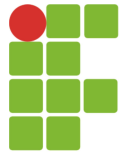
C++ para Programação Competitiva - Parte 2

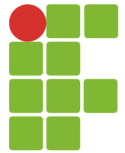
Introdução à Programação Competitiva

Prof. Daniel Saad Nogueira Nunes

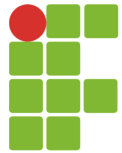
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

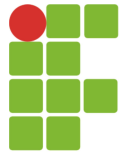


Sumário



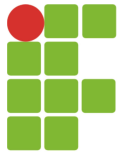
Classes

- Em uma visão simplista, classes são coleções extensíveis, pelo programador, de componentes, os quais podem ser: objetos, funções, variáveis de tipos primitivos, constantes, dentre outros.
- Objetos são instâncias de classe, mas nem toda classe precisa servir para instanciar objetos (exemplo, interfaces).
- Os componentes, também chamados de membros, podem ter as seguintes visibilidades:
  - `public`: podem ser acessíveis através do operador `.`
  - `private`: só são acessíveis através de funções membro da classe
  - `protected`: similar, ao `private`, mas podem ser acessados por classes `friend`. Existem algumas nuances em relação ao `proteced` e herança, mas não entraremos neste escopo.



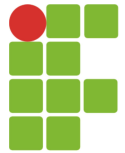
Classes

- Classes podem ter como membro funções, que podem ser invocadas através dos objetos instanciados.
- Essas funções membros também são chamadas de **métodos**.
- Variáveis e objetos membros da classe são denominadas **atributos**.



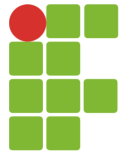
Classes

- A seguir ilustraremos um exemplo de uma classe `pessoa`.
- A classe `pessoa` define como atributos:
  - Idade: um inteiro;
  - Nome: uma string;
  - CPF: uma string.
- E define como métodos:
  - Leitura de um objeto do tipo pessoa.
  - Escrita de um objeto do tipo pessoa.



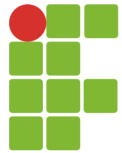
Classes

```
1  #include <iostream>
2
3  class pessoa {
4    public:
5      void le_pessoa() { std::cin >> nome >> idade >> cpf; }
6      void imprime_pessoa() {
7          std::cout << "Nome: " << nome << std::endl;
8          std::cout << "Idade: " << idade << std::endl;
9          std::cout << "CPF: " << cpf << std::endl;
10     }
11
12   private:
13     int idade;
14     std::string nome;
15     std::string cpf;
16 };
```



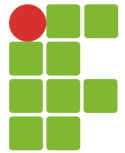
Classes

- Na nossa classe `pessoa`, os atributos `idade`, `nome` e `cpf` são privados.
- Só podem ser manipulados através dos métodos da classe.
- Como os métodos `le_pessoa` e `imprime_pessoa` possuem visibilidade pública, é possível invocá-los diretamente do objeto instanciado.
- Se os métodos fossem privados, só poderiam ser chamados a partir de outros métodos.



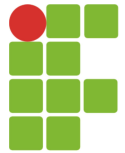
Classes

- Agora, podemos instanciar um objeto do tipo `pessoa`, ler e imprimir os dados de uma pessoa.
- O acesso aos membros é feito através do operador `.`

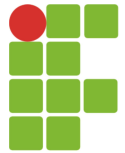


Classes

```
18 int main() {
19     pessoa p;
20     p.le_pessoa();
21     p.imprime_pessoa();
22     return 0;
23 }
```

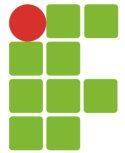


Sumário



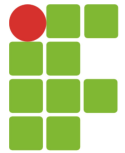
Construtores

- Um construtor é um método especial que tem o mesmo **nome** da classe é invocado quando o objeto é instanciado.
- Pode-se ter vários construtores com assinaturas diferentes.



Construtores

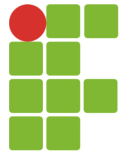
```
1  #include <iostream>
2
3  class pessoa {
4    public:
5      pessoa() {}
6      pessoa(std::string nome, int idade, std::string cpf) {
7          this->nome = nome;
8          this->idade = idade;
9          this->cpf = cpf;
10     }
11     void le_pessoa() { std::cin >> nome >> idade >> cpf; }
12     void imprime_pessoa() {
13         std::cout << "Nome: " << nome << std::endl;
14         std::cout << "Idade: " << idade << std::endl;
15         std::cout << "CPF: " << cpf << std::endl;
16     }
17
18   private:
19     int idade;
20     std::string nome;
21     std::string cpf;
22 };
```



Construtores

```
24 int main() {
25     pessoa p("Daniel", 34, "001.002.003-04");
26     p.imprime_pessoa();
27     return 0;
28 }
```

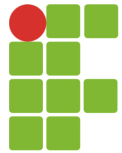
- Aqui utiliza-se o construtor que inicializa os membros com os valores passados.



Construtores

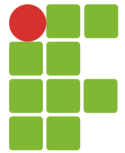
```
29
30  int main() {
31          pessoa p;
32          p.le_pessoa();
33          p.imprime_pessoa();
34          return 0;
35  }
```

- Aqui utiliza-se o construtor que não inicializa os atributos.



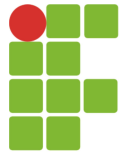
Construtores padrão

- O construtor que não inicializa nenhum atributo é conhecido como construtor padrão.
- A partir do C++11, é possível explicitar que trata-se de um construtor padrão através da palavra-chave `default`.



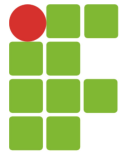
Construtor default

```
1  #include <iostream>
2
3  class pessoa {
4    public:
5      pessoa() = default;
6      pessoa(std::string nome, int idade, std::string cpf) {
7          this->nome = nome;
8          this->idade = idade;
9          this->cpf = cpf;
10     }
11     void le_pessoa() { std::cin >> nome >> idade >> cpf; }
12     void imprime_pessoa() {
13         std::cout << "Nome: " << nome << std::endl;
14         std::cout << "Idade: " << idade << std::endl;
15         std::cout << "CPF: " << cpf << std::endl;
16     }
17
18   private:
19     int idade;
20     std::string nome;
21     std::string cpf;
22 };
```

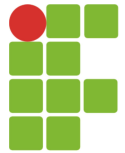


Construtor default

- Caso nenhum construtor seja declarado, o compilador, implicitamente, declara um construtor `default`.

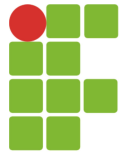


Sumário



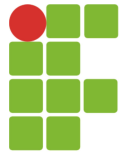
Destrutores

- Como o nome indica, um destrutor é um método que é chamado quando o objeto será destruído.
- Seu nome sempre é igual ao nome da classe precedido por um `~`.
- Ao contrário dos construtores, só é possível ter um destrutor, que não possui parâmetros.
- Pode ser utilizado para limpar a memória alocada, ou liberar recursos.

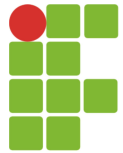


Destrutor

```
1  #include <cstring>
2
3  class my_string {
4    public:
5      my_string(char *s) {
6          size_t size = strlen(s);
7          str = new char[size + 1];
8          strcpy(str, s);
9      }
10     ~my_string() { delete[] str; }
11
12   private:
13     char *str;
14 };
```

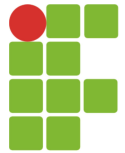


Sumário



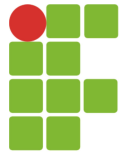
Sobrecarga de operadores

- Em C++ é possível escrever um método com o nome de um operador, como `+`, `*`, `()`, `-`, `<<`, `>>`, dentre outros.
- Assim, ao utilizar o operador, o método relacionado é chamado.
- Visa aumentar a capacidade de escrita enquanto torna o código legível em relação à semântica do operador.



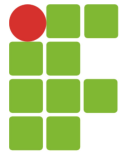
Sobrecarga de operadores

```
1  #include <iostream>
2
3  class Complex {
4    public:
5      Complex(int r = 0, int i = 0) {
6          real = r;
7          imag = i;
8      }
9
10     Complex operator+(Complex const &obj) {
11         Complex res;
12         res.real = real + obj.real;
13         res.imag = imag + obj.imag;
14         return res;
15     }
16     void print() { std::cout << real << " + i" << imag << '\n'; }
17
18   private:
19     int real, imag;
20 };
```

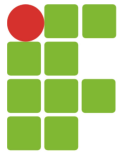


Sobrecarga de operadores

```
22 int main() {
23     Complex c1(10, 5), c2(2, 4);
24     Complex c3 = c1 + c2;
25     c3.print();
26 }
```

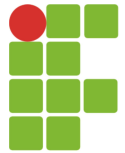


Sumário

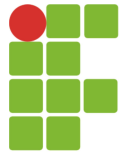


Structs

- Em C++, as `struct` também podem ter métodos, construtores, destrutores.
- A única diferença de `struct` para `class` é que a visibilidade **padrão** dos membros em uma `struct` é `public`, enquanto em uma `class` a visibilidade padrão é `private`.
- A visibilidade padrão é aplicada quando não especificada na `struct` ou `class`.

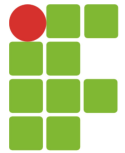


Sumário



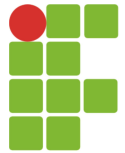
Vector

- Um `vector` em C++ é um vetor redimensionável.
- Ele é totalmente parametrizável, pode ser de qualquer tipo, inclusive de tipos criados pelo usuário.
- Os elementos podem ser acessados através do operador `[]`.



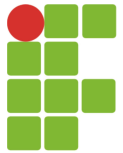
Vector

```
1 std::vector<int> v_int;
2 std::vector<double> v_double;
3 std::vector<bool> v_bool;
4 std::vector<string> v_string;
5 std::vector<pessoa> v_pessoa;
6 std::vector<vector<int>> vvi;
```



Vector

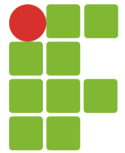
- Alguns métodos importantes são:
  - `size()` : retorna o número de elementos do vetor;
  - `resize(n)` : redimensiona o vetor para o novo tamanho n. Se n for maior que o tamanho anterior, o vetor cresce, senão, o vetor é encolhido.
  - `push_back(x)` : insere o elemento x ao final do vetor.
  - `pop_back()` : remove o elemento no final do vetor.
  - `clear()` : limpa o vetor.
  - `emplace_back(x)` : similar ao `push_back(x)` , mas é in-place.
  - `assign(n,value)` : atribui o valor `value` aos n elementos do vetor.



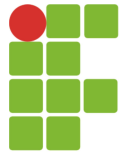
Vector

```cpp
#include <vector>

int main() {
    std::vector<int> v;
    v.assign(5, 0);     // {0,0,0,0,0}
    v.size();           // retorna 5
    v.push_back(1);     // {0,0,0,0,0,1}
    v.emplace_back(2); // {0,0,0,0,0,1,2} (in-place)
    v[6];               // retorna 2
    v.pop_back();       // remove o valor 1 ao final do vetor
    v.resize(3);        // diminui o tamanho do vetor para 3
    v.clear();          // limpa o vetor, o tamanho agora é 0
    return 0;
}
```

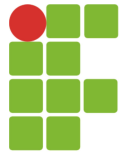


Sumário



Inicialização

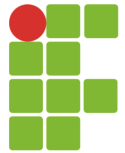
- Um `vector` pode ser inicializado de várias formas, sendo algumas delas:
  - Através de uma lista de inicialização, em que os valores são colocados entre chaves;
  - Por meio de um construtor default;
  - Utilizando um construtor indicando o número de elementos e o valor inicial de todos os elementos.



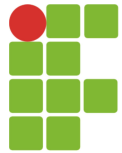
Vector

```cpp
#include <vector>

int main() {
    std::vector<int> v1 = {1, 2, 3, 4, 5}; // lista inicializadora
    std::vector<double> v2;                // construtor padrão
    std::vector<bool> v3(50, false);       // construtor assignment
    std::vector<std::vector<int>> matrix(
        100, vector<int>(100, 0)); // matriz 100 x 100
    return 0;
}
```

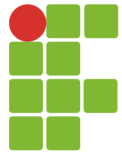


Sumário



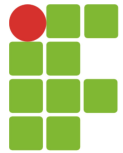
String

- O C++, diferentemente do C, possui um tipo `string`, que facilita a operação sobre palavras.
- Assim como o `vector`, um objeto `string` também pode ser redimensionado.



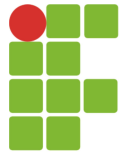
Vector

- Alguns métodos importantes são:
  - `size()` : retorna o número de elementos da string;
  - `resize(n)` : redimensiona a string para o tamanho n.
  - `push_back(x)` : insere o elemento x ao final da string.
  - `pop_back()` : remove o elemento do final da string
  - `clear()` : limpa a string.
  - `c_str()` : obtém a string `C` equivalente.
  - `+` : concatena duas strings;
  - `==` : compara duas strings.

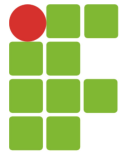


Strings

```
1  #include <string>
2
3  int main() {
4      std::string r;             // r == ""
5      std::string s = "abra"; // s == "abra"
6      s.push_back('c');          // s == "abrac"
7      s.push_back('a');          // s == "abraca"
8      s.pop_back();              // s == abrac
9      s.resize(3);               // s == abr
10     s[1];                      // retorna b
11     s = s + "acadabra";        // s == abracadabra;
12     s.clear();                 // s == ""
13     s == r;                    // true;
14     return 0;
15 }
```



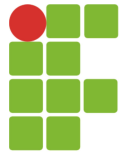
Sumário



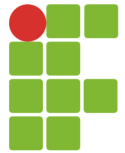
Leitura

- Strings podem ser lidas através do operador `>>` do stream `cin`. A leitura irá parar assim que encontrar um espaço em branco, tabulação ou fim de linha, como no `scanf`;
- Para ler uma linha inteira, pode-se usar o `getline`:

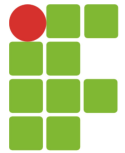
  `getline(cin,str);`. O `'\n '` não é inserido ao final de `str`.



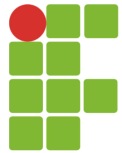
Sumário



Sumário



Sumário



Exercício

- Implemente uma classe `bitvector` que implemente um vetor de bits, com um número qualquer de bits.
- A classe deve sobrecarregar os operadores `|`,`&`,`^`,`~`, de modo a realizar a operação bit a bit entre dois `bitvector` ou, no caso do operador unário `~`, obtenha o complemento do `bitvector`. As operações binárias só deverão funcionar com dois `bitvector` do mesmo tamanho.
- Todos os operadores deverão retornar um novo `bitvector` com o novo resultado.
- Sua implementação deverá ser o mais espaço-eficiente possível.