

# Curso de Extensão - INF1900

# Programação em C++



## Módulo 1

# Introdução à Programação em C++

## Aulas 1&2

Profa. Dra. Esther Luna Colombini

[esther@ic.unicamp.br](mailto:esther@ic.unicamp.br)

Agosto de 2023

# Módulo 1: Introdução à Programação em C++ (10h)



Profa. Dra. Esther Luna Colombini



Familiarizar os alunos com os conceitos básicos da linguagem C++ e prepará-los para escrever programas simples, aprofundando o conhecimento em estruturas de dados e gerenciamento da memória em programas C++.

- Introdução à linguagem C++ e sua história
- Ambiente de desenvolvimento, configuração do compilador, pré-processador e link estático e dinâmico
- Estrutura básica de um programa em C++
- Tipos de dados, variáveis e constantes
- Conversões de tipos e value categories
- Operadores aritméticos, lógicos e relacionais e operadores bit a bit
- Controle de fluxo: estruturas condicionais e laços de repetição
- Funções e procedimentos em C++
- Funções lambda
- Manipulação de entradas e saídas
- Arrays e matrizes
- Strings e manipulação de cadeias de caracteres
- Ponteiros e referências
- Alocação dinâmica de memória
- Gerenciamento de memória e desalocação
- Estruturas de dados avançadas: listas, pilhas e filas

# Monitorias

- **Monitores do Módulo:**
  - Alana Correia
  - Iury Cleveston
- **Dia/Horário do atendimento dos monitores:**
  - A definir em conjunto
- **Dia/Horário do atendimento do professor:**
  - Quintas às 18:00h

# Calendário

**Aulas:** segunda-feira e quarta-feira

**Horário:** 8:00h às 10:00h



14/08/23	MÓDULO 1
16/08/23	MÓDULO 1
18/08/23	MÓDULO 1 - ATENDIMENTO
21/08/23	MÓDULO 1
23/08/23	MÓDULO 1
24/08/23	MÓDULO 1 - ATENDIMENTO

# Avaliação

- **Avaliação:**
  - Atividades práticas a serem realizadas em dupla liberadas todas às quartas
- **Média final:** média aritmética das notas

# Bibliografia

- Stroustrup, B. (2013). The C++ Programming Language (4th Edition). Addison-Wesley Professional.
- Josuttis, N. (2019). C++17 - The Complete Guide: First Edition. Leanpub.
- Schildt, H. (2017). C++: The Complete Reference (5th Edition). McGraw-Hill Education.

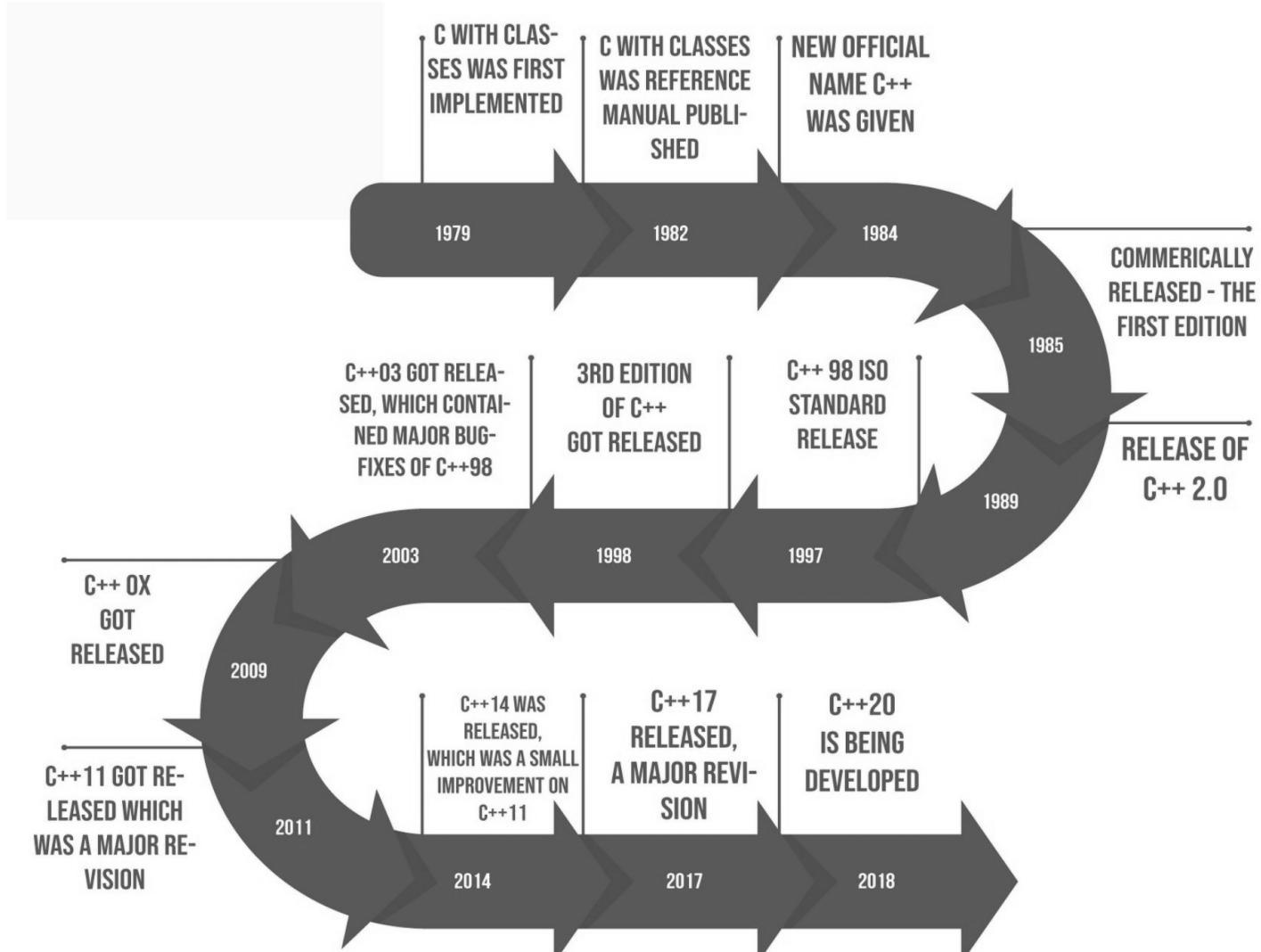
# C++

# Um pouco sobre C++



- Criada, projetada e desenvolvida por **Bjarne Stroustrup** no **Bell Telephone Laboratories**
- C++ é uma linguagem multiparadigma com suporte a programação orientada a objetos
  - Inspirada na linguagem de programação Simula67
  - Mas é possível escrever código em C++ sem usar classes
- É considerada uma linguagem de nível médio
  - uma combinação de linguagens de baixo e alto nível
- Inicialmente, semelhante a C, mas com adicionalidade de verificação de tipo ativo, herança básica, classes, inlining, entre outros
- Chamada de C com classes
- Renomeada para C++ em 1983

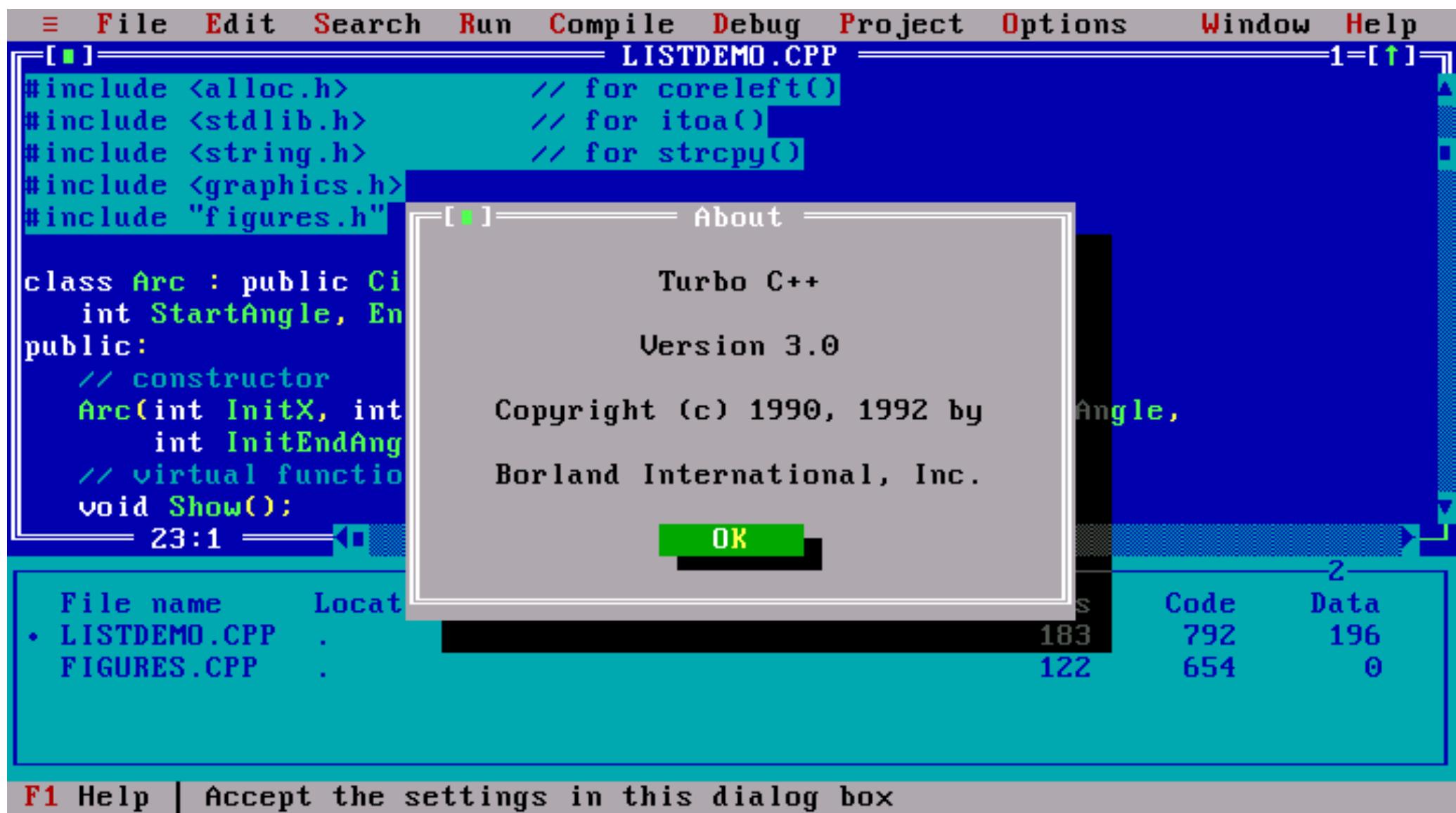
# Um pouco da história



Fonte: <https://www.geeksforgeeks.org/history-of-c/>

# Borland C++ Compiler

Última versão estável de 2006



# Popularidade

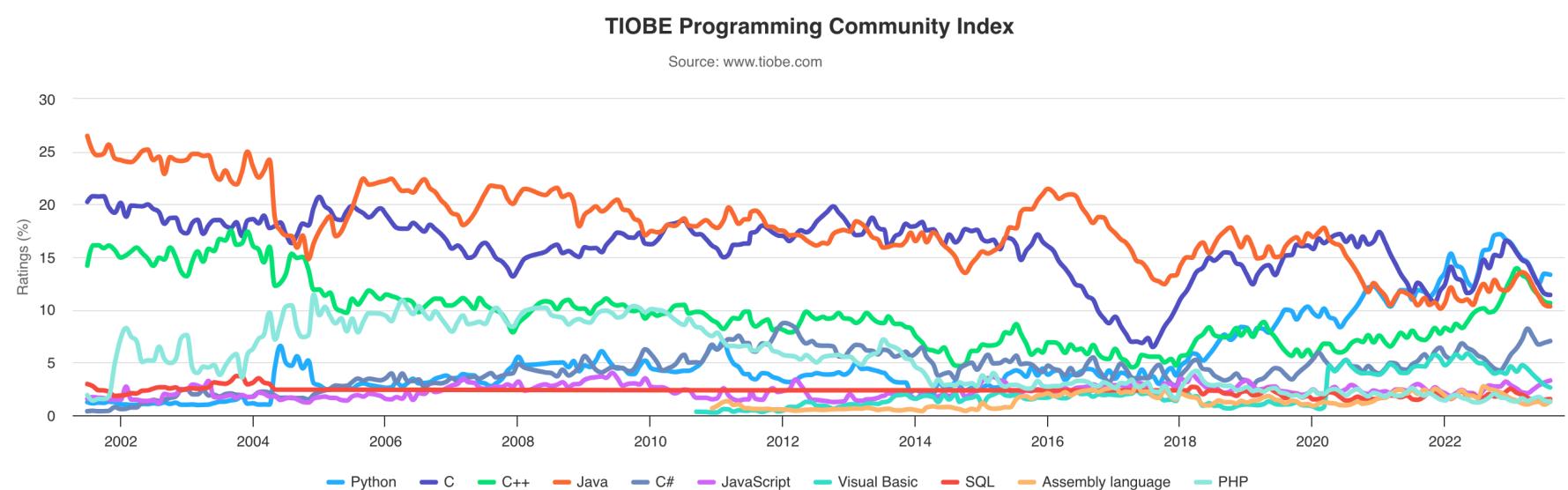
RedMonk	Stack Overflow	SlashData	TIOBE Index 7/19
JavaScript	JavaScript	JavaScript	Java
Java	HTML/CSS	Python	C
Python	SQL	Java	Python
PHP	Python	C#	C++
Tie: C++/C#	Java	C/C++	C#
#rowspan#	Bash/Shell/Powershell	PHP	Visual Basic.NET
CSS	C#	Visual tools	JavaScript
Ruby	PHP	Swift	PHP

# Popularidade

Aug 2023	Aug 2022	Change	Programming Language	Ratings	Change
1	1		 Python	13.33%	-2.30%
2	2		 C	11.41%	-3.35%
3	4		 C++	10.63%	+0.49%
4	3		 Java	10.33%	-2.14%
5	5		 C#	7.04%	+1.64%
6	8		 JavaScript	3.29%	+0.89%
7	6		 Visual Basic	2.63%	-2.26%
8	9		 SQL	1.53%	-0.14%
9	7		 Assembly language	1.34%	-1.41%
10	10		 PHP	1.27%	-0.09%

Fonte: [https://www.tiobe.com/tiobe-index/programminglanguages\\_definition/](https://www.tiobe.com/tiobe-index/programminglanguages_definition/)

# Popularidade

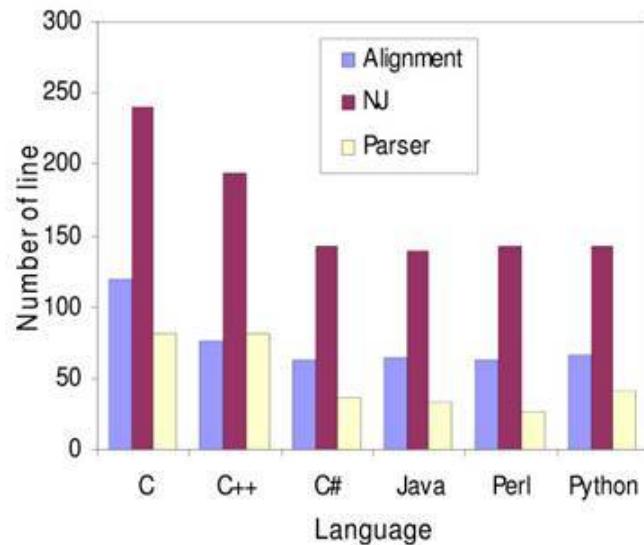
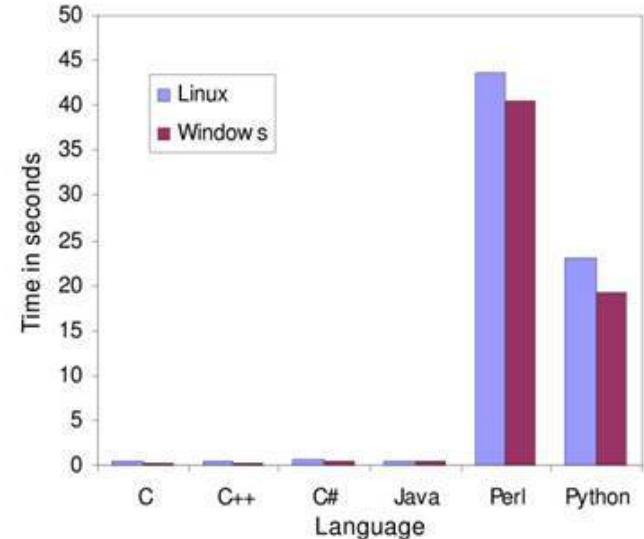
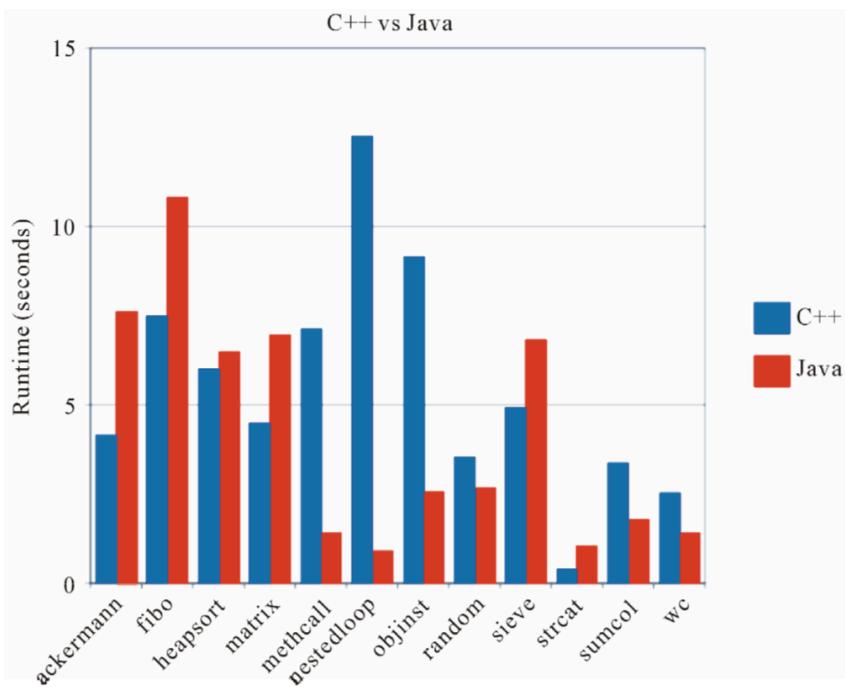


Fonte: <https://www.tiobe.com/>

# 10 razões porque C++ é popular (ainda)

1. **Desempenho:** é conhecida por seu desempenho de alto nível. Permite programação de baixo nível, permitindo aos desenvolvedores um controle próximo do hardware. Isso é especialmente importante para sistemas que precisam de alta velocidade e eficiência, como sistemas embarcados, jogos, simulações científicas e software de tempo real.
2. **Versatilidade:** pode ser usada para desenvolver uma ampla variedade de tipos de aplicações, desde sistemas operacionais até aplicações de desktop, jogos, aplicativos móveis e muito mais.
3. **Herança do C:** C++ é uma extensão da linguagem C. Muitos programas legados e sistemas são escritos em C, e a transição para C++ pode ser mais fácil para esses sistemas, pois C++ mantém a compatibilidade com a sintaxe e recursos do C.
4. **Recursos avançados:** C++ oferece recursos avançados como classes e objetos (programação orientada a objetos), templates (programação genérica), manipulação direta de memória e outras características que permitem uma abordagem mais sofisticada e eficiente para o desenvolvimento de software.
5. **Comunidade e Bibliotecas:** Existem bibliotecas poderosas em C++ que facilitam o desenvolvimento, como a Standard Template Library (STL), que fornece estruturas de dados e algoritmos prontos para uso. A comunidade C++ é ativa e existem muitos recursos, fóruns e materiais de aprendizado disponíveis.
6. **Indústrias específicas:** C++ é amplamente utilizada em indústrias como jogos, sistemas embarcados, engenharia de software, finanças e simulações científicas, onde seu desempenho e controle são críticos.
7. **Compatibilidade e Portabilidade:** Embora C++ permita programação de baixo nível, também é possível desenvolver em um nível mais alto de abstração, tornando-o adequado para diferentes plataformas e sistemas operacionais.
8. **Legado e Investimentos:** Muitos sistemas existentes e projetos grandes são escritos em C++. Mudar para outra linguagem exigiria um esforço significativo e, em muitos casos, pode não ser viável devido ao investimento já feito.
9. **Evolução contínua:** A linguagem C++ está em constante evolução. Novos recursos e melhorias são adicionados em padrões revisados da linguagem (como o C++11, C++14, C++17, etc.), o que mantém a linguagem relevante e atualizada.
10. **Ponteiros** ☺

# C++: Desempenho



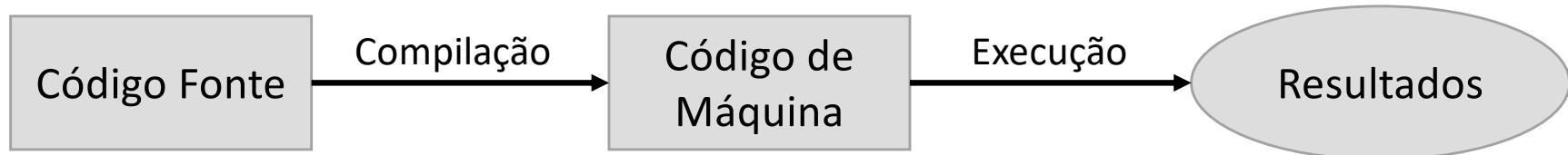
Fonte: <http://dx.doi.org/10.4236/jsea.2012.58072>

# C++: Características

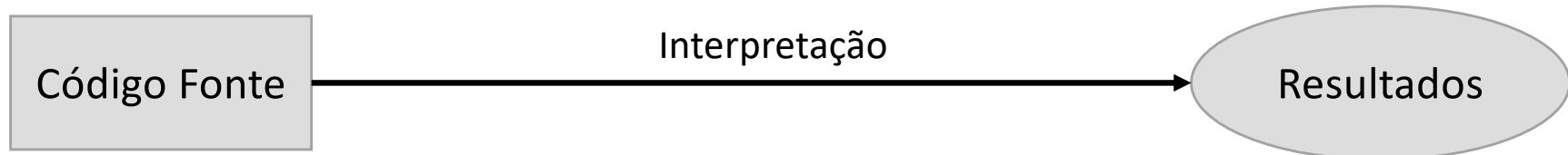
- Programação Orientada a Objetos
- Independente da Máquina
- Simples
- Linguagem de alto nível
- Popular
- Case-sensitive
- **Compilada**
- Alocação Dinâmica de Memória
- Gerenciamento de memória
- Multi-threading
- Tratamento de exceções

# C++: Características

- Compilação



- Interpretação



# C++: Compiladores

- **GNU Compiler Collection (GCC):** Amplamente utilizado e maduro, oferece suporte a uma ampla gama de plataformas, geração de código bem otimizada, suporte abrangente a idiomas, extensos sinalizadores de compilador, forte suporte da comunidade.
- **Clang:** Tempos de compilação rápidos, fornecem mensagens de erro muito detalhadas e legíveis por humanos, projetadas com a modularidade em mente, parte do projeto LLVM que promove tecnologia de compilador inovadora, conformidade com bons padrões.
- **Microsoft Visual C++ (MSVC):** Integrado ao IDE do Microsoft Visual Studio, boa integração com o desenvolvimento do Windows, oferece suporte a recursos específicos da Microsoft, bom desempenho em plataformas Windows.
- **Compilador Intel C++:** Bem conhecido por produzir código altamente otimizado, forte suporte para paralelismo e vetorização, frequentemente usado para computação científica e de alto desempenho.
- **MinGW-W64:** Fornece uma porta do GCC para desenvolvimento do Windows, permitindo o desenvolvimento multiplataforma com um compilador familiar, frequentemente atualizado e aprimorado.
- **Embarcadero C++:** Integrado ao Embarcadero RAD Studio, projetado para desenvolvimento rápido de aplicativos (especialmente para Windows), biblioteca de componentes visuais para construção de aplicativos GUI.

# C++: Build System

- **Makefile** é uma ferramenta direta e simples para compilação em sistemas Unix-like
- O **CMake** é uma ferramenta mais avançada e abstrata de geração de build, projetada para simplificar o processo de compilação, especialmente em projetos que precisam ser compilados em diferentes plataformas e ambientes. Ele gera Makefiles (ou outros tipos de arquivos de construção, como soluções Visual Studio no Windows) automaticamente com base em um arquivo de configuração chamado CMakeLists.txt.
- O CMake é muitas vezes preferido em cenários onde a portabilidade é uma prioridade e onde a manutenção de arquivos de construção complexos pode ser problemática.

# C++: Build System

```
# Nome do programa a ser gerado
TARGET = meu_programa

# Compilador C++
CXX = g++

# Flags de compilação
CXXFLAGS = -Wall -std=c++11

# Arquivo-fonte
SOURCE = main.cpp

# Comando para gerar o programa
$(TARGET) : $(SOURCE)
    $(CXX) $(CXXFLAGS) -o $(TARGET) $(SOURCE)

# Comando para limpar os arquivos gerados
clean:
    rm -f $(TARGET)
```

**TARGET:** Nome do programa que será gerado.

**CXX:** Compilador C++ a ser utilizado.

**CXXFLAGS:** Flags de compilação, como -Wall para ativar avisos e -std=c++11 para usar o padrão C++11.

**SOURCE:** Nome do arquivo-fonte.

# C++: Build System

CMakeLists.txt (no diretório raiz do projeto):

```
cmake_minimum_required(VERSION 3.10)

project(MeuProjeto)

# Define a versão do C++
set(CMAKE_CXX_STANDARD 11)

# Adiciona o diretório "src" ao processo de compilação
add_subdirectory(src)

# Adiciona o diretório "lib" ao processo de compilação
add_subdirectory(lib)
```

```
meu_projeto/
└── CMakeLists.txt
└── src/
    ├── main.cpp
    ├── calculadora.cpp
    └── calculadora.h
└── lib/
    ├── CMakeLists.txt
    ├── funcoes.cpp
    └── funcoes.h
```

# C++: Build System

CMakeLists.txt (no diretório src do projeto):

```
# Lista de arquivos-fonte
set(SOURCES
    main.cpp
    calculadora.cpp
)
# Cria um executável a partir dos arquivos-fonte
add_executable(meuprojeto ${SOURCES})
# Link com a biblioteca "minhabiblioteca"
target_link_libraries(meuprojeto PRIVATE minhabiblioteca)
```

```
meu_projeto/
└── CMakeLists.txt
└── src/
    ├── main.cpp
    ├── calculadora.cpp
    └── calculadora.h
└── lib/
    ├── CMakeLists.txt
    ├── funcoes.cpp
    └── funcoes.h
```

# C++: Build System

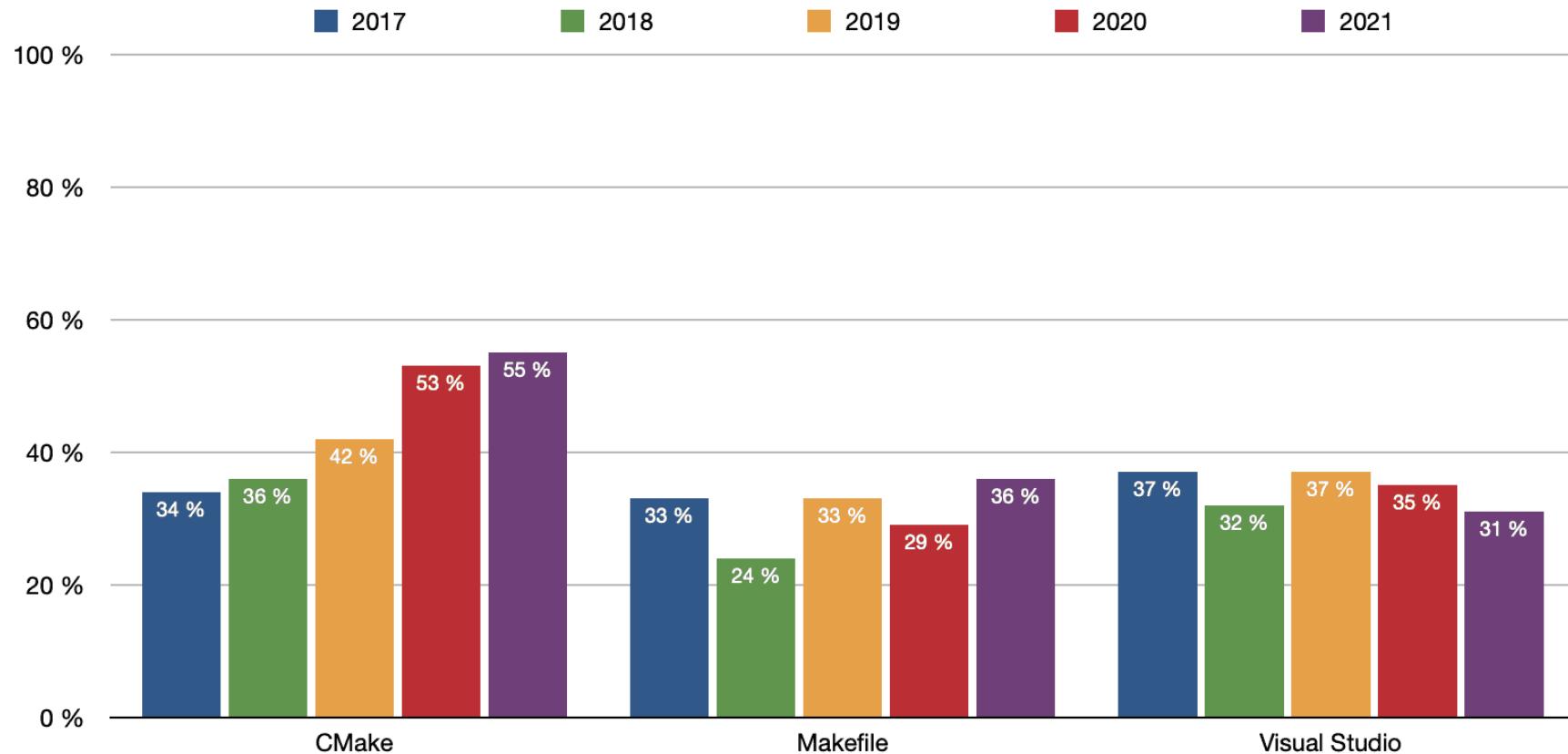
CMakeLists.txt (no diretório lib do projeto):

```
# Lista de arquivos-fonte
set(LIB_SOURCES
    funcoes.cpp
)

# Cria uma biblioteca a partir dos arquivos-fonte
add_library(minhabiblioteca ${LIB_SOURCES})
```

```
meu_projeto/
└── CMakeLists.txt
└── src/
    ├── main.cpp
    ├── calculadora.cpp
    └── calculadora.h
└── lib/
    ├── CMakeLists.txt
    ├── funcoes.cpp
    └── funcoes.h
```

# C++: Build System



# C++: Como seu Código vira um programa

- O **compilador** é apenas parte do processo de construção do executável
  - Um compilador gera arquivos de código-objeto (linguagem de máquina) a partir do código-fonte e faz otimizações
  - Quando compilamos um programa C++, cada **translation unit** é compilada separadamente em um arquivo objeto.
  - Uma translation unit é o conjunto de todos os arquivos necessários para compilar um único arquivo de código-fonte. Ela é a unidade lógica de compilação
- Para que esse código seja útil ele precisa estar
  - organizado
  - no formato que o SO comprehenda
  - disponível para ser usado
- Um **linker** combina esses arquivos de código de objeto em um executável
  - Resolve chamadas de funções
  - Formata o código para o SO
  - Faz relocação

# C++: Linker

- Junta arquivos objeto em um arquivo executável que o SO possa usar
- O compilador gera um **.obj** para cada **.cpp**
- O linker junta esses arquivos em outros tipos
  - **EXE**: um arquivo executável contém código executável que pode ser executado diretamente pelo sistema operacional
  - **DLL**: um arquivo contendo código que pode ser compartilhado por vários programas. Ela contém funções, classes ou recursos que os programas podem usar
  - **SYS**: são arquivos de sistema utilizados pelo Windows para controlar dispositivos de hardware ou fornecer funcionalidades essenciais para o sistema operacional
  - **DRV**: um driver é um tipo específico de arquivo .sys que controla e permite a comunicação entre o sistema operacional e dispositivos de hardware, como impressoras, placas de vídeo, dispositivos USB, etc.
  - **LIB**: é usado para armazenar código e dados que podem ser usados por programas durante o processo de compilação e vinculação. Não usável pelo SO – apenas uma junção de obj
- O linker também usa, além do código de máquina
  - Variáveis globais, estáticas, TLS, Resources (Win), informação de debug, tabela de símbolos
  - Sem o linker, não poderíamos quebrar o código fonte em vários arquivos

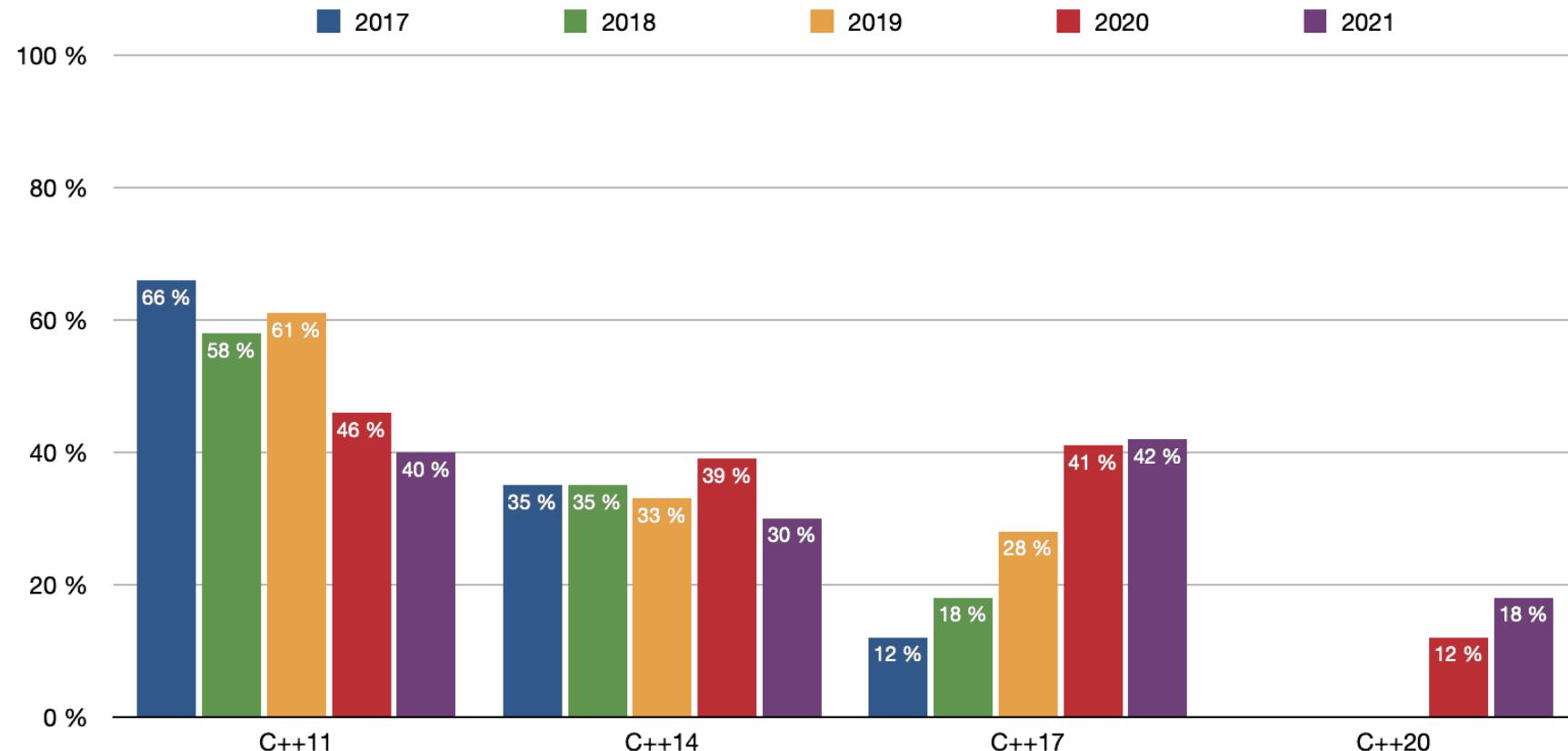
# C++: Linker

- Como o linker trabalha?
  - Abre todos os .obj
  - Verifica as funções não resolvidas internamente (em arquivos separados)
  - Procura essas funções em outros arquivos .obj ou .lib
  - Coloca essas funções dentro de uma executável apontando corretamente uma para outra (**resolução de símbolos**)
- **Link estático:** todos os códigos de todas as funções são copiados para dentro do executável (sem dependências externas de DLLs). Isso significa que o executável é autocontido e pode ser distribuído sem depender de bibliotecas externas. No entanto, isso pode levar a executáveis maiores e menos flexibilidade em atualizações.
- **Link dinâmico:** a fase de **resolução de símbolos** é carregada quando o executável é carregado pelo SO
  - Cada DLL tem uma tabela de símbolos exportada
  - Cada EXE tem uma tabela que contém os nomes das DLLs e das funções importadas

# C++: Versões

Versão	Data do Release	Maiores mudanças
C++98 (ISO/IEC 14882:1998)	Out 1998	a primeira versão
C++03 (ISO/IEC 14882:2003)	Fev 2003	Introduction of value initialization.
C++11	Ago 2011	Introdução de Expressões Lambda, Delegating Constructors, Uniform Initialization Syntax, nullptr, Automatic Type Deduction e decltype, Rvalue References etc.
C++14	Ago 2014	Introdução de lambdas polimórficos, separadores de dígitos, captura lambda generalizada, modelos de variáveis, literais inteiros binários, strings entre aspas, etc.
C++17	Dez 2017	Introdução de expressões de dobra, literais de ponto flutuante hexadecimal, um literal de caractere u8, instruções de seleção com inicializador, variáveis inline, etc.
C++20	Mar 2020	Esta atualização estende C++ com as facilidades para inspecionar entidades de programa como variáveis, enumerações, classes e seus membros, lambdas e suas capturas, etc.
C++23	Próxima Versão	A próxima grande revisão do padrão C++

# C++: Versões



# C x C++

C	C++
C criada por Dennis Ritchie entre 1969 e 1973 no AT&T Bell Labs.	C++ criada por Bjarne Stroustrup em 1979.
C não suporta programação orientada a objetos.	C++ suporta polimorfismo, encapsulamento e herança porque é uma linguagem de programação orientada a objetos.
C é (principalmente) um subconjunto de C++.	C++ é (principalmente) um superconjunto de C.
Número de <u>keywords</u> em C: * C90: 32 * C99: 37 * C11: 44 * C23: 59	Número de <u>keywords</u> em C++: * C++98: 63 * C++11: 73 * C++17: 73 * C++20: 81
Para o desenvolvimento de código, C oferece suporte à programação procedural.	C++ é conhecida como linguagem híbrida porque suporta os paradigmas de programação procedural e orientada a objetos.
Dados e funções são separados em C porque é uma linguagem de programação procedural.	Dados e funções são encapsulados juntos na forma de um objeto em C++.
C não suporta ocultação de informações.	Os dados são ocultados pelo encapsulamento para garantir que as estruturas de dados e os operadores sejam usados conforme pretendido.
Os tipos de dados integrados são suportados em C.	Tipos de dados integrados e definidos pelo usuário são suportados em C++.
C é uma linguagem orientada a funções porque C é uma linguagem de programação procedural.	C++ é uma linguagem orientada a objetos porque é uma programação orientada a objetos.
A sobrecarga de funções e operadores não é suportada em C.	A sobrecarga de funções e operadores é suportada em C++.
Os recursos de namespace não estão presentes dentro do C.	<u>Namespace</u> é usado por C++, o que evita colisões de nomes.
O cabeçalho de E/S padrão é stdio.h.	O cabeçalho de E/S padrão é iostream.h.
Variáveis de referência não são suportadas por C.	Variáveis de referência são suportadas por C++.
Funções virtual e friend não são suportadas por C.	Funções virtual e friend são suportadas por C++.
C não suporta herança.	C++ suporta herança.

# C x C++

C	C++
C fornece funções malloc() e calloc() para alocação dinâmica de memória e free() para desalocação de memória.	C++ fornece novo operador para alocação de memória e operador delete para desalocação de memória.
Suporte direto para manipulação de exceção não é suportado por C.	<u>O tratamento de exceções é compatível com C++.</u>
<u>As funções scanf() e printf() são usadas para entrada/saída em C.</u>	<u>cin e cout são usados para entrada/saída em C++.</u>
As estruturas C não têm modificadores de acesso.	As estruturas C++ têm modificadores de acesso.
C segue a abordagem de cima para baixo	C++ segue a abordagem Bottom-up
Não há verificação de tipo estrita na linguagem de programação C.	A verificação estrita de tipo é feita em C++. Programas que funcionam bem no compilador C resultarão em muitos avisos e erros no compilador C++.
C não suporta sobrecarga	C++ suporta sobrecarga
Os inicializadores nomeados podem aparecer fora de ordem	Os inicializadores nomeados devem corresponder ao layout de dados da estrutura
A extensão do arquivo é ".c"	A extensão do arquivo é ".cpp" ou ".c++" ou ".cc" ou ".cxx"
Metaprogramação: macros + _Generic()	Metaprogramação: modelos (macros ainda são suportados, mas desencorajados)
Existem 32 palavras-chave no C	Existem 97 palavras-chave no C++

# Meu primeiro programa C++

# Olá Mundo em C++ (Sem OO)

```
#include <iostream>           // Inclui a biblioteca para entrada/saída

int main() {                  // Função principal
    std::cout << "Olá, mundo!" << std::endl; // Imprime a mensagem
    return 0;                   // Retorna 0 para indicar sucesso
}
```

# Olá Mundo em C++ (Com OO)

```
#include <iostream>

// Definição da classe
class Saudacao {
public:
    void imprimirMensagem() {
        std::cout << "Olá, mundo!" << std::endl;
    }
};

int main() {
    Saudacao saudacao;           // Cria um objeto da classe Saudacao
    saudacao.imprimirMensagem(); // Chama o método para imprimir a
                                // mensagem
    return 0;
}
```

## Structure of C++ Program

1	<code>#include &lt;iostream&gt;</code>	Header File
2	<code>using namespace std;</code>	Standard Namespace
3	<code>int main()</code>	Main Function
FUNCTION BODY	4    {	
	5 <code>int num1 = 24;</code>	Declaration of Variable
	6 <code>int num2 = 34;</code>	
	7 <code>int result = num1 + num2;</code>	Expressions
	8 <code>cout &lt;&lt; result &lt;&lt; endl;</code>	Output
	9 <code>return 0;</code>	Return Statement
	10   }	

# Estrutura básica de um programa

```
// Inclusão de bibliotecas
#include <iostream>

// Uso do namespace padrão
using namespace std;

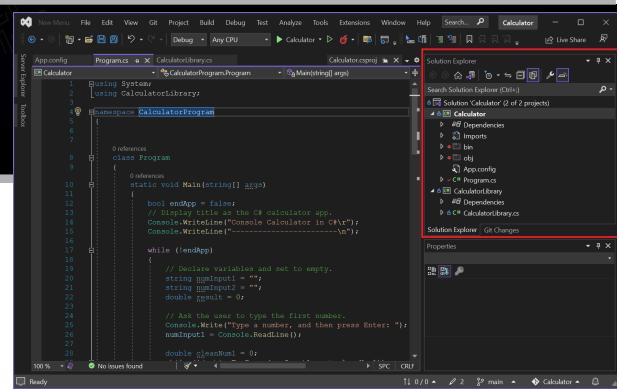
// Função principal
int main() {
    // Código do programa

    return 0; // Retorna um valor inteiro (0) para indicar sucesso ao
    sistema operacional
}
```

# Estrutura básica de um programa

- Os **arquivos de cabeçalho** contêm a definição das funções e macros que estamos usando em nosso programa. Eles são definidos na parte superior do programa C++.
- Um **namespace** em C++ é usado para fornecer um escopo ou uma região onde definimos identificadores. Ele é usado para evitar conflitos de nome entre dois identificadores, pois apenas nomes exclusivos podem ser usados como identificadores.

# Ambiente de Desenvolvimento



The screenshot shows the Microsoft Visual Studio IDE interface. The code editor displays a C++ file named 'CalculatorProgram.cs' with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using CalculatorLibrary;

namespace CalculatorProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            bool endApp = false;
            // Display title as the C# calculator app.
            Console.WriteLine("Console Calculator In C#(r)");
            Console.WriteLine("Type 'exit' to quit.(r)n");
            while (!endApp)
            {
                // Declare variables and set to empty.
                string numInput = "";
                string result = "";
                double result = 0;
                // Ask the user to type the first number.
                Console.Write("Type a number, and then press Enter: ");
                numInput = Console.ReadLine();
                double cleanNum = 0;
                if (double.TryParse(numInput, out cleanNum))
                {
                    if (cleanNum == 0)
                    {
                        result = "0";
                    }
                    else
                    {
                        result = calculate(cleanNum);
                    }
                }
                else
                {
                    result = "Error";
                }
                Console.WriteLine(result);
                // Ask the user if they want to exit.
                Console.Write("Type 'exit' to quit, or any other key to continue: ");
                numInput = Console.ReadLine();
                if (numInput.ToLower() == "exit")
                {
                    endApp = true;
                }
            }
        }

        static string calculate(double num)
        {
            string result = num.ToString();
            if (result.Contains("."))
            {
                result = result.Substring(0, result.IndexOf('.'));
            }
            return result;
        }
    }
}
```

- O **ambiente de desenvolvimento** é o conjunto de ferramentas e recursos empregados para escrever, compilar, depurar e testar o código
- Geralmente, inclui um editor de código, um compilador, um depurador e outras ferramentas relevantes.
- Exemplos:



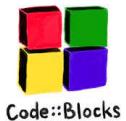
Visual Studio  
(Windows)



Visual Studio Code  
(Multiplataforma)



CLion  
(Multiplataforma)



Code::Blocks  
(Multiplataforma)



QT Creator  
(Multiplataforma)



Eclipse CDT  
(Multiplataforma)

# Diretivas de pré-processamento

- O pré-processador é uma etapa inicial do processo de compilação.
- Ele lida com **diretivas do pré-processador**
- As **diretivas do pré-processador** são comandos especiais inseridos no código-fonte C++ com o prefixo `#`, processadas antes da compilação propriamente dita e afetam o comportamento do código durante a compilação
- As principais diretivas são:
  - **#include**: usada para incluir um arquivo de cabeçalho no código-fonte
  - **#define**: usada para definir macros, que são substituições de texto. Isso é útil para criar atalhos para código repetitivo, constantes ou funções simples
  - **#ifdef, #ifndef, #else, #endif**: usadas para realizar compilação condicional. Essas diretivas podem ser empregadas para incluir ou excluir blocos de código com base em condições predefinidas
  - **#error**: usada para gerar um erro de compilação com uma mensagem personalizada. Isso pode ser útil para impor certas restrições ou condições em seu código.
  - **#pragma**: usada para fornecer informações específicas ao compilador ou alterar seu comportamento. As pragmas não são padrões, então seu comportamento pode variar entre compiladores.
  - **#undef**: usada para desdefinir uma macro definida anteriormente usando `#define`. Isso permite remover uma macro e seus efeitos no código.

# Diretivas de pré-processamento

```
#include <iostream>
// Definindo uma macro para calcular o quadrado de um número
#define QUADRADO(x) ((x) * (x))

// Checa se a macro DEBUG está definida
#ifndef DEBUG
    #define DEBUG_MESSAGE "Modo de depuração ativado"
#else
    #define DEBUG_MESSAGE "Modo de depuração desativado"
#endif

int main() {
    int num = 5;
    // Utilizando a macro QUADRADO
    std::cout << "O quadrado de " << num << " é: " << QUADRADO(num) << std::endl;

    // Exibindo mensagem de depuração condicionalmente
    std::cout << DEBUG_MESSAGE << std::endl;

    // Checa se a macro OUTRO_DEBUG está definida
#ifndef OUTRO_DEBUG
    std::cout << "OUTRO_DEBUG não está definido" << std::endl;
#endif
    return 0;
}
```

# Compilação Condicional

- A compilação condicional em C++ é realizada usando diretivas do pré-processador, como `#ifdef`, `#ifndef`, `#else` e `#endif`
- Isso é particularmente útil para criar versões diferentes de um programa, com partes do código que são ativadas ou desativadas dependendo das condições especificadas.

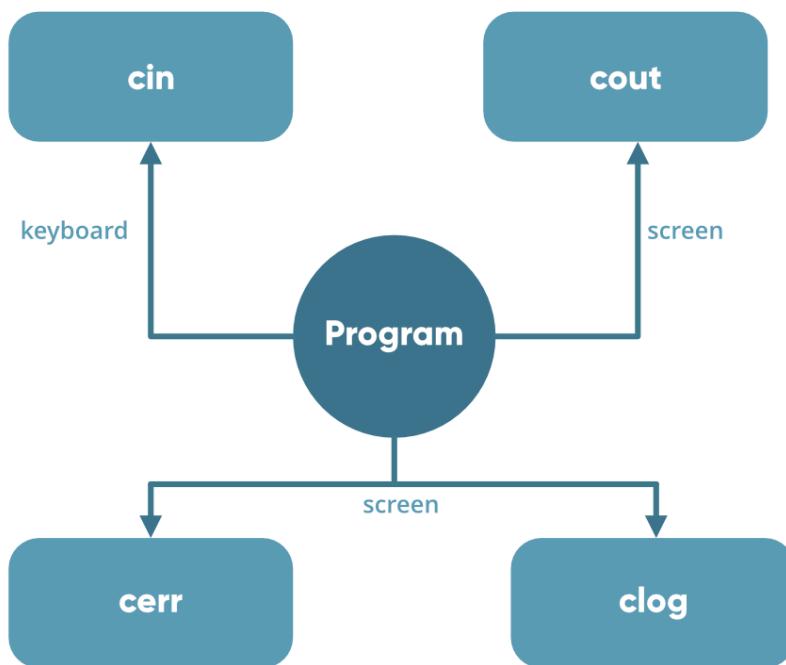
```
#ifndef RELEASE
    // Código ativado quando RELEASE não está definido
    std::cout << "Esta é uma versão de desenvolvimento" << std::endl;
#else
    // Código ativado quando RELEASE está definido
    std::cout << "Esta é uma versão de lançamento" << std::endl;
#endif

#ifndef DEBUG
    // Código ativado quando DEBUG está definido
    std::cout << "Modo de depuração ativado" << std::endl;
#elif defined(TEST)
    // Código ativado quando TEST está definido, mas DEBUG não está definido
    std::cout << "Modo de teste ativado" << std::endl;
#else
    // Código ativado quando nem DEBUG nem TEST estão definidos
    std::cout << "Modo padrão" << std::endl;
#endif
```

# E/S em C++

# Entrada e Saída Básica

- Em C++, a entrada e a saída são executadas na forma de uma sequência de bytes ou mais comumente conhecida como fluxos.
  - **Fluxo de entrada:** Se a direção do fluxo de bytes for do dispositivo (por exemplo, teclado) para a memória principal, esse processo é chamado de entrada.
  - **Fluxo de saída:** Se a direção do fluxo de bytes for oposta, ou seja, da memória principal para o dispositivo (tela de exibição), esse processo é chamado de saída.



# Entrada e Saída Básica

- **cin (Console Input):** é um objeto da classe **istream** e é usado para receber dados de entrada do usuário via teclado. Ele é comumente usado junto com o operador de extração **>>** para ler valores de diferentes tipos (como inteiros, floats, strings, etc.) da entrada padrão.

```
int x;
std::cout << "Digite um número: ";
std::cin >> x;
std::cout << "Você digitou: " << x << std::endl;
```

- **cout (Console Output):** é um objeto da classe **ostream** e é usado para exibir dados na saída padrão (geralmente no console). Ele é comumente usado junto com o operador de inserção **<<** para imprimir valores de diferentes tipos.

```
int y = 42;
std::cout << "O valor de y é: " << y << std::endl;
```

# Entrada e Saída Básica

- **unbuffered standard cerr (Console Error):** é um objeto da classe **ostream** que também é usado para exibir mensagens na saída padrão de erro. Diferentemente de cout, cerr é usado para mensagens de erro ou saída que não devem ser redirecionadas ou intercaladas com saída normal.

```
int divisor = 0;
if (divisor == 0) {
    std::cerr << "Erro: Divisão por zero!" << std::endl;
}
```

- **buffered standard clog (Console Log):** é um objeto da classe **ostream** usado para exibir erros. Ao contrário de cerr, o erro é inserido primeiro em um buffer e armazenado no buffer até que o mesmo não esteja cheio ou que não seja liberado explicitamente (usando flush()). A mensagem de erro será exibida na tela também

```
double result = calculateSomething();
std::clog << "Resultado do cálculo: " << result << std::endl;
```

# Entrada e Saída Básica

- **Alguns métodos de cin**
- **>>** Operador de Extração
  - **>>** sempre retorna uma referência ao objeto cin

```
float c;
float d;
cin >> c >> d;    // entra com um float, pressiona <ENTER>, entra com outro float,
pressiona <ENTER>
```

- **getline()** é usado para ler uma linha inteira da entrada, incluindo espaços em branco. Ele é útil para ler strings.

```
char nome[5];

// Lê uma string com 3 caracteres
cin.getline(nome, 3);
```

- **ignore()** é usado para descartar um número específico de caracteres da entrada, incluindo espaços em branco.

```
std::cin.ignore(100, '\n'); // Descarta os próximos 100 caracteres ou até encontrar
uma quebra de linha
```

# Entrada e Saída Básica

- Os manipuladores são funções auxiliares que podem modificar o fluxo de entrada/saída. Isso não significa que alteramos o valor de uma variável, apenas modificamos o fluxo de I/O usando operadores de inserção (<<) e extração (>>).
- **endl**: gera um caractere nova linha, e também descarrega o buffer de saída
- **setw (val)**: usado para definir a largura do campo nas operações de saída.
- **setfill(c)**: usado para preencher o caracter ‘c’ no fluxo de saída.
- **setprecision(val)**: define val como o novo valor para a precisão dos valores de ponto flutuante.
- **setbase(val)**: usado para definir o valor base numérico para valores numéricos.
- **setiosflags(flag)**: usado para definir os sinalizadores de formato especificados pelo parâmetro mask.
- **resetiosflags(m)**: usado para redefinir os sinalizadores de formato especificados pelo parâmetro mask.

# Entrada e Saída Básica

```
#include <iostream>
#include <iomanip>

using namespace std;
main()
{
    int number = 100;
    cout << "Hex Value =" << " " << hex << number << endl;
    cout << "Octal Value=" << " " << oct << number << endl;
    cout << "Setbase Value=" << " " << setbase(8) << number << endl;
    cout << "Setbase Value=" << " " << setbase(16) << number << endl;
    return 0;
}
```

# Entrada e Saída Básica

- **Alguns métodos de cout**
- **<< Operador de Inserção**
  - << sempre retorna uma referência ao objeto cout
- Formatação

```
#include <iostream>
#include <iomanip> // Para manipuladores de saída de formatação

int main() {
    double valor = 123.456789;

    std::cout << std::fixed << std::setprecision(2);      // Fixa a quantidade de casas decimais
    std::cout << "Valor formatado: " << valor << std::endl;

    return 0;
}
```

# Entrada e Saída Básica

## Alguns métodos de cout

- Alinhamento e largura

```
#include <iostream>
#include <iomanip>

int main() {
    int numero = 42;

    std::cout << std::setw(10) << std::left << numero << "Texto" << std::endl;

    return 0;
}
```

```
int numero = 42;
std::cout.width(10);
std::cout.fill('*'); // Define o caractere de preenchimento como '*'
std::cout << "Número: " << numero << std::endl;
```

# C++: Sintaxe

# Variáveis: Tipos de dados

- **Tipos Inteiros:**
  - **int**: Representa números inteiros, por exemplo, 5, -10, 1000.
  - **short**: Inteiro curto, geralmente com menos bits do que um int.
  - **long**: Inteiro longo, geralmente com mais bits do que um int.
  - **long long**: Inteiro longo longo, com ainda mais bits do que long.
- **Tipos Ponto Flutuante:**
  - **float**: Representa números de ponto flutuante de precisão simples.
  - **double**: Representa números de ponto flutuante de precisão dupla (mais precisos que float).
  - **long double**: Representa números de ponto flutuante de maior precisão do que double.
- **Tipos Caractere:**
  - **char**: Armazena um caractere individual (um byte).
  - **wchar\_t**: Armazena um caractere "wide" (mais de um byte, usado para lidar com caracteres Unicode).
  - **char16\_t**: Armazena um caractere UTF-16.
  - **char32\_t**: Armazena um caractere UTF-32.

# Variáveis: Tipos de dados

- **Tipo Booleano:**
  - **bool**: Armazena valores booleanos true (verdadeiro) ou false (falso).
- **Tipos Compostos:**
  - **enum**: Define um tipo enumerado com um conjunto de valores constantes.
  - **struct**: Define uma estrutura que agrupa variáveis com diferentes tipos sob um único nome.
  - **union**: Define um tipo de união que permite armazenar diferentes tipos de dados em uma única variável.
- **Tipo Ponto Flutuante com Precisão Arbitrária:**
  - float e double podem ser usados com modificadores long e unsigned para definir diferentes intervalos de valores inteiros.
- **Tipos de Ponto Fixo:**
  - O C++20 introduziu tipos de ponto fixo, como std::fixed\_point, para representar números reais com ponto fixo.
- **Ponteiros e Tamanhos de Tipos:**
  - nullptr: Valor nulo para ponteiros.
  - sizeof: Operador para obter o tamanho em bytes de um tipo.

# Variáveis: Tipos de dados

```
#include <iostream>

// Definição de enum para representar os dias da semana
enum DiasDaSemana {DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO};

// Definição de uma struct para representar informações de uma pessoa
struct Pessoa {
    std::string nome;
    int idade;
};

// Definição de uma union para representar um valor que pode ser int ou float
union ValorInteiroOuFloat {
    int valorInt;
    float valorFloat;
};

int main() {
    DiasDaSemana dia = TERCA;
    std::cout << "Dia da semana: " << dia << std::endl;

    Pessoa pessoa;
    pessoa.nome = "João";
    pessoa.idade = 30;
    std::cout << "Nome: " << pessoa.nome << ", Idade: " << pessoa.idade << " anos" << std::endl;

    ValorInteiroOuFloat valor;
    valor.valorInt = 5;
    std::cout << "Valor Inteiro: " << valor.valorInt << std::endl;

    valor.valorFloat = 3.14;
    std::cout << "Valor Float: " << valor.valorFloat << std::endl;

    return 0;
}
```

# Conversão de Tipo

- As conversões de tipos referem-se à maneira como os tipos são convertidos de um para outro, seja implicitamente pelo compilador ou explicitamente pelo programador.
  - Conversões Implícitas:** O compilador realiza automaticamente a conversão entre tipos compatíveis.

```
int valorInteiro = 5;  
double valorDouble = valorInteiro; // Conversão implícita de int para double
```

- Conversões Explícitas:** O programador usa operadores ou funções para indicar ao compilador para fazer a conversão.

```
double valorDouble = 3.14;  
  
// Conversão explícita de double para int  
int valorInteiro = static_cast<int>(valorDouble);
```

# Conversão de Tipo

- **dynamic\_cast:**
  - Usado principalmente para conversões entre tipos de classes (herança).
  - Realiza verificação em tempo de execução para garantir que a conversão seja segura em termos de herança.
  - Retorna um ponteiro ou referência nula se a conversão falhar em tempo de execução.

```
class Base {  
    // ...  
};  
class Derivada : public Base {  
    // ...  
};  
int main() {  
    Base *ptrBase = new Derivada;  
    Derivada *ptrDerivada = dynamic_cast<Derivada*>(ptrBase);  
    if (ptrDerivada) {  
        // A conversão foi bem-sucedida  
    } else {  
        // A conversão falhou  
    }  
    delete ptrBase;  
    return 0;  
}
```

# Conversão de Tipo

- **const\_cast**:

- Usado para adicionar ou remover a qualificação `const` ou `volatile` de um tipo.
- Não deve ser usado para contornar a imutabilidade, pois isso pode levar a comportamento indefinido.
- Útil quando você deseja modificar a constância de um objeto de maneira controlada.

```
const int valorConst = 10;
int *ptrNaoConst = const_cast<int*>(&valorConst);
*ptrNaoConst = 20; // Isso é comportamento indefinido!
```

# Conversão de Tipo

- **reinterpret\_cast**

- Realiza conversões de tipo que podem não ter qualquer relação semântica.
- Realiza uma conversão bit a bit entre os tipos, sem verificar semântica ou segurança.
- Geralmente usado para converter ponteiros de tipos diferentes, como ponteiros para números inteiros e vice-versa.

```
int numero = 42;
double *ptrDouble = reinterpret_cast<double*>(&numero);
```

# Variáveis: Escopo

- **Variáveis Locais:**

- São declaradas dentro de uma função ou bloco e existem apenas durante a execução desse bloco.
- Seu ciclo de vida está limitado ao escopo onde foram declaradas.

```
#include <iostream>

void funcaoExemplo() {
    int variavelLocal = 10; // Declaração de uma variável local

    std::cout << "Valor da variável local: " << variavelLocal << std::endl;
} // A variável local "variavelLocal" sai de escopo e é destruída aqui

int main() {
    funcaoExemplo(); // Chamada da função
    // A variável local "variavelLocal" não é acessível aqui, pois está fora
    de escopo

    return 0;
}
```

# Variáveis: Escopo

- **Variáveis Globais:**

- São declaradas fora de qualquer função ou bloco e têm um escopo global em todo o programa.
- Elas existem durante toda a execução do programa.

```
#include <iostream>

// Declaração de uma variável global
int variavelGlobal = 20;

void funcaoExemplo() {
    std::cout << "Valor da variável global dentro da função: " << variavelGlobal << std::endl;
}

int main() {
    std::cout << "Valor da variável global dentro do main: " << variavelGlobal << std::endl;

    funcaoExemplo(); // Chamada da função que acessa a variável global

    variavelGlobal = 30; // Modificando o valor da variável global

    std::cout << "Novo valor da variável global: " << variavelGlobal << std::endl;

    return 0;
}
```

# Variáveis: Escopo

- **Variáveis Membro de Classe:**

- São variáveis que pertencem a uma classe e são acessíveis por meio de objetos dessa classe.
- Também conhecidas como variáveis de instância.

```
#include <iostream>

class MinhaClasse {
public:
    int variavelDeMembro; // Variável de membro pública
    void definirValor(int valor) {
        variavelDeMembro = valor;
    }
    void mostrarValor() {
        std::cout << "Valor da variável de membro: " << variavelDeMembro << std::endl;
    }
};

int main() {
    MinhaClasse objeto;
    objeto.variavelDeMembro = 10; // Atribuição direta à variável de membro
    objeto.mostrarValor();

    objeto.definirValor(25);      // Atribuição por meio de um método
    objeto.mostrarValor();

    return 0;
}
```

# Variáveis: Escopo

- **Variáveis Estáticas:**

- São variáveis que mantêm seu valor entre chamadas de função e têm um ciclo de vida mais longo.
- Em contextos globais, elas têm escopo restrito ao arquivo em que foram definidas.

```
#include <iostream>

class MinhaClasse {
public:
    static int variavelEstatica; // Variável estática
    void incrementarEstatica() {
        variavelEstatica++;
    }
    void mostrarEstatica() {
        std::cout << "Variável estática: " << variavelEstatica << std::endl;
    }
};
// Inicialização da variável estática fora da classe
int MinhaClasse::variavelEstatica = 0;

int main() {
    MinhaClasse objeto1;
    MinhaClasse objeto2;
    objeto1.incrementarEstatica();
    objeto1.mostrarEstatica();
    objeto2.incrementarEstatica();
    objeto2.mostrarEstatica();
    return 0;
}
```

# Variáveis: Escopo

- **Parâmetros de Valor:**

- Os parâmetros de valor permitem que você passe cópias dos valores originais para a função.
- As alterações feitas nos parâmetros dentro da função não afetam os valores originais fora da função.

```
#include <iostream>

void dobrar(int numero) {
    numero *= 2;
    std::cout << "Dobro dentro da função: " << numero << std::endl;
}

int main() {
    int valor = 5;
    dobrar(valor);
    std::cout << "Valor fora da função: " << valor << std::endl;
    return 0;
}
```

# Variáveis: Escopo

- **Parâmetros de Referência:**

- Os parâmetros de referência permitem que você passe referências aos valores originais para a função.
- As alterações feitas nos parâmetros de referência dentro da função afetam diretamente os valores originais fora da função.

```
#include <iostream>

void dobrarPorReferencia(int &numero) {
    numero *= 2;
    std::cout << "Dobro dentro da função: " << numero << std::endl;
}

int main() {
    int valor = 5;
    dobrarPorReferencia(valor);
    std::cout << "Valor fora da função: " << valor << std::endl;
    return 0;
}
```

# Variáveis: Escopo

- **Parâmetros Default:**
  - Permitem valores padrão para os parâmetros de uma função

```
#include <iostream>

void saudacao(const std::string &nome = "Amigo") {
    std::cout << "Olá, " << nome << "!" << std::endl;
}

int main() {
    saudacao();           // Usará o valor padrão "Amigo"
    saudacao("Alice");   // Usará o valor "Alice"
    saudacao("Bob");     // Usará o valor "Bob"
    return 0;
}
```

# Constantes

- **Usando const:**

```
#include <iostream>

int main() {
    const int numeroConstante = 10; // Declaração de uma constante usando const
    std::cout << "Valor da constante: " << numeroConstante << std::endl;
    // Tente modificar a constante (isso resultará em um erro)
    // numeroConstante = 20;

    return 0;
}
```

- **Usando constexpr (a partir do C++11):**

- implica que a constante será calculada em tempo de compilação sempre que possível.

```
#include <iostream>

int main() {
    constexpr int numeroConstante = 10; // Declaração de uma constante usando constexpr
    std::cout << "Valor da constante: " << numeroConstante << std::endl;
    return 0;
}
```

# Operadores Aritméticos

Operador	Uso	Descrição
+	a + b	Soma <b>a</b> e <b>b</b>
-	a - b	Subtrai <b>b</b> de <b>a</b>
-	-a	Nega aritimeticamente <b>a</b>
*	a * b	Multiplica <b>a</b> e <b>b</b>
/	a / b	Divide <b>a</b> por <b>b</b>
%	a % b	Retorna o resto da divisão de <b>a</b> por <b>b</b> (o operador de módulo)
++	a++	Incrementa <b>a</b> de 1; usa o valor de <b>a</b> antes de incrementar
++	++a	Incrementa <b>a</b> de 1; usa o valor de <b>a</b> depois de incrementar
--	a--	Decrementa <b>a</b> de 1; usa o valor de <b>a</b> antes de incrementar
--	--a	Decrementa <b>a</b> de 1; usa o valor de <b>a</b> depois de incrementar
+=	a += b	Abreviação de $a = a + b$
-=	a -= b	Abreviação de $a = a - b$
*=	a *= b	Abreviação de $a = a * b$
/=	a /= b	Abreviação de $a = a / b$
%=	a %= b	Abreviação de $a = a \% b$

# Operadores Lógicos e Relacionais

Operador	Uso	Retorna true se...
>	a > b	a for maior que b
>=	a >= b	a é maior que ou igual a b
<	a < b	a é menor que b
<=	a <= b	a é menor que ou igual a b
==	a == b	a é igual a b
!=	a != b	a é diferente a b
&&	a && b	a e b são true. Condicionalmente avalia b (se a for false, b não será avaliado)
	a    b	a ou b é true; condicionalmente avalia b (se a for true, b não será avaliado)
!	!a	a é false
&	a & b	realiza uma operação AND bit a bit entre cada bit correspondente de dois operandos.
	a   b	realiza uma operação OR bit a bit entre cada bit correspondente de dois operandos.
^	a ^ b	realiza uma operação XOR bit a bit entre cada bit correspondente de dois operandos.

# Operadores Lógicos e Relacionais

```
#include <iostream>

int main() {
    // Exemplo de operações bit a bit
    int a = 5;      // Representado em binário: 0101
    int b = 3;      // Representado em binário: 0011

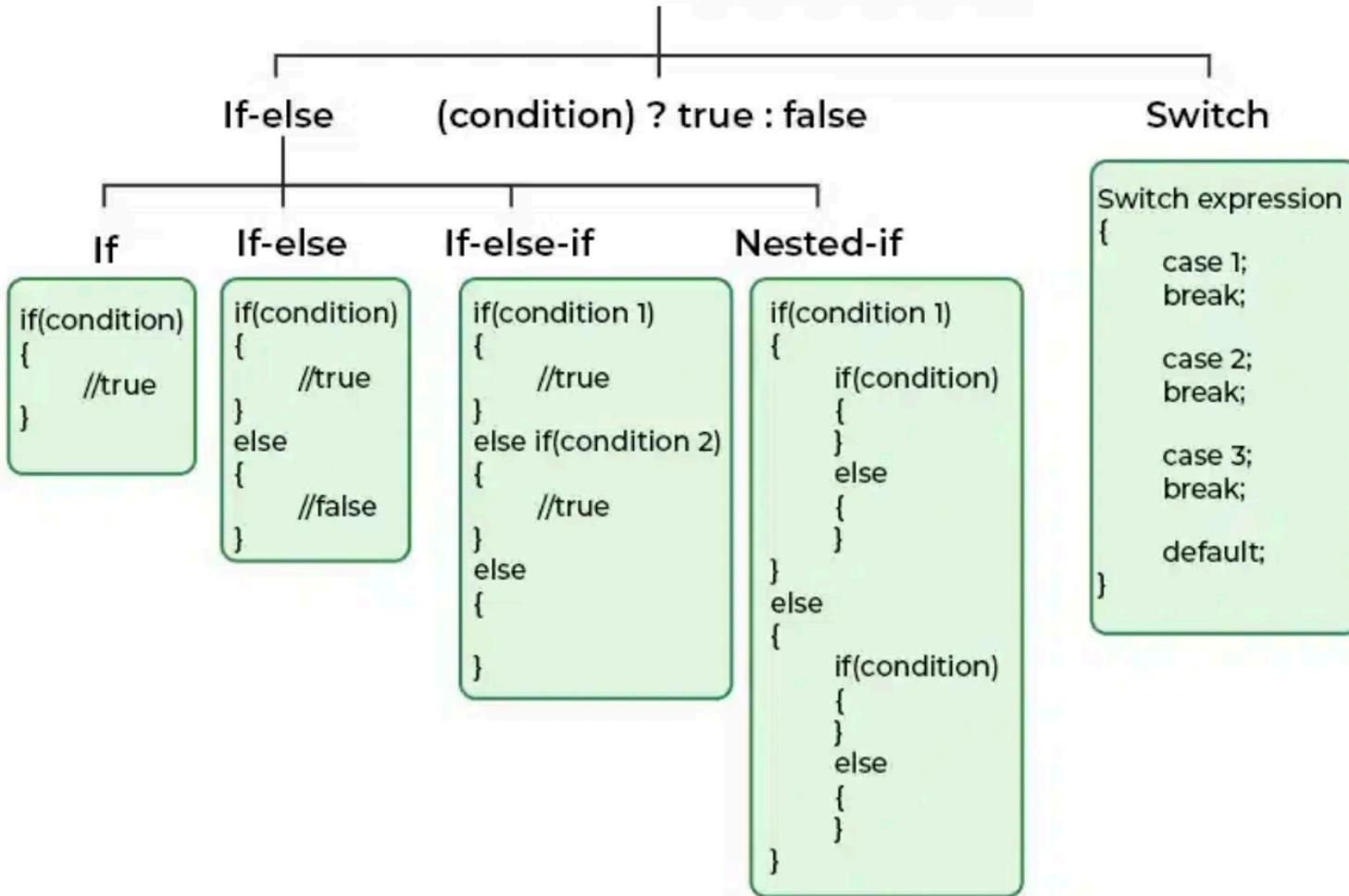
    int bitwise_and = a & b;    // Operação AND bit a bit
    int bitwise_or = a | b;    // Operação OR bit a bit
    int bitwise_xor = a ^ b;   // Operação XOR bit a bit
    int bitwise_not_a = ~a;    // Operação NOT bit a bit em 'a'

    std::cout << "Bitwise AND: " << bitwise_and << std::endl;    // Saída: 1 (binário: 0001)
    std::cout << "Bitwise OR: " << bitwise_or << std::endl;     // Saída: 7 (binário: 0111)
    std::cout << "Bitwise XOR: " << bitwise_xor << std::endl;   // Saída: 6 (binário: 0110)
    std::cout << "Bitwise NOT of 'a': " << bitwise_not_a << std::endl; // Saída: -6
    (binário: 11111111111111111111111111111111010)

    return 0;
}
```

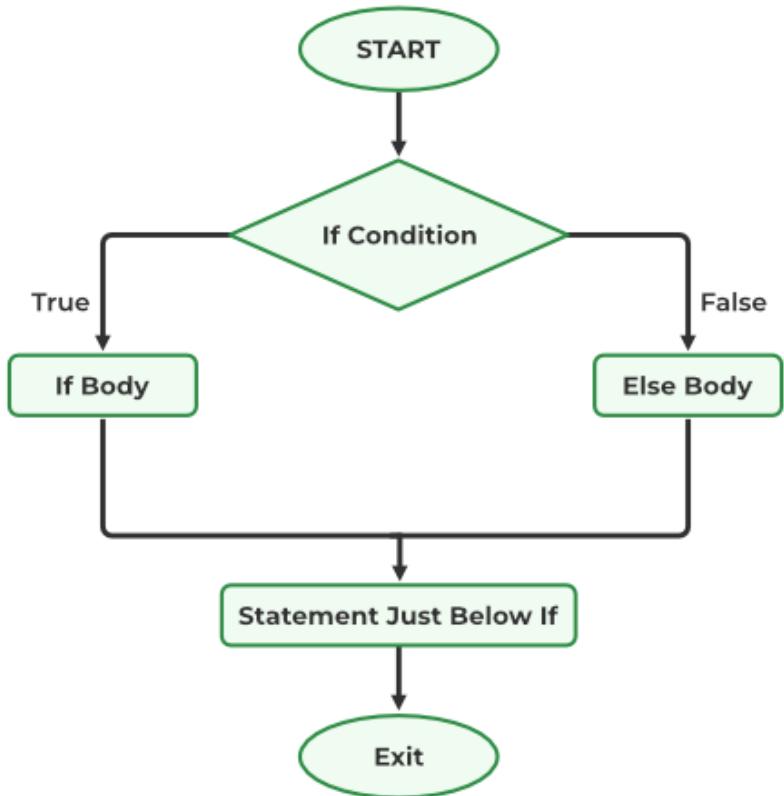
# Controle de Fluxo

# Controle de Fluxo



Fonte: <https://www.geeksforgeeks.org/>

# Controle de Fluxo



```
#include <iostream>

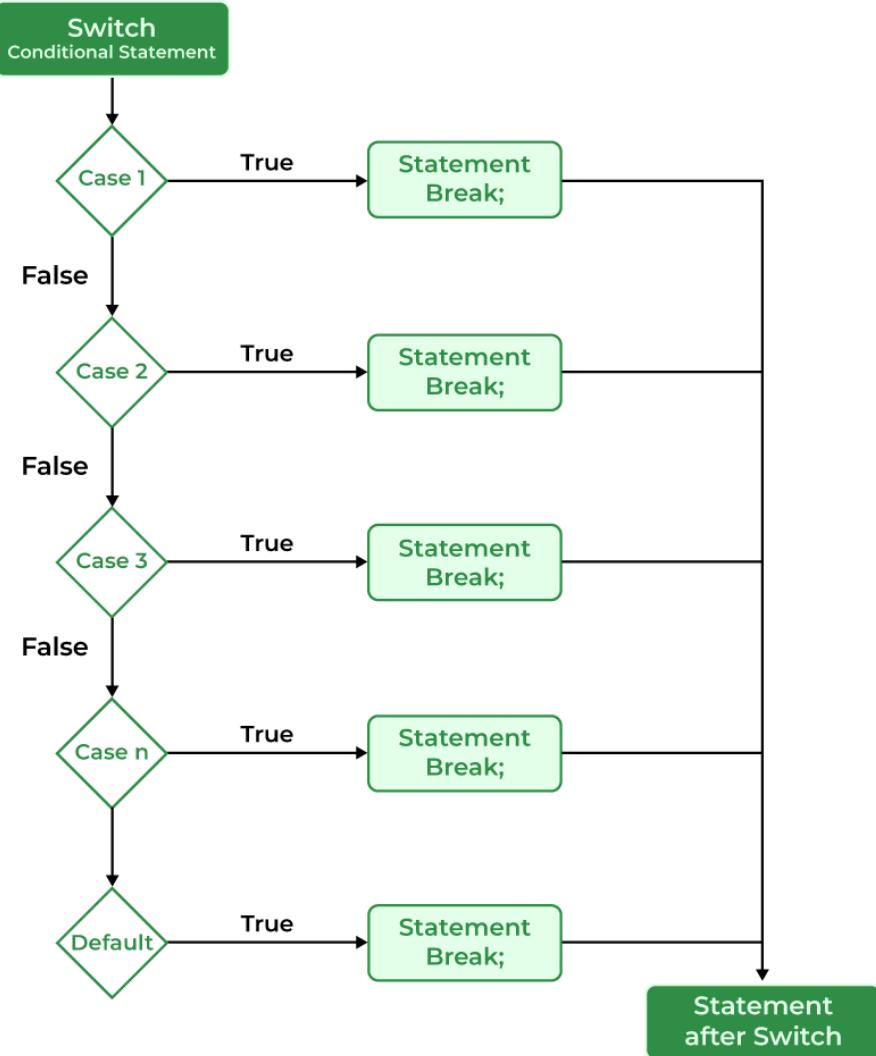
int main() {
    int idade;

    std::cout << "Digite sua idade: ";
    std::cin >> idade;

    if (idade >= 18) {
        std::cout << "Você é maior de idade." <<
std::endl;
    } else {
        std::cout << "Você é menor de idade." <<
std::endl;
    }

    return 0;
}
```

# Controle de Fluxo



```
#include <iostream>

int main() {
    char operacao;
    double num1, num2;

    std::cout << "Digite a operação (+, -): ";
    std::cin >> operacao;

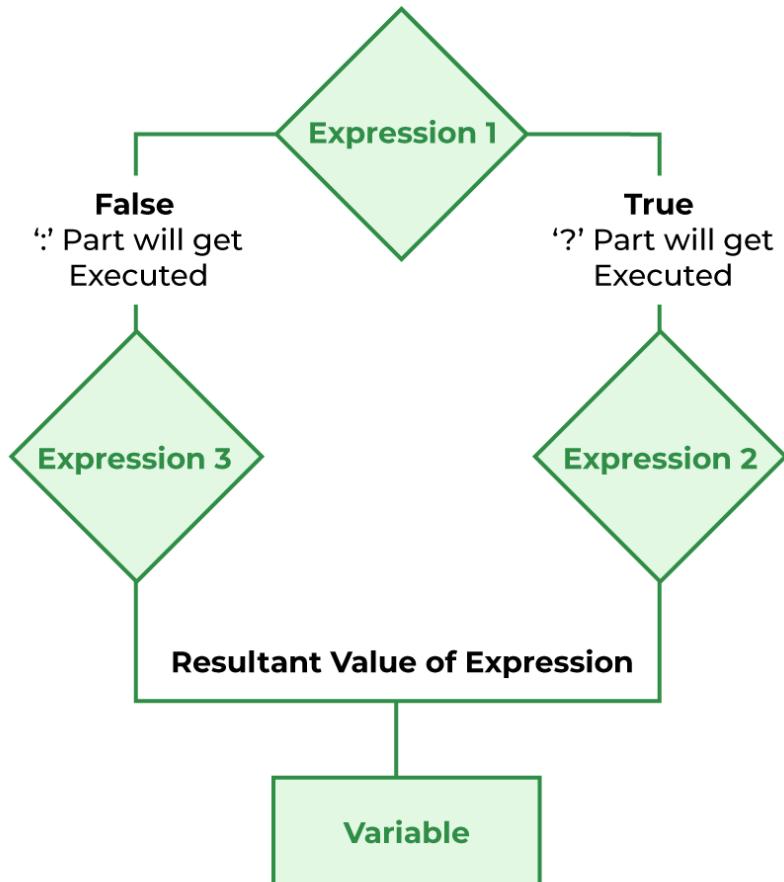
    std::cout << "Digite dois números: ";
    std::cin >> num1 >> num2;

    switch (operacao) {
        case '+':
            std::cout << "Resultado: " << num1 + num2
<< std::endl;
            break;
        case '-':
            std::cout << "Resultado: " << num1 - num2
<< std::endl;
            break;
        default:
            std::cout << "Operação inválida!" <<
std::endl;
            break;
    }

    return 0;
}
```

Fonte: <https://www.geeksforgeeks.org/>

# Controle de Fluxo



```
#include <iostream>

int main() {
    int numero;

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

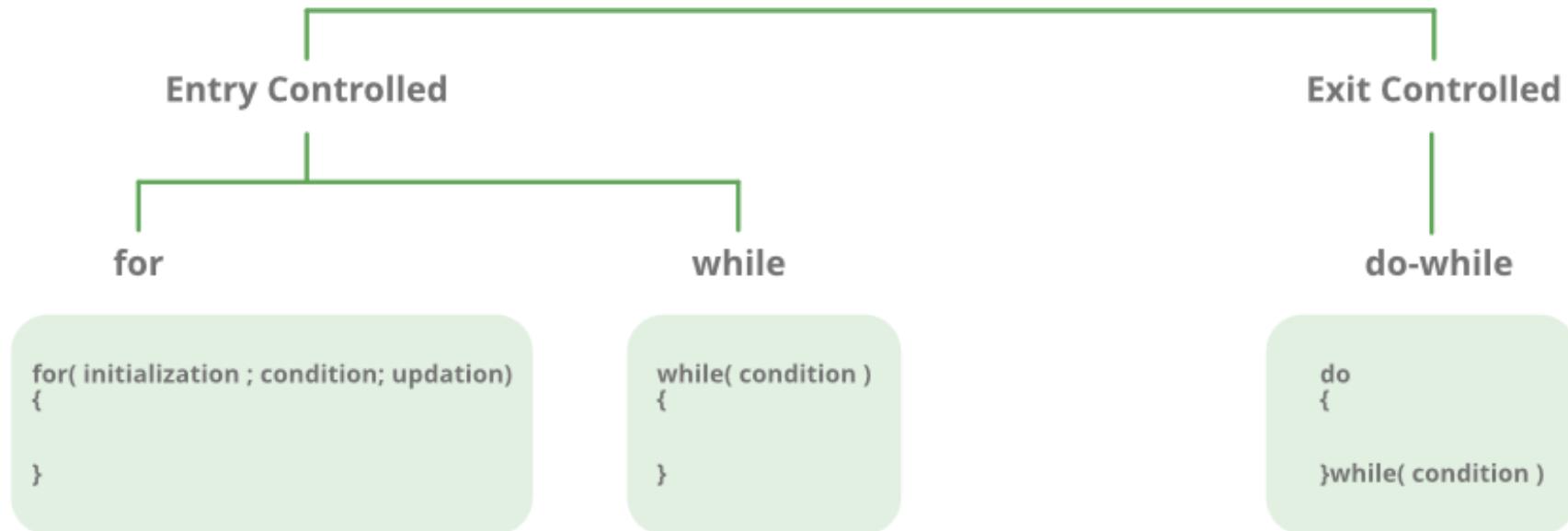
    // Usando o operador condicional para determinar se o número é positivo
    // ou negativo
    std::string resultado = (numero >= 0) ? "positivo" : "negativo";

    std::cout << "O número é " << resultado << "." << std::endl;

    return 0;
}
```

# Loops

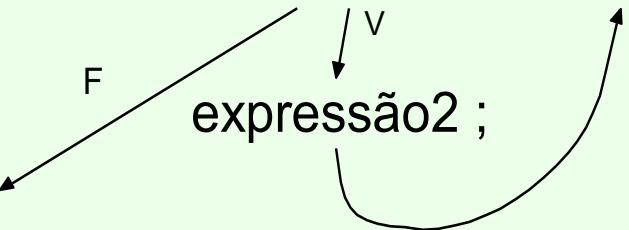
# Loops



Fonte: <https://www.geeksforgeeks.org/>

# Loops

```
for (expressão1; condição; expressão3)
```



```
#include <iostream>

int main() {
    int limite;

    std::cout << "Digite um número
limite: ";
    std::cin >> limite;

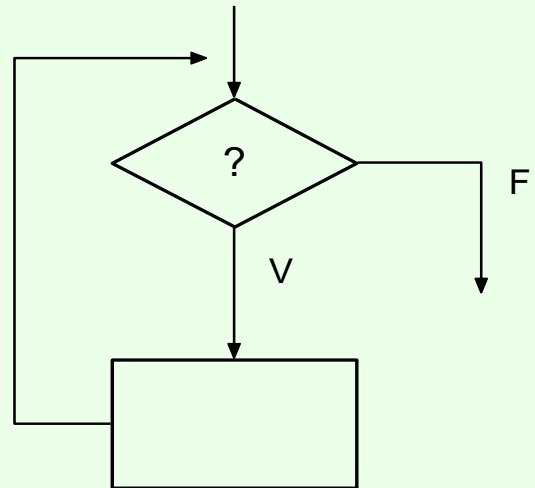
    // Usando um loop for para imprimir
    os números de 1 até o limite
    for (int i = 1; i <= limite; ++i) {
        std::cout << i << " ";
    }

    std::cout << std::endl;
}

return 0;
}
```

- **expressão1:** uma expressão ou qualquer comando ou chamada de função. Normalmente uma atribuição;
- **expressão2:** qualquer comando ou chamada de função;
- **expressão3:** uma expressão ou qualquer comando ou chamada de função. Normalmente um incremento;
- Todos são opcionais

# Loops



```
#include <iostream>

int main() {
    int contador = 1;
    int limite;

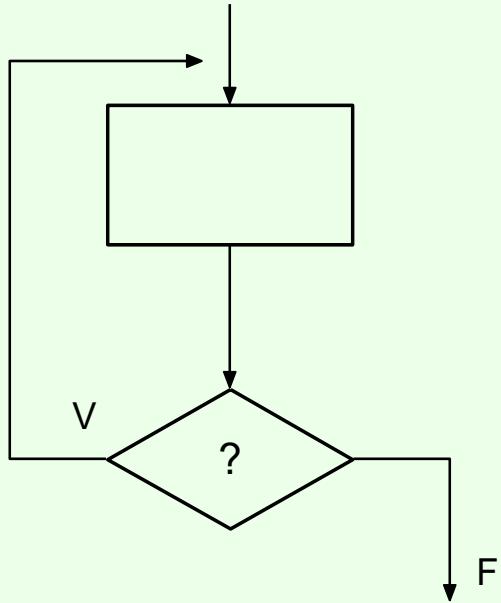
    std::cout << "Digite um número limite: ";
    std::cin >> limite;

    // Usando um loop while para imprimir os números de 1
    // até o limite
    while (contador <= limite) {
        std::cout << contador << " ";
        ++contador;
    }

    std::cout << std::endl;

    return 0;
}
```

# Loops



```
#include <iostream>

int main() {
    int contador = 1;
    int limite;

    std::cout << "Digite um número limite: ";
    std::cin >> limite;

    // Usando um loop do/while para imprimir os números de
    // 1 até o limite
    do {
        std::cout << contador << " ";
        ++contador;
    } while (contador <= limite);

    std::cout << std::endl;

    return 0;
}
```

- Na próxima aula
  - Matrizes e vetores
  - Alocação de memória
  - Funções e procedimentos em C++
  - Funções lambda
  - Value categories