

Introdução C++

ELC1067 - Laboratório de Programação II

João Vicente Ferreira Lima (UFSM)

Universidade Federal de Santa Maria
jvlima@inf.ufsm.br
<http://www.inf.ufsm.br/~jvlima>

2023/1

Outline

- 1 Introdução
- 2 Variáveis
- 3 Containers STL
- 4 Entrada e saída
- 5 Escopos
- 6 Tratamento de erros

Outline

- 1 Introdução
- 2 Variáveis
- 3 Containers STL
- 4 Entrada e saída
- 5 Escopos
- 6 Tratamento de erros

Unix history

UNIX was written in assembly by Ken Thompson at Bell Labs (AT&T) in 1969 for a Digital PDP-7 minicomputer. It took several ideas from MULTICS such as:

- tree-structured file system.
- separate program for interpreting commands (shell).
- notion of files as unstructured streams of bytes.



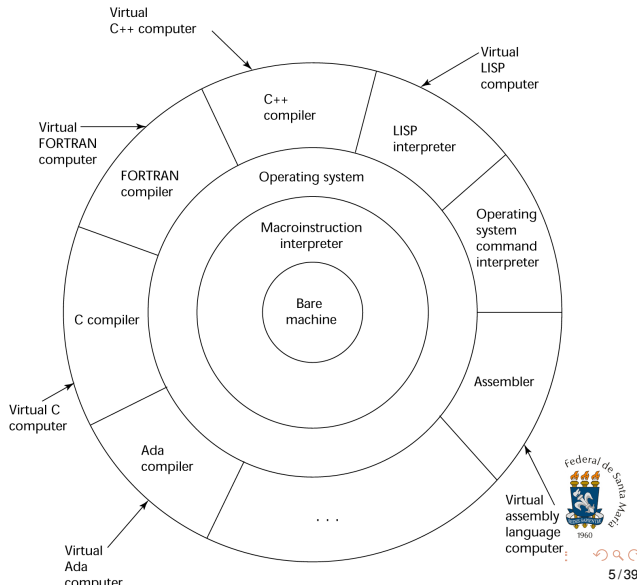
Dennis Ritchie designed and implemented C language between 1969-1973 at Bell Labs. By 1973, UNIX was almost totally written in C.

C for system programming

In the 70s, widely used languages were designed to other purposes:

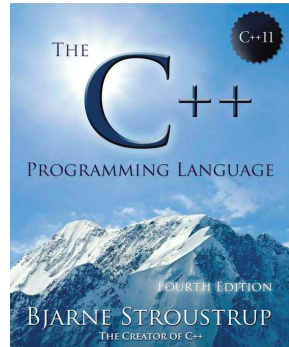
- FORTRAN for mathematical tasks
- COBOL for commercial systems

O **sistema operacional** e as **linguagens de programação** criam uma interface ao computador

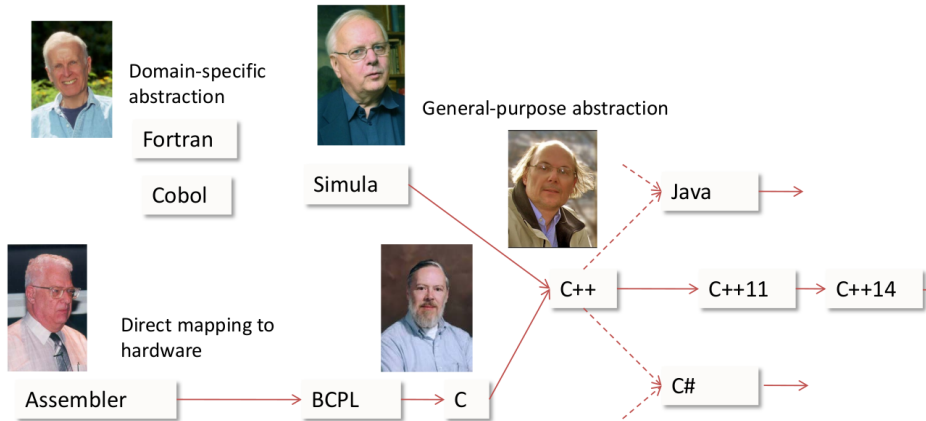


História do C++

- Criado em 1979 como *C with Classes* por Bjarne Stroustrup
- C++ é compatível com C
- Abstrações sem comprometer o desempenho
- 1985: primeira edição do livro *The C++ Programming Language*
- 1994: STL (*Standard Template Library*) por Alexander Stepanov
- 1998: ISO C++ standard.



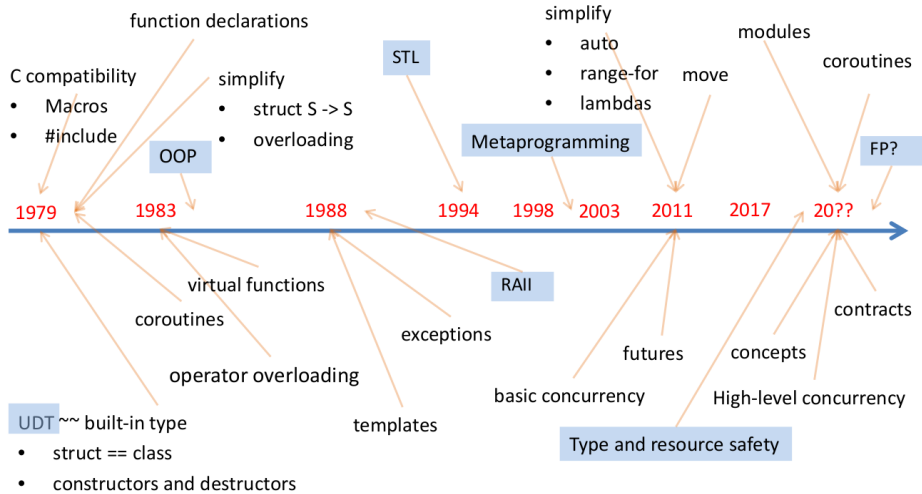
História do C++



Bjarne Stroustrup, *The Evolution of C++: Past, Present, and Future*, CppCon 2016

https://youtu.be/_wzc7a3McOs

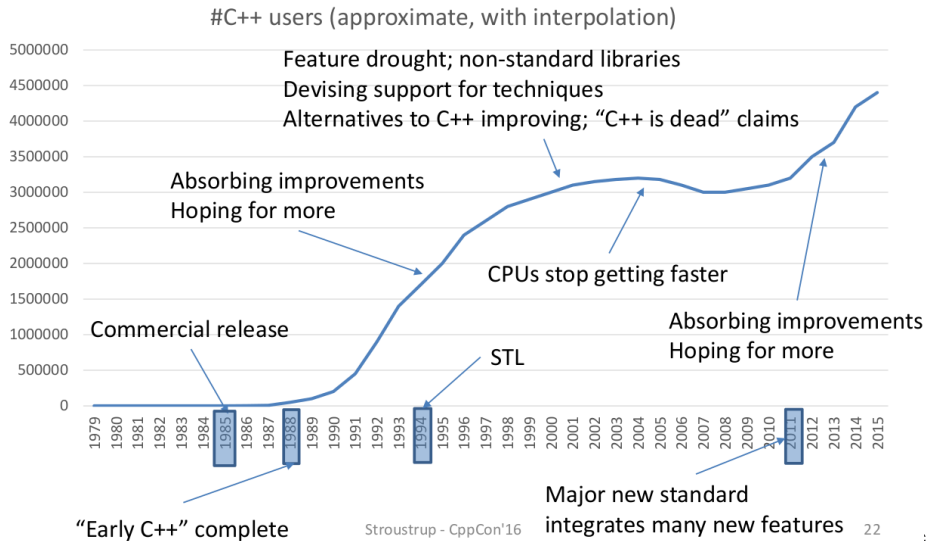
História do C++



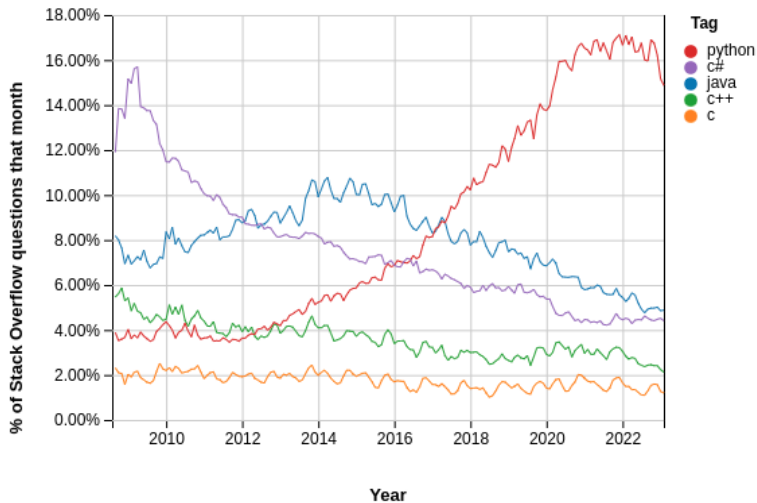
Bjarne Stroustrup, *The Evolution of C++: Past, Present, and Future*, CppCon 2016

https://youtu.be/_wzc7a3McOs

História do C++

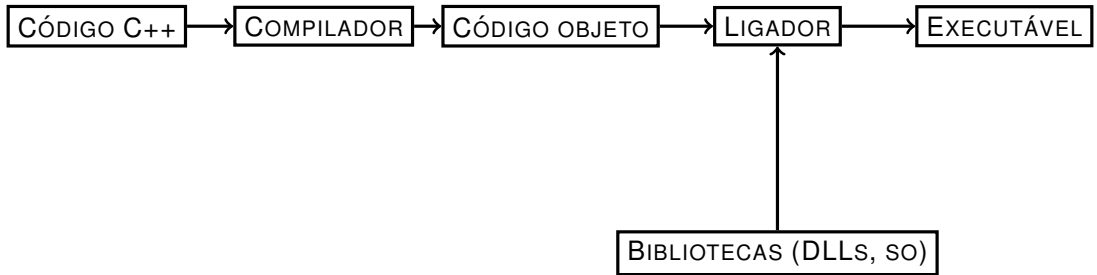


Stack Overflow



Fonte: <https://insights.stackoverflow.com/>

Compilação de programas



- **iostream** para entrada e saída básica.
- **std::string** é uma estrutura (“classe”) C++ para string.
- **std::cout** é a saída padrão.
- **std::endl** é uma nova linha
 - UNIX/Linux - `\n`
 - Windows - `\r\n`

ola.cpp

```
#include <iostream>

int main(void)
{
    std::string mensagem{"Ola mundo!"}
    // isso eh um comentario
    std::cout << "Saida: " << mensagem <<
        std::endl;
    return 0;
}
```

Instalação do GCC em sistemas Debian/Ubuntu

```
$ apt install gcc g++
```

Instalar ferramentas essenciais

```
$ apt install make valgrind gdb cppcheck
```

Dica

Semelhante ao C, pode-se usar o **GCC** (g++) e **Clang** (clang++). Clang mostra erros de compilação de forma mais amigável, além de ser mais eficiente que o GCC.

Linha de comando

```
$ g++ -std=c++11 -O2 -Wall -g -o ola ola.cpp  
$ ./ola
```

Linha de comando (alternativa)

```
$ g++ -std=c++11 -O2 -Wall -g ola.cpp  
$ ./a.out
```

params.cpp

```
#include <iostream>

// argc eh o numero de parametros passados
// argv eh um vetor de strings com os valores
int main(int argc, char **argv)
{
    for(auto i= 0; i < argc; i++){
        std::cout << "Param " << i
            << " valor -> " << argv[i] << std::endl;
    }
    return 0;
}
```

Outline

- 1 Introdução
- 2 Variáveis
- 3 Containers STL
- 4 Entrada e saída
- 5 Escopos
- 6 Tratamento de erros

Pode-se usar **auto** quando o tipo é deduzido pelo compilador.

Exemplo

```
auto x = 1;      // inteiro
auto y = 2.0;    // double
auto teste = true; // booleano

// i abaixo eh um inteiro
for(auto i= 0; i < 10; i++)
    std::cout << "Valor: " << i << std::endl;
```

Inicialização padrão

É uma forma de padronizar a inicialização de variáveis em C++ usando `{ }`.

Exemplos

```
double x {1.0};           // declara um double
int a[] {1, 2, 3, 4};     // vetor com 4 elementos sem =
int b[] = {1, 2, 3, 4};   // mesma coisa
std::string nome {"Meu nome"} ; // uma string
```

Atenção

Não funciona com **auto**.

```
auto x {1.0}; // double ou float ?
```

Casting (conversão)

C++ apresenta quatro tipos de conversão:

- **static_cast** - tipos relacionados: `int` para `char`, ou `double*` para `int*`.
- **reinterpret_cast** - tipos não relacionados (inteiro para ponteiro, etc).
- **const_cast** - `const` ou `volatile`.
- **dynamic_cast** (*não usado aqui*).

Exemplo

```
int num = 97;                // inteiro
char letra = static_cast<char>(num); // agora letra A

char *dados = new char[100]; // 100 chars alocados
int* vetor = reinterpret_cast<int*>(dados); // mudei agora para int
```

Passagem por referência

Passagem por referência possibilita passar variáveis por **referência** (&) ao invés de valor ou ponteiro.

Exemplo

```
void f(int val, int& ref)
{
    val++; // incrementa a copia local de val
    ref++; // incrementa realmente a variavel
}
```

Importante

Evite usar passagem por referência porque deixa o programa mais difícil de entender. Use apenas quando queremos evitar uma cópia e não vamos alterar a variável (`const`), como por exemplo um vetor ou uma string:

```
void imprimir(const std::string& texto)
{
    std::cout << texto << std::endl; // nao altera variavel
}
```

struct Ponto

```
struct Ponto {  
    float x; // variaveis  
    float y;  
  
    // zera o ponto  
    void zera(void) {  
        x = 0.0f;  
        y = 0.0f;  
    }  
    // distancia deste ponto (x, y) ate p1  
    float distancia(Ponto& p1) const {  
        return std::sqrt( std::pow((x-p1.x), 2) + std::pow((y-p1.y), 2) );  
    }  
};
```

struct Ponto

```
int main(void)
{
    Ponto p1 {1.0, 1.0};
    Ponto p2;
    p2.zera();
    p2.x = 19.0;
    p2.y = 20.0;

    auto distancia = p1.distancia(p2);
    std::cout << "Distancia: " << distancia << std::endl;
}
```

- 1 Introdução
- 2 Variáveis
- 3 Containers STL**
- 4 Entrada e saída
- 5 Escopos
- 6 Tratamento de erros

A C++ STL (*Standard Template Library*) consiste em iteradores, containers (ou TADs), algoritmos, e funções parte da biblioteca padrão do C++.

Vetores C++

```
std::vector<int> v1 = {1, 2, 3, 4};  
std::vector<char> v2;
```

```
// insere no fim  
v1.push_back(5);
```

```
// insere no começo  
v1.push_front(0);
```

```
// imprime  
std::cout << v1[3] << std::endl;
```

```
// insere um caractere  
v2.push_back('a');
```

Iteradores são “ponteiros” ou “cursors” genéricos aos dados dentro de um container.

Os containers possuem os métodos:

- `begin()` para começo.
- `end()` para o fim (ou além dele).
- `last()` para o último elemento.
- `size()` número de elementos.
- `capacity()` capacidade total em memória.
- `clear()` “limpa” o container.

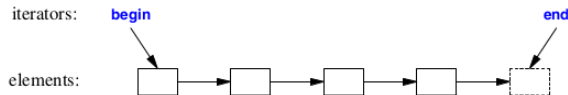
O iterador tem o operador `*` para acessar o elemento apontado.

Vetores C++

```
std::vector<int> v1 = {1, 2, 3, 4};
```

```
for(auto it= v1.begin(); it != v1.end();  
    it++)  
    std::cout << *it;
```

```
std::vector<int>::iterator it;  
for(it= v1.begin(); it != v1.end(); it  
    ++)  
    std::cout << *it;
```



Tem como alterar os valores?

Iteradores são “ponteiros” ou “cursors” genéricos aos dados dentro de um container. Os containers possuem os métodos:

- `begin()` para começo.
- `end()` para o fim (ou além dele).
- `last()` para o último elemento.
- `size()` número de elementos.
- `capacity()` capacidade total em memória.
- `clear()` “limpa” o container.

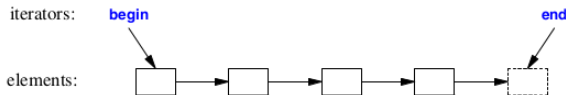
O iterador tem o operador `*` para acessar o elemento apontado.

Vetores C++

```
std::vector<int> v1 = {1, 2, 3, 4};
```

```
for(auto it= v1.begin(); it != v1.end();  
    it++)  
    std::cout << *it;
```

```
std::vector<int>::iterator it;  
for(it= v1.begin(); it != v1.end(); it  
    ++)  
    std::cout << *it;
```



Tem como alterar os valores?

Iteradores são “ponteiros” ou “cursos” genéricos aos dados dentro de um container. Os containers possuem os métodos:

- `begin()` para começo.
- `end()` para o fim (ou além dele).
- `last()` para o último elemento.
- `size()` número de elementos.
- `capacity()` capacidade total em memória.
- `clear()` “limpa” o container.

O iterador tem o operador `*` para acessar o elemento apontado.

Vetores C++

```
std::vector<int> v1 = {1, 2, 3, 4};
```

```
for(auto it= v1.begin(); it != v1.end();  
    it++)  
    *it = *it + 1;
```

Outline

- 1 Introdução
- 2 Variáveis
- 3 Containers STL
- 4 Entrada e saída**
- 5 Escopos
- 6 Tratamento de erros

Entrada e saída

As operações são efetuadas por **streaming** ou fluxo onde dados:

- **std::ifstream** para leitura (operador >>).
- **std::ofstream** para escrita (operador <<).

Exemplo

```
#include <iostream>
#include <fstream>

int main(void)
{
    int n1, n2;
    std::ifstream entrada {"entrada.txt"};
    std::ofstream saida {"saida.txt"};
    entrada >> n1 >> n2;
    saida << n1 << " " << n2 << std::endl;
    return 0;
}
```

EOF - *end-of-file*

```
#include <iostream>
#include <fstream>
int main(void) {
    int n;
    std::ifstream entrada {"numeros.txt"};
    std::ofstream saida {"saida.txt"};
    if(entrada.is_open() == false)
        throw std::runtime_error{"ERRO arquivo!"};

    while(entrada.eof() == false){
        entrada >> n;
        saida << n << std::endl;
    }
    entrada.close();
    saida.close();
    return 0;
}
```

Ler linha em C++ (continua)

```
#include <iostream>
#include <fstream>
#include <vector>

struct Aluno {
    int matricula;
    std::string nome;
};
```


Ler linha em C++

```
int main(void)
{
    int matricula;
    std::string nome;
    std::vector<Aluno> alunos;    // vetor de alunos
    std::ifstream entrada {"alunos.txt"};
    while( entrada >> matricula ) { // le matricula
        std::getline(entrada, nome); // le resto da linha
        alunos.push_back( Aluno{matricula, nome} );
    }

    for(Aluno& v: alunos)
        std::cout << v.matricula << " -> " << v.nome << std::endl;
    return 0;
}
```

Outline

- 1 Introdução
- 2 Variáveis
- 3 Containers STL
- 4 Entrada e saída
- 5 Escopos**
- 6 Tratamento de erros

Namespaces em C++ são **escopos nomeados** e aumenta a modularidade do código. A `std` é o namespace padrão do C++.

```
#include <iostream>

int main(void)
{
    std::string mensagem{"Ola mundo!"}
    std::cout << << mensagem << std::endl;
    return 0;
}
```

Namespaces

Pode-se criar novos escopos para bibliotecas ou versões de software.

```
namespace Uteis {  
    void foo(void) {  
        std::cout << "Funcao foo aqui" << std::endl;  
    }  
}  
  
// usando a funcao assim  
Uteis::foo();
```

Outline

- 1 Introdução
- 2 Variáveis
- 3 Containers STL
- 4 Entrada e saída
- 5 Escopos
- 6 Tratamento de erros**

- C++ permite o tratamento de erros em tempo de execução com **exceptions**.
- A palavra chave `throw` cria um erro.

```
#include <iostream>

int main(void)
{
    int n;
    std::cout << "Digite um numero: ";
    std::cin >> n;
    if(n < 0)
        throw std::runtime_error {"Digite apenas
                                     numeros positivos!"};
    std::cout << n << std::endl;
    return 0;
}
```

- Tratar exceções depende da estrutura `try/catch`
- O bloco `try` é o código protegido
- `catch` é executado somente quando ocorrer uma exceção.

```
#include <iostream>

int main(void)
{
    try {
        auto a = 33;
        auto b = 0;
        std::cout << "a/b =" << a/b << std::endl;
    } catch(std::exception& e) {
        std::cout << "ERRO: " << e.what();
        throw; // recria
    }
    return 0;
}
```

<https://joao-ufsm.github.io/l22023a/>

