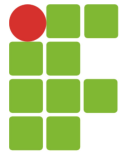
C++ para Programação Competitiva - Parte 3

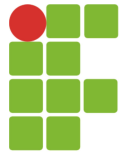
Introdução à Programação Competitiva

Prof. Daniel Saad Nogueira Nunes

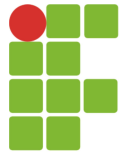
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

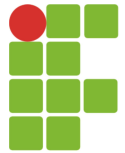


Sumário



A palavra-chave auto

- A partir do C++11 é possível utilizar a palavra-chave `auto`, em vez de explicitamente colocar o tipo de uma variável ou objeto, que o compilador inferirá o tipo.
- Ex: `auto x = f(a,b);`
- O tipo de `x` é **automaticamente** inferido pelo compilador.
- Deve sempre ser seguido de uma atribuição, ou o compilador não conseguirá inferir o tipo.

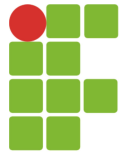


A palavra-chave auto

- Isso simplifica bastante a programação em C++.

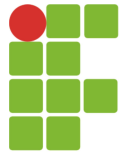
```
1 std::shared_ptr<::pplx::default_scheduler_t>
  ↪  s_ambientScheduler =
2     std::make_shared<::pplx::default_scheduler_t>();
```

```
1 auto s_ambientScheduler =
  ↪  std::make_shared<::pplx::default_scheduler_t>();
```



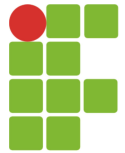
A palavra-chave auto

- Os exemplos a seguir utilizam **iteradores** para percorrer um mintinlinecppvetor e imprimir os valores.
- Iteradores são objetos que iteram sobre uma coleção de itens.
- Os dois códigos farão a mesma coisa, mas um deles utiliza a declaração com `auto`.



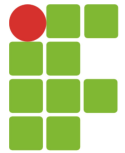
A palavra-chave auto

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6      vector<int> v = {1, 2, 3, 4, 5};
7      for (vector<int>::iterator it = v.begin(); it !=
   ↪   v.end(); ++it) {
8          cout << *it << endl;
9      }
10 }
```



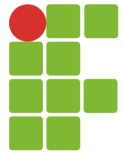
A palavra-chave auto

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6      vector<int> v = {1, 2, 3, 4, 5};
7      for (auto it = v.begin(); it != v.end(); ++it) {
8          cout << *it << endl;
9      }
10 }
```



A palavra-chave auto

- Nada impede que seja utilizado `const auto` ou `auto&`.
- O primeiro indica que é uma variável constante cujo tipo deve ser inferido pelo compilador.
- O segundo, que é uma referência cujo tipo deve ser inferido pelo compilador.
- Uma combinação dos dois também pode ser utilizada: `const auto&`



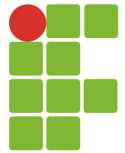
Range based loops

- A partir do C++11 é possível iterar sobre uma coleção de itens, como um `vector`, utilizando os chamados *range based loops*.
- Sintaxe:

```
1 for(auto x: v){
2     //...
3 }
```

.

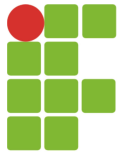
- A cada iteração, `x` recebe o próximo valor da coleção `v`.



Range based loops

- Caso a intenção seja alterar a coleção, podemos usar uma referência da seguinte forma:

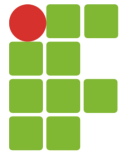
```
1 for(auto& x: v){
2     //...
3 }
```



Range based loops

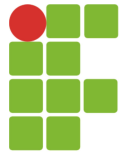
- Se queremos evitar uma cópia desnecessária de um objeto grande enquanto mantemos ele inalterado, podemos utilizar `const auto&`.

```
1 for(const auto& x: v){
2     //...
3 }
```

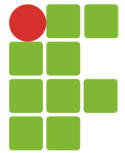


Exemplo

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main() {
7      vector<int> v(100);
8      for (auto &x : v)
9          cin >> x;
10     for (auto x : v)
11         cout << x << endl return 0;
12     return 0;
13 }
```

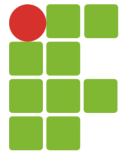


Sumário



Funções Lambda

- Funções lambda são funções anônimas que permitem que você defina funções exatamente onde elas são necessárias no código.
- Evita ter que criar uma função, identificada por um nome, em outra parte do arquivo fonte.
- São mecanismos que possibilitam passagem de funções para funções de alta ordem, isto é, funções cujos argumentos são outras funções.

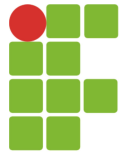


Funções lambda

- A sintaxe de funções lambda é definida da seguinte forma:
  `[captura](parametros) -> tipo_de_retorno { corpo };`
- O tipo pode ser omitido se é `void` ou pode ser deduzido do corpo pelo compilador.
- Exemplo:

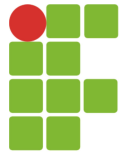
```
[](int &x) -> void { x++ };
```

é uma função lambda que não captura nenhum contexto, recebe uma referência para um inteiro e incrementa este inteiro.



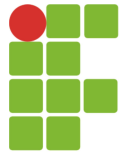
Funções lambda

- O exemplo a seguir incrementa todos os elementos de um vetor através do comando `for_each`, que aceita uma função como argumento.
- A função lambda envolvida não captura nada, mas recebe uma referência de um inteiro, que é incrementado.



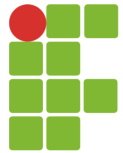
Exemplo

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  int main() {
8      vector<int> v(10);
9      for (auto &x : v)
10         cin >> x;
11     for_each(v.begin(), v.end(), [](int &x) -> void { x++; });
12     for (auto x : v) {
13         cout << x << endl;
14     }
15     return 0;
16 }
```



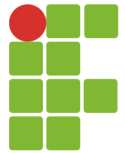
Funções lambda

- Se quiséssemos que todo elemento aumentasse de acordo com uma variável, poderíamos capturá-la na função lambda.
- Neste sentido, funções lambda são mais poderosas que funções convencionais, podem podem capturar um contexo.



Exemplo

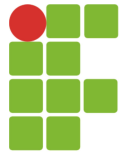
```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  int main() {
8      vector<int> v(10);
9      for (auto &x : v)
10         cin >> x;
11     int incremento = 10;
12     for_each(v.begin(), v.end(),
13              [incremento](int &x) -> void { x += incremento; });
14     for (auto x : v) {
15         cout << x << endl;
16     }
17     return 0;
18 }
```



Funções lambda

- A lista de variáveis capturadas é separada por vírgulas.

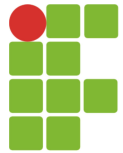
```
1  [x, &y, z](int a, int b) -> int {
2      y++;
3      return a - b + x * y / z;
4  }
```



Funções lambda

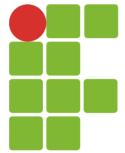
É possível, por exemplo:

- Capturar todas as variáveis ou nenhuma.
- Capturar todas as variáveis por valor ou por referência.
- Capturar apenas uma por valor e as demais por referência.
- Capturar apenas uma por referência e as demais por valor.

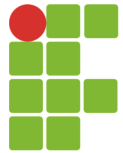


Funções lambda

```
1  []        // No captures, the lambda is implicitly convertible to a function pointer.
2  [x, &y]   // x is captured by value and y is captured by reference.
3  [&]       // Any external variable is implicitly captured by reference if used
4  [=]       // Any external variable is implicitly captured by value if used.
5  [&, x]    // x is captured by value. Other variables will be captured by reference.
6  [=, &z]   // z is captured by reference. Other variables will be captured by value.
```

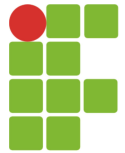


Sumário

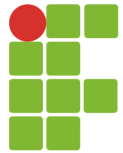


Tuplas

- O C++ possibilita o uso de pares e tuplas.
- Agregam múltiplos valores sob um único identificador.

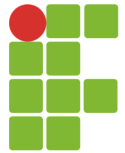


Sumário



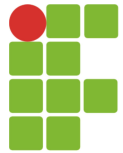
Pares

- Pares possibilitam agregar dois objetos em uma única estrutura.
- Não é necessário que os dois objetos sejam do mesmo tipo.
- Exemplo `pair<int,double> par;`



Pares: inicialização

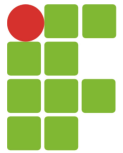
- `pair<int,double> par = {1,-2.5};`
- `pair<int,double> par = make_pair(1,-2.5);`
- `pair<int,double> par = outro_par;`



Pares: acesso

- Para acessar o primeiro membro do pair, utilizamos o campo `first`.
- O segundo membro do par pode ser obtido a partir do campo `second`.

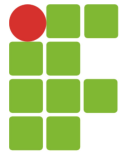
```
1 pair<int, double> p = {1, 2.5};
2 p.first;  // 1
3 p.second; // 2.5
```



Pares: acesso

- Também é possível acessar os membros de um par atráves da função `get`.

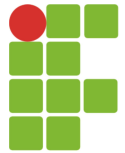
```
1 pair<int, double> p = {1, 2.5};
2 get<0>(p); // 1
3 get<1>(p); // 2.5
```



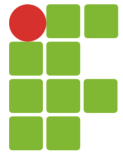
Pares: binding

- A partir do C++17 é possível atribuir os membros de um par diretamente a duas variáveis através do *structured binding*.

```
1 pair<int, double> p = {1, 2.5};
2 auto [i, d] = p; // i == 1 e d == 2.5
```

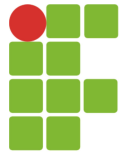


Sumário



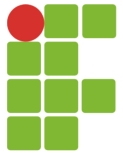
Tuplas

- Tuplas generalizam a noção de pares, suportando 2 ou mais membros.
- Exemplo: `tuple<int,double,char> tripla;`



Tuplas: inicialização

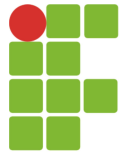
- ```
  tuple<int,double,char> tripla = {1,-2.5,'c'};
  ```
- ```
  tuple<int,double,char> tripla = make_tuple(1,-2.5,'c');
  ```
- ```
  tuple<int,double,char> tripla = outra_tripla;
  ```
  ```



Tuplas: acesso

- Os membros de uma tupla são acessíveis através da função `get`.

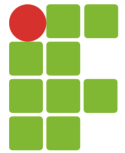
```
1 tuple<int, double, char> t = {1, 2.5,'c'};
2 get<0>(t); // 1
3 get<1>(t); // 2.5
4 get<2>(t); // 'c'
```



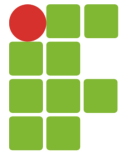
Tuplas: binding

- A partir do C++17 é possível atribuir os membros de uma tupla diretamente a variáveis através do *structured binding*.

```
1 tuple<int, double, char> p = {1, 2.5, 'c'};
2 auto [i, d, c] = p; // i == 1, d == 2.5 e c=='c'
```

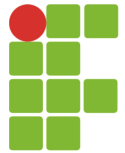


Sumário

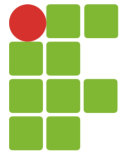


Algoritmos

- Através do cabeçalho `<algorithm>`, o C++ fornece uma série de ferramentas para desempenhar funções muito comuns na linguagem.
- Veremos algumas delas.

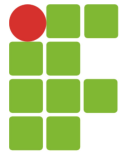


Sumário



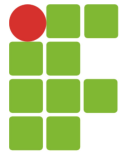
`for each`

- O comando `for_each` recebe dois iteradores para a coleção de objetos, o inicial e o final, além de receber como terceiro argumento uma função, que será aplicada a todos os objetos entre os iteradores inicial e final.



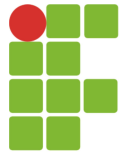
Exemplo

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  struct Sum {
6      void operator()(int n) { sum += n; }
7      int sum{0};
8  };
9
```



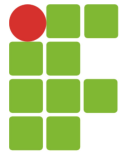
Exemplo

```
10  int main() {
11      std::vector<int> nums{3, 4, 2, 8, 15, 267};
12
13      auto print = [](const int &n) { std::cout << " " << n; };
14
15      std::cout << "before:";
16      std::for_each(nums.cbegin(), nums.cend(), print);
17      std::cout << '\n';
18
19      std::for_each(nums.begin(), nums.end(), [](int &n) { n++; });
20
21      // calls Sum::operator() for each number
22      Sum s = std::for_each(nums.begin(), nums.end(), Sum());
23
24      std::cout << "after: ";
25      std::for_each(nums.cbegin(), nums.cend(), print);
26      std::cout << '\n';
27      std::cout << "sum: " << s.sum << '\n';
28      return 0;
29  }
```

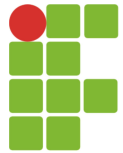


Exemplo

- O código anterior imprime todos os elementos da coleção através da função lambda `print`.
- Após isso, todos os elementos são incrementados através de um `for_each`.
- Através de um functor, que é um objeto que age como uma função por meio da sobrecarga do operador `()`, utiliza-se outro `for_each` para computar a soma da coleção de itens.

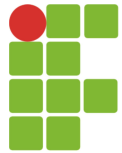


Sumário



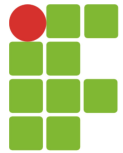
transform

- O `transform` recebe dois iteradores de início e fim da coleção a ser transformada, um iterador de início de onde os dados transformados devem ser escritos, e uma função, que deve ser aplicada sobre os dados da coleção original.
- O iterador de início da nova sequência pode ser o mesmo da coleção original, fazendo com que os dados sejam sobrescritos.



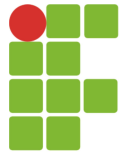
Exemplo

```
1  #include <algorithm>
2  #include <cctype>
3  #include <iomanip>
4  #include <iostream>
5  #include <string>
6  #include <vector>
7
8  int main() {
9      std::string s{"hello"};
10
11     std::transform(s.cbegin(), s.cend(),
12                    s.begin(), // write to the same location
13                    [](unsigned char c) { return std::toupper(c); });
14
15     std::vector<std::size_t> ordinals;
16     std::transform(s.cbegin(), s.cend(), std::back_inserter(ordinals),
17                    [](unsigned char c) { return c; });
18
19     std::cout << "ordinals: ";
```



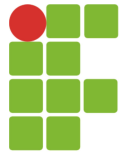
Exemplo

```
20      for (auto ord : ordinals) {
21          std::cout << ord << ' ';
22      }
23
24      std::transform(ordinals.cbegin(), ordinals.cend(), ordinals.cbegin(),
25                     ordinals.begin(), std::plus<>{});
26
27      std::cout << "\nordinals: ";
28      for (auto ord : ordinals) {
29          std::cout << ord << ' ';
30      }
31      std::cout << '\n';
32  }
```

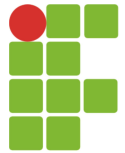


transform

- No exemplo anterior, primeiramente os caracteres são transformados na sua versão maiúscula.
- Após, utiliza-se `transform` para obter o valor numérico associado a cada caractere, os inserindo em outro vetor de inteiros.
- Finalmente, a função `transform` é aplicada mais uma vez para duplicar os valores do vetor de inteiros.

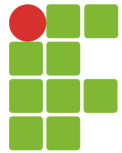


Sumário



all of

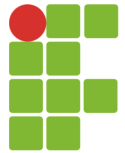
- O mecanismo `all of` recebe os iteradores de início e fim, e um predicado, e verifica se **todos** os elementos possuem a propriedade descrita pelo predicado.



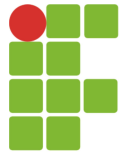
Exemplo

```cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <numeric>
#include <vector>

int main() {
    std::vector<int> v(10, 2);
    std::partial_sum(v.cbegin(), v.cend(), v.begin()); // v = {2,4,6,...}
    if (std::all_of(v.cbegin(), v.cend(), [](int i) { return i % 2 == 0; })) {
        std::cout << "All numbers are even\n";
    }
    return 0;
}
```

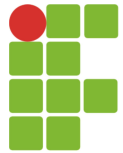


Sumário



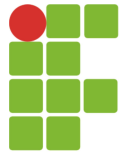
any of

- O mecanismo `any of` recebe os iteradores de início e fim, e um predicado, e verifica se **pelo menos um** os elementos possuem a propriedade descrita pelo predicado.

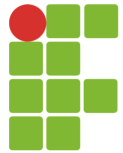


Exemplo

```
1  #include <algorithm>
2  #include <functional>
3  #include <iostream>
4  #include <iterator>
5  #include <numeric>
6  #include <vector>
7
8  struct DivisibleBy {
9      const int d;
10     DivisibleBy(int n) : d(n) {}
11     bool operator()(int n) const { return n % d == 0; }
12 };
13
14 int main() {
15     std::vector<int> v(10, 2);
16     std::partial_sum(v.cbegin(), v.cend(), v.begin()); // v = {2,4,6,...}
17
18     if (std::any_of(v.cbegin(), v.cend(), DivisibleBy(7))) {
19         std::cout << "At least one number is divisible by 7\n";
20     }
21     return 0;
22 }
```

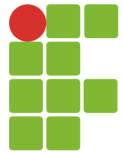


Sumário



count

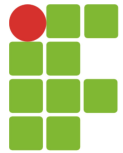
- O `count` conta o número de elementos entre os iteradores de início e fim que são iguais a um determinado valor.



count

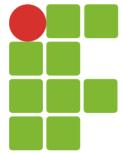
```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>

int main() {
    constexpr std::array v = {1, 2, 3, 4, 4, 3, 7, 8, 9, 10};
    for (const int target : {3, 4, 5}) {
        const int num_items = std::count(v.cbegin(), v.cend(), target);
        std::cout << "number: " << target << ", count: " << num_items << '\n';
    }
    return 0;
}
```



count if

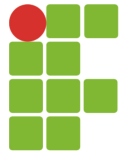
- O `count_if`, é similar ao `count`, mas em vez de receber um valor, recebe um predicado. Ao final, retorna o número de elementos que atendem o predicado.



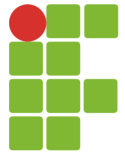
count if

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>

int main() {
    constexpr std::array v = {1, 2, 3, 4, 4, 3, 7, 8, 9, 10};
    int count_div4 =
        std::count_if(v.begin(), v.end(), [](int i) { return i % 4 == 0; });
    std::cout << "numbers divisible by four: " << count_div4 << '\n';
    return 0;
}
```

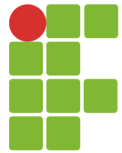


Sumário



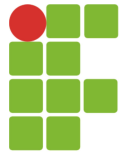
find

- O `find` recebe os iteradores de início e fim e retorna, através de um iterador, o elemento procurado, caso ele ocorra. Em caso negativo, retorna-se `end`.
- `find_if` e `find_if_not` funcionam de maneira similar, mas em vez de um valor, recebem um predicado (função) e retornam um iterador para o primeiro elemento com a propriedade, no caso de `find_if` ou o primeiro elemento que não possua a propriedade, no caso de `find_if_not`.

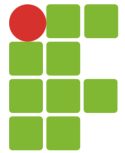


Exemplo

```
1  #include <algorithm>
2  #include <iostream>
3  #include <iterator>
4  #include <vector>
5
6  int main() {
7      std::vector<int> v{1, 2, 3, 4};
8      int n1 = 3;
9      int n2 = 5;
10     auto is_even = [](int i) { return i % 2 == 0; };
11
12     auto result1 = std::find(begin(v), end(v), n1);
13     auto result2 = std::find(begin(v), end(v), n2);
14     auto result3 = std::find_if(begin(v), end(v), is_even);
15
16     (result1 != std::end(v)) ? std::cout << "v contains " << n1 << '\n'
17                              : std::cout << "v does not contain " << n1 << '\n';
18
19     (result2 != std::end(v)) ? std::cout << "v contains " << n2 << '\n'
20                              : std::cout << "v does not contain " << n2 << '\n';
21
22     (result3 != std::end(v))
23         ? std::cout << "v contains an even number: " << *result3 << '\n'
24         : std::cout << "v does not contain even numbers\n";
25     return 0;
26 }
```

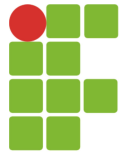


Sumário



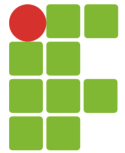
accumulate

- O accumulate recebe iteradores de início e fim, o valor inicial e realiza a soma (`operator+`) de todos os elementos a partir do valor inicial.
- Opcionalmente, pode-se fornecer como quarto parâmetro uma função binária que será aplicada nos termos na ordem em que estão na coleção.

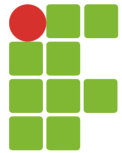


Exemplo

```
1  #include <functional>
2  #include <iostream>
3  #include <numeric>
4  #include <string>
5  #include <vector>
6
7  int main() {
8      std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9      int sum = std::accumulate(v.begin(), v.end(), 0);
10     int product =
11         std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());
12     auto dash_fold = [](std::string a, int b) {
13         return std::move(a) + '-' + std::to_string(b);
14     };
15     std::string s =
16         std::accumulate(std::next(v.begin()), v.end(),
17                         std::to_string(v[0]), // start with first element
18                         dash_fold);
19     std::cout << "sum: " << sum << '\n'
20               << "product: " << product << '\n'
21               << "dash-separated string: " << s << '\n';
22     return 0;
23 }
```



Sumário



Referências

M. Chowdhury, *How I discovered the C++ algorithm library and learned not to reinvent the wheel* , https://www.freecodecamp.org/news/how-i-discovered-the-c-algorithm-library-and-learned-not-t

cppreference, *cppreference.com*, https://en.cppreference.com/.

Deb Haldar, *C++11 auto: how to use and avoid abuse*, https://www.acodersjourney.com/c-11-auto/.

Wikipedia, *Anonymous function*, https://en.wikipedia.org/wiki/Anonymous_function.