

Linguagem C++ I

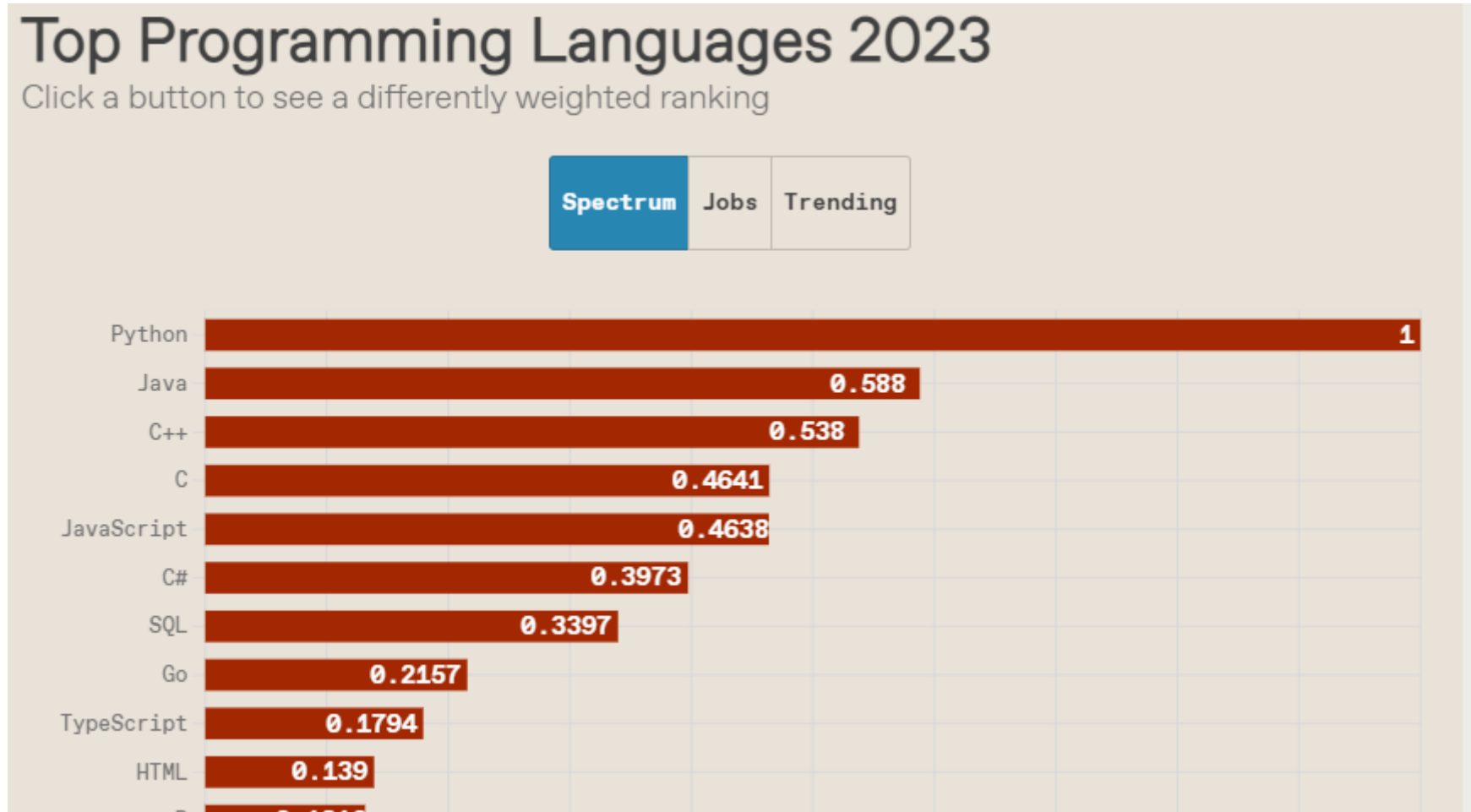
04/12/2023

Sumário

- C++ vs C
- Funções – Passagem de argumentos
- Funções – Argumentos por omissão
- Funções – Overloading
- Funções Genéricas – Template Functions
- Gestão da Memória
- Tratamento de Exceções
- Ligações úteis
- Referência

Motivação

IEEE Spectrum – Top Programming Languages



<https://spectrum.ieee.org/the-top-programming-languages-2023>

A linguagem C++

- Linguagem de programação **vasta** e **complexa**
- É uma **expansão** da linguagem **C**
- A maioria dos programas escritos em C são válidos em C++ **!!**

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

- Freestanding implementations
- ASCII chart
- Language**
 - Basic concepts
 - Keywords
 - Preprocessor
 - Expressions
 - Declarations
 - Initialization
 - Functions
 - Statements
 - Classes
 - Overloading
 - Templates
 - Exceptions
- Standard library (headers)**
- Named requirements**
- Feature test macros (C++20)**
- Language support library**
 - source_location (C++20)
 - Type support
 - Program utilities
 - Coroutine support (C++20)
 - Three-way comparison (C++20)
 - numeric_limits – type_info
 - initializer_list (C++11)
- Concepts library (C++20)**
- Diagnostics library**
 - exception – System error
 - basic_stacktrace (C++23)
- Memory management library**
 - unique_ptr (C++11)
 - shared_ptr (C++11)
 - Low level management

- Metaprogramming library (C++11)**
 - Type traits – ratio
 - integer_sequence (C++14)
- General utilities library**
 - Function objects – hash (C++11)
 - Swap – Type operations (C++11)
 - Integer comparison (C++20)
 - pair – tuple (C++11)
 - optional (C++17)
 - expected (C++23)
 - variant (C++17) – any (C++17)
 - String conversions (C++17)
 - Formatting (C++20)
 - bitset – Bit manipulation (C++20)
- Strings library**
 - basic_string – char_traits
 - basic_string_view (C++17)
 - Null-terminated strings:
 - byte – multibyte – wide
- Containers library**
 - array (C++11)
 - vector – deque
 - list – forward_list (C++11)
 - set – multiset
 - map – multimap
 - unordered_map (C++11)
 - unordered_multimap (C++11)
 - unordered_set (C++11)
 - unordered_multiset (C++11)
 - stack – queue – priority_queue
 - flat_set (C++23)
 - flat_multiset (C++23)
 - flat_map (C++23)
 - flat_multimap (C++23)
 - span (C++20) – mdspan (C++23)

- Iterators library**
- Ranges library (C++20)**
- Algorithms library**
 - Execution policies (C++17)
 - Constrained algorithms (C++20)
- Numerics library**
 - Common math functions
 - Mathematical special functions (C++17)
 - Mathematical constants (C++20)
 - Numeric algorithms
 - Pseudo-random number generation
 - Floating-point environment (C++11)
 - complex – valarray
- Date and time library**
 - Calendar (C++20) – Time zone (C++20)
- Localizations library**
 - locale – Character classification
- Input/output library**
 - Print functions (C++23)
 - Stream-based I/O – I/O manipulators
 - basic_istream – basic_ostream
 - Synchronized output (C++20)
- Filesystem library (C++17)**
 - path
- Regular expressions library (C++11)**
 - basic_regex – algorithms
- Concurrency support library (C++11)**
 - thread – jthread (C++20)
 - atomic – atomic_flag
 - atomic_ref (C++20)
 - memory_order – condition_variable
 - Mutual exclusion – Semaphores (C++20)
 - future – promise – async
 - latch (C++20) – barrier (C++20)

<https://en.cppreference.com/w>

C++ vs C

C++ vs C – O que é “idêntico” ?

- Valores, tipos, literais, expressões
 - O tipo **booleano** (**bool**) é um tipo pré-definido !!
- Variáveis
- Instruções condicionais: **if**, **switch**
- Ciclos: **while**, **for**, **do-while** e **iteradores**
- Call-return: por **valor**, por **ponteiro** e **por referência**

C++ vs C – O que é diferente ?

- C++ é uma **linguagem OO** !!
 - Classes, herança e polimorfismo
- C++ suporta **programação genérica: templates**
- C++ permite o **tratamento de exceções: throw – try – catch**
- C++ disponibiliza **bibliotecas** poderosas
 - **Strings** library + **Containers** library + **Algorithms** library + ...

hello.cpp

```
#include<iostream>
```



Para usar o **stream** **std::cout** e o **operator <<**

```
/* This is a  
   comment */
```

```
int main(void) {  
    // Another comment
```

```
name space -> std::cout << "Hello world!\n";
```

```
    return 0;
```

```
}
```

hello.cpp

```
#include<iostream>
```

```
using namespace std;
```



Para facilitar a escrita do código

```
int main(void) {  
    // A comment  
    cout << "Hello world!" << endl;  
    return 0;  
}
```

Compilação e execução

- Linux

```
c++ source_file.cpp -> ./a.out
```

```
c++ -Wall -Wextra source_file.cpp
```

```
c++ -Wall -Wextra my_file.cpp -o exec_name -> ./exec_name
```

- Windows

```
g++ source_file.cpp -> .\a.exe
```

```
g++ -Wall -Wextra source_file.cpp
```

```
g++ -Wall -Wextra my_file.cpp -o exec_name -> .\exec_name
```

Input-Output

```
#include <iostream>

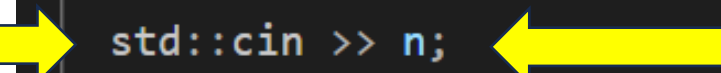
✓ int main(void)
{
    int n = 0;

    std::cout << "Enter an integer value? ";

    std::cin >> n;


    ✓ for( int i = 1; i <= n; ++i )
    {
        std::cout << i << std::endl;
    }

    return 0;
}
```



A diagram illustrating the flow of data in the provided C++ code. A yellow arrow points from the left towards the line `std::cin >> n;`, representing input from the user. Another yellow arrow points from the right towards the same line, representing the output of the program.


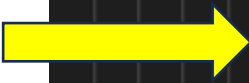

Formatação do Output



```
#include <math.h>
#include <iostream>
#include <iomanip>

using namespace std; // make all symbols of the std namespace directly visible

void do_it(int N)
{
    cout << " n n*n      sqrt(n)\n"
    << "-- --- -----\n";
    for(int i = 1; i <= N; i++)
        cout << setw(2) << i
        << " "
        << setw(3) << i * i
        << " "
        << fixed << setw(17) << setprecision(15) << sqrt((double)i)
        << endl;
}
```



Name Spaces

Name Spaces

- A **visibilidade** de variáveis e de funções pode ser controlada definindo-os em diferentes **name spaces**

namespace NEW

```
{  
    static int t_bytes;  
    int f(int x) { return 2 * x; }  
}
```

namespace OLD

```
{  
    static int t_bytes;  
    int f(int x) { return 3 * x; }  
}
```

- O acesso a elementos de um name space é feito usando **NEW::t_bytes** e **OLD::f**
- Ou atribuindo-lhes visibilidade: **using namespace OLD;**

Funções



– Passagem de Argumentos

Funções – Passagem de Argumentos

- Tal como em C, os **argumentos** de uma função podem ser passados **por valor** ou **por ponteiro**
- Também podem ser passados **por referência**, **sem usar** explicitamente **ponteiros**

```
// C++; called as follows: swap(var1, var2);  
void swap(int &x, int &y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```



Call-by-Value



```
double ChangeItByValue(double it) {  
    it += 10.0;  
    std::cout << "Within function, it = " << it << std::endl;  
    return it;  
}
```


```
std::cout << "Call-by-value" << std::endl;  
std::cout << "Before function execution, it = " << it << std::endl;  
double result = ChangeItByValue(it);  
std::cout << "After function execution, it = " << it  
    << "\nResult returned is " << result << std::endl;
```

Call-by-Pointer




```
double ChangeItByPointer(double* p) {  
    *p += 10.0;  
    std::cout << "Within function, *p = " << *p << std::endl;  
    return *p;  
}
```

```
std::cout << "Call-by-pointer" << std::endl;  
std::cout << "Before function execution, it = " << it << std::endl;  
result = ChangeItByPointer(&it);  
std::cout << "After function execution, it = " << it  
    << "\nResult returned is " << result << std::endl;
```



Call-by-Reference



```
double ChangeItByReference(double& it) {  
    it += 10.0;  
    std::cout << "Within function, it = " << it << std::endl;  
    return it;  
}
```

```
std::cout << "Call-by-reference" << std::endl;  
std::cout << "Before function execution, it = " << it << std::endl;  
result = ChangeItByReference(it);  
std::cout << "After function execution, it = " << it  
    << "\nResult returned is " << result << std::endl;
```

Funções

– Argumentos por Omissão

Funções – Argumentos por Omissão

- Os **últimos argumentos** de uma função podem ter valores atribuídos por omissão (**default values**)
- A **inicialização** desses argumentos por omissão é habitualmente feita no **protótipo da função**, caso exista

// function prototype (usually placed in a header file)

```
int f(int x, int y = 2, int z = 3);
```

```
int f(int x,int y,int z) { // actual definition of the function  
    return x + 2 * y + 3 * z;  
}
```

Funções

– Overloading

Funções – Overloading

- Funções com o mesmo nome e diferentes listas de argumentos podem coexistir e ser invocadas de acordo com os tipos dos seus argumentos
- Mas não é permitido que duas funções se distingam apenas pelo tipo do seu resultado

```
int    square(int    x) { return x * x; }  
double square(double x) { return x * x; }
```

Function Overloading

```
void show(const int i)
{
    cout << "int: " << i << endl;
}

void show(const double d)
{
    cout << std::fixed << "double: " << d << endl;
}
```

Function Overloading

```
void show(const char *s)
{
    cout << "string: " << s << endl;
}
```

```
void show(const char *s,const char *h)
{
    cout << h << s << endl;
}
```

```
void show(const int *a,const int n = 3)
{
    cout << "array: [";
    for(int i = 0;i < n;i++)
    {
        if(i != 0)
            cout << ',';
        cout << a[i];
    }
    cout << "]" << std::endl;
}
```

Function Overloading

```
✓ int main(void)
{
    show(1.0);
    show("hello");
    show(-3);
    show("John", "name: ");

    int a[3] = { 1, 2, -3 };

    show(a);

    return 0;
}
```

Function Overloading

```
int plus(int a, int b) { return a + b; }

double plus(double x, double y) { return x + y; }

// Concatenating strings
std::string plus(std::string s1, std::string s2) { return s1 + s2; }
```

```
int n = plus(3, 4);
std::cout << "plus(3, 4) returns " << n << std::endl;

double d = plus(3.2, 4.2);
std::cout << "plus(3.2, 4.2) returns " << d << std::endl;

std::string s = plus("he", "llo");
std::cout << "plus(\"he\", \"llo\") returns " << s << std::endl;
```

Function Overloading

```
int compare(const int &v1, const int &v2) {  
    if (v1 < v2) return -1;  
    if (v2 < v1) return 1;  
    return 0;  
}  
  
int compare(const double &v1, const double &v2) {  
    if (v1 < v2) return -1;  
    if (v2 < v1) return 1;  
    return 0;  
}
```

Function Overloading

```
int compare(const char &v1, const char &v2) {  
    if (v1 < v2) return -1;  
    if (v2 < v1) return 1;  
    return 0;  
}  
  
int compare(const std::string &v1, const std::string &v2) {  
    if (v1 < v2) return -1;  
    if (v2 < v1) return 1;  
    return 0;  
}
```

Funções Genéricas

– Template Functions

Funções Genéricas – Template Functions

- Uma **função genérica** é definida sem que sejam especificados os tipos de todos os seus argumentos ou do seu resultado
- Ficando definida uma **família de funções**
- Tal permite a sua **invocação para diferentes tipos de dados**

```
template <typename T> T f(T x) {  
    return T(7) * x;    // multiply x by 7  
                        // 7 is cast to type T (must be possible)  
}
```

Funções Genéricas – Template Functions

- A invocação de uma função genérica pode ou não concretizar explicitamente os tipos genéricos associados


<code>int i = f<int>(3);</code>	<code>// i = 7 * 3</code>
<code>double d = f<double>(5.0)</code>	<code>// d = 7.0 * 5.0</code>
<code>int n = f(1);</code>	<code>// n = 7 * 1</code>
<code>double x = f(10.0);</code>	<code>// x = 7.0 * 10.0</code>

Funções Genéricas – Template Functions


```
template <typename T>  
✓ T plus(T a, T b) {  
    return a + b;  
}
```

```
template <typename T>  
int compare(const T &v1, const T &v2) {  
    if (v1 < v2) return -1;  
    if (v2 < v1) return 1;  
    return 0;  
}
```

Funções Genéricas – Template Functions



```
template <typename T>
T sum(const T *a,int n)
{
    T s = T(0);
    for(int i = 0;i < n;i++)
        s += a[i];
    return s;
}
```



```
template <typename T>
double mean(const T *a,int n)
{
    T s = T(0);
    for(int i = 0;i < n;i++)
        s += a[i];

    return double(s) / double(n);
}
```

Funções Genéricas – Template Functions

```
#define size(x) (int)(sizeof(x) / sizeof(x[0]))
```

```
int ia[] = { 1,2,3,4,5 };  
double da[] = { 1.0,3.0,5.0 };  
  
cout << "ia[] sum: "  
    << sum<int>(ia,size(ia))  
    << "\nda[] sum: "  
    << sum<double>(da,size(da))  
    << endl;  
  
cout << "ia[] mean: "  
    << mean<int>(ia,size(ia))  
    << "\nda[] mean: "  
    << mean<double>(da,size(da))  
    << endl;
```

auto

– Dedução automática de tipo

auto – Dedução automática de tipo

- O **tipo da variável** declarada é automaticamente **deduzido** a partir do seu **valor inicial**
- O **tipo do resultado de uma função** é automaticamente **deduzido** a partir das suas **instruções de return**

auto – Dedução automática de tipo

```
template<typename T, typename U>
```

```
auto add(T t, U u) { return t + u; }
```

```
// the return type is the type of operator+(T, U)
```

```
auto a = 1 + 2;           // type of a is int
```

```
auto b = add(1, 1.2);     // type of b is double
```


Gestão da Memória

Gestão da Memória

- A **alocação** de memória é feita usando o **operador new**
- A **libertação** de memória é feita usando o **operador delete**
- No caso de arrays é usado **operador delete[]**

```
int *p_i = new int;    // get memory for an integer
*p_i = 3;              // give it the value 3
delete p_i;            // free its memory
p_i = new int(10);     // another integer initialized with the value 10
double *p_d = new double[100]; // an array of 100 doubles
delete[] p_d;          // free its memory
```

Tratamento de Exceções

Tratamento de Exceções

- Um modo habitual de lidar com **ocorrências excecionais** é **terminar a execução** do programa
- Em **aplicações críticas**, tal não é desejável, sendo necessário **gerir a ocorrência** sem terminar a execução
- Para tal, o código que se pretende “proteger” é colocado num bloco **try {...}** e o código de gestão de cada tipo de ocorrência é colocado em um ou mais blocos **catch(...) {...}**
- As **ocorrências** (excecionais) são assinaladas **lançando uma exceção**, usando a instrução **throw**

Tratamento de exceções

```
double sqrt(double x) {  
    if(x < 0.0) throw 0;    // throw an integer exception with the value 0  
    return sqrt(x);  
}  
...  
try {  
    cout << sqrt(-1.0) << endl;  
}  
catch(int i) {  
    cout << "integer exception number " << i << " caught" << endl;  
    exit(1);  
}
```

Tratamento de Exceções

```
const double special_value = 1.0; // CHANGED - J. Madeira

double my_sqrt(double x)
{
    if(x == special_value)
        throw 3;    // int exception (with value 3)
    if(x < 0.0)
        throw x;    // double exception (with value x, which is a negative number)
    return sqrt(x);
}
```

Tratamento de Exceções

```
try
{
    for(double x = 5; x >= -5.0; x -= 1.0)
        cout << x << " " << my_sqrt(x) << endl;
}
catch(int i)
{
    cout << "sqrt of the special_value [" << i << "]" << endl;
    exit(1);
}
catch(double d)
{
    cout << "sqrt of a negative number [" << d << "]" << endl;
    exit(1);
}
```

Ligações úteis

Ligações úteis

- [C++ reference at cppreference.com](http://cppreference.com)
- [C++ tutorial at tutorialspoint.com](http://tutorialspoint.com)
- [C++ Tutorial at w3schools.com](http://w3schools.com)
- [C++ coding tutor at pythontutor.com](http://pythontutor.com)

Referência

Referência

Tomás Oliveira e Silva, *AED Lecture Notes*, 2022