iostream>
using namespace std;

class Rect
{
    private:
        double length, width;

    public:
        Rect(double, double);
        double getArea();
        double getPerimeter();
};

Rect::Rect(double l, double w)
{
    length = l;
    width = w;
}

double Rect::getArea()
{
    return length * width;
}

double Rect::getPerimeter()
{
    return 2*(length+width);
}

int main()
{
    double len, wid;

    cout<<"Comprimento: ";
    cin >> len;

    cout<<"Largura: ";
    cin >> wid;

    Rect r(len, wid);

    cout<<"Area: " << r.getArea() << endl;
    cout<<"Perimetro: " << r.getPerimeter() << endl;
}

```

```
    return 0;  
}
```

Note que no cabeçalho do construtor, já especificamos que ele vai receber duas variáveis do tipo double.

Na implementação, chamamos essas variáveis de 'l' e 'w', que serão fornecidas mais adiante pelo usuário.. E o que nosso construtor faz? Vai setar os valores corretos de 'length' e 'width'.

Na main, pedimos os valores de comprimento e largura, e depois simplesmente instanciamos um objeto do tipo Rect, de nome 'r'. Note agora que nele enviamos dois argumentos, que são os dois argumentos que a função construtora vai usar!

Pronto, internamente, dentro do objeto, o construtor já fez tudo e as funções getArea() e getPerimeter() já podem ser usadas.

## Um pouco mais sobre Função Construtora

Uma outra curiosidade sobre construtores é que eles SEMPRE existem! Mesmo que você não defina nenhum, o C++ vai lá e cria um, vazio, que não faz nada.

Quando você faz:

- MyClass c;

Ele automaticamente vai invocar o construtor, mas você não vê nada ocorrendo, pois não definiu nenhum construtor.

Outro truque que você pode precisar e usar, é através de ponteiros, declarando um ponteiro de um objeto:

- MyClass \*ptrClass;

Ao fazer isso, o método construtor não será executado, pois não foi criado um objeto, apenas um ponteiro para um objeto.

Agora se fizer isso em seguida:

- ptrClass = new MyClass;

Aí sim, o objeto foi instanciado e alocado na memória, e o construtor será sumariamente executado.

# Função Destruidor (destructor) em C++

Neste tutorial de nosso **curso de C++**, vamos conhecer e aprender a usar as funções destruidoras.

## A função Destruidor (Destructor)

No tutorial passado, falamos sobre a [Função Construtor](#), que existe em toda classe e é executada sempre que um objeto é instanciado.

Analogamente, existe a função destruidor, que é executada apenas quando o objeto é destruído.

Para criarmos uma função destructor, basta definirmos uma função com o mesmo nome da classe, com o símbolo de til ( ~ ) antes. Vamos ver um exemplo, criando uma classe com construtor e destruidor:

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

class Test
{
    public:
        Test();
        ~Test();
};

Test::Test()
{
    cout << "Constructor" << endl;
}

Test::~Test()
{
    cout << "Destructor" << endl;
}

int main()
{
    Test t;

    return 0;
}
```

O resultado vai ser as duas palavras na tela. Uma ocorre ao criarmos o objeto e outra ocorre quando o programa se encerra, e o nesse caso o destructor também é invocado.

E assim também como os construtores, toda classe tem destruidor. Mesmo que você não defina um, vai existir um padrão em branco, mas vai existir sempre.

Também não aceitam nenhum argumento, nem lista de parâmetros e não retornam nenhuma informação.

## Para que serve a Função Destruidor ?

Basicamente, para fazer coisas que devem ser feitas quando um objeto deixa de existir.

Um exemplo muito comum é liberar memória que foi alocada dentro de um objeto.

No momento, estamos usando objetos bobinhos, pequenos, mas o normal é serem gigantes, com classes com centenas de variáveis e dezenas de funções. E lá dentro, é comum fazer alocação dinâmica de memória, e quando esse objeto deixar de existir, é uma boa prática liberar toda essa memória (principalmente em sistemas mais críticos com pouca memória, como seu relógio ou o sistema digital da sua geladeira).

Vamos supor que você criou um sistema pro banco, e quer sempre fazer um teste: se alguém acessou a classe Bank, instanciando algum objeto. Para isso, você define uma variável de escopo global chamada "spy" com valor inicial 0.

Pra verificar se algum objeto foi criado, faça com que 'spy=1' no destructor, veja:

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

```
int spy=0;
```

```
class Bank
{
    public:
        ~Bank();
}
```

```
};
```

```
Bank::~~Bank()
```

```
{  
    spy = 1;  
}
```

```
int main()
```

```
{  
    Bank *b;  
    b = new Bank;  
  
    delete b;  
  
    cout<<"Spy: " << spy << endl;  
    return 0;  
}
```

Vamos supor que seja uma classe gigante, com vários membros, várias coisas acontecendo...você criou um objeto, usou ele, fez tudo direitinho e tal. Então chega a hora de dar um delete nele, e nessa hora a função destructor vai ser executada.

O resultado vai ser 'Spy: 1' ali na main(), indicando que alguém mexeu nessa classe Bank. Ou seja, o destruidor, por sempre ser executado, vai executar a operação 'spy=1'.

Um outro exemplo função destructor, é se você criar um jogo e tiver um personagem (que é um objeto) e ele simplesmente morrer ou sair do jogo. Na sua função destruidor, você deleta o nome, tirar seus pontos, tira ele dos times, tira do ranking.

E etc etc, por ai vai pessoal, quase sempre tem coisas pra se fazer quando um objeto deixa de existir, ok?