Técnicas e Análise de Algoritmos

Manipulação de Programas e Introdução ao C++ - Parte 02

Professor: **Jeremias Moreira Gomes**
E-mail: jeremias.gomes@idp.edu.br

Introdução

Classes e Structs

Classes

- Em uma visão mais simplificada (porque vocês vão ver em detalhes em outra disciplina), classes são **coleções extensíveis de componentes**, os quais podem ser: objetos, funções, variáveis de tipos primitivos, constantes e outros
- **Objetos são instância de classe**
  - Mas nem toda classe precisa servir para instanciar objetos
  - Exemplos: interfaces e até structs (no caso do C++)

Classes

- Os componentes (membros), podem ter diferentes visibilidades:
  - **`public`**: podem ser acessíveis pelo operador ponto (.)
  - **`private`**: são acessíveis somente por funções membros da classe
  - **`protected`**: similar ao private, mas também podem ser acessados por classes friend
    - Tem mais detalhes, mas não é foco da disciplina

Classes

- Classes podem ter como membros funções ou variáveis
  - Diferença para structs em C
  - Funções membro de classes são chamadas de **métodos**
  - Variáveis membro de classes são chamadas de **atributos**

Classes - Exemplo

- Criação de uma classe **`pessoa`**
  - **Atributos**:
    - **`idade`**: inteiro
    - **`nome`**: string
    - **`cpf`**: string
  - **Métodos**:
    - **`imprime_pessoa`**: leitura de um objeto do tipo pessoa
    - **`le_pessoa`**: escrita de um objeto do tipo pessoa

Classes - Exemplo

```
class pessoa {
    private:
        int idade;
        std::string nome;
        std::string cpf;

    public:
        void imprime_pessoa() {
            std::cout << "Nome: " << nome << std::endl;
            std::cout << "Idade: " << idade << std::endl;
            std::cout << "CPF: " << cpf << std::endl;
        }
        void le_pessoa() {
            std::cin >> nome >> idade >> cpf;
        }
};
```

Classes - Exemplo

- Nessa implementação da classe pessoa, os atributos **`idade`**, **`nome`** e **`cpf`** são privados
  - Não podem ser acessados ou alterados fora da própria classe
- Os dois métodos (**`le_pessoa`** e **`imprime_pessoa`**) tem visibilidade pública, então é possível invocá-los a partir do uso da classe
  - Exemplo, instanciar um objeto do tipo **`pessoa`** na **`main`** e acessar membros públicos com o operador ponto (**`.`**)

Classes - Exemplo

```
int main()
{
    pessoa p1;
    p1.le_pessoa();
    p1.imprime_pessoa();
    return 0;
}
```

Classes - Construtores

- Construtor é um método especial que tem o mesmo nome da classe e é chamado (invocado) quando o objeto é instanciado

Classes - Construtores

- Construtor é um método especial que tem o mesmo nome da classe e é chamado (invocado) quando o objeto é instanciado

```
class pessoa {
    public:
        pessoa() {
            std::cout << "Construtor chamado: pessoa criada" << std::endl;
        }
};
int main()
{
    pessoa p1;
    return 0;
}
```

Classes - Construtores

- Construtor é um método especial que tem o mesmo nome da classe e é chamado (invocado) quando o objeto é instanciado

```
class pessoa {
    public:
        pessoa(int idade, string nome, string cpf) {
            this->idade = idade;
            this->nome = nome;
            this->cpf = cpf;
        }
    private:
        int idade;
        string nome, cpf;
};
```

```
int main()
{
    pessoa p1(20, "Davi", "123456789-00");
    return 0;
}
```

Classes - Construtores Padrão

- Se um construtor não inicializar nenhum atributo, ele é conhecido como construtor padrão
  - Pode-se explicitar (a partir do C++11) a palavra-chave **default**
  - Caso nenhum construtor seja declarado, o compilador, implicitamente, declara um construtor do tipo **default**

Classes - Construtores Padrão

```
using namespace std;
class ponto {
    public:
        ponto() = default;
        ponto(double x, double y) {
            this->x = x;
            this->y = y;
        }
    private:
        double x;
        double y;
};
```

```
int main()
{
    ponto p1;
    ponto p2(1.0, 2.0);

    return 0;
}
```

Classes - Sobrecarga de Operadores

- Em C++ é possível escrever um método com o nome de um operador
  - Exemplos: `+`, `*`, `()`, `-`, `<<`, `>>`
- Assim, ao utilizar o operador, o método refeito é chamado
- O objetivo da sobrecarga é aumentar a capacidade de escrita enquanto torna o código legível em relação a semântica do operador

Classes - Sobrecarga de Operadores

```
class ponto {
    public:
        ponto(double x, double y) {
            this->x = x;
            this->y = y;
        }
        ponto operator+(ponto p) {
            ponto novo_p(x + p.x, y + p.y);
            return novo_p;
        }
    private:
        double x;
        double y;
};
```

```
int main()
{
    ponto p1(5.0, 7.0);
    ponto p2(1.0, 2.0);
    ponto p3 = p1 + p2;

    return 0;
}
```

Classes - Sobrecarga de Operadores

- Algo que é muito útil na sobrecarga de operadores, é permitir a comparação entre dois objetos
    - A partir disso, a ordenação é possível

```
class ponto {
    public:
        ponto(double x, double y) {
            this->x = x;
            this->y = y;
        }
        int operator<(ponto p) {
            return x < p.x || (x == p.x && y < p.y);
        }
    private:
        double x, y;
};
```

Structs

- Em C++ **`structs`** também podem ter métodos, construtores e destrutores (não falei sobre esse, mas tem)
- A única diferença entre **`struct`** e classe é que a visibilidade padrão de uma struct é **`public`** e a visibilidade padrão de uma classe é **`private`**
  - Visibilidade padrão ocorre quando a visibilidade não é especificada

Structs

```
struct ponto {
    ponto(double x, double y) {
        this->x = x;
        this->y = y;
    }
    int operator<(ponto p) {
        return x < p.x || (x == p.x && y < p.y);
    }
    private:
        double x;
        double y;
};
```

Vector

Vector

- Um vector em C++ é um **vetor redimensionável**
- Pertence a biblioteca **`<vector>`**
- Ele é totalmente parametrizável
  - Pode ser de qualquer tipo
  - Pode ser de qualquer tipo criado pelo usuário
- Elementos são acessados por meio do operador **[]**

Vector

```cpp
#include <vector>

int main()
{
    std::vector<int> vetor;
    std::vector<double> vet_d;
    std::vector<bool> vet_b;
    std::vector<std::vector<int>> vet_vet;
    std::vector<ponto> vet_p;
}
```

Vector

- Alguns métodos mais utilizados
  - **`size()`**: retorna o número de elementos (`O(1)`)
  - **`resize(n)`**: redimensiona o vetor para o tamanho `n` (`O(n)`)
  - **`push_back(x)`**: insere o elemento `x` ao final do vetor (`O(1)` ???)
  - **`pop_back()`**; remove o elemento do final do vetor (`O(1)`)
  - **`clear()`**: limpa o vetor (`O(n)`)
  - **`emplace_back(x)`**: similar ao `push_back`, porém in-place (`O(1)`)
  - **`assign(n, x)`**: atribui o valor `x` a `n` elementos do vetor (`O(n)`)

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.pop_back();            // remove o valor 2 ao final do vetor
    v[5];                    // retorna 1
    v.assign(5, 7);          // {7,7,7,7,7}
    v.resize(10);            // redimensiona para: {7,7,7,7,7,0,0,0,0,0}
    v.clear();               // limpa o vetor, o tamanho agora é 0
    return 0;
}
```

Vector - Inicialização

- Um vector pode ser inicializado de várias formas:
  - Lista de inicialização (utilizando chaves)
  - Construtor `default`
  - Construtor com número de elementos e valor padrão

Vector - Inicialização

```
using namespace std;
int main()
{
    vector<int> v1 = {1, 2, 3, 4, 5};   // lista inicializada
    vector<double> v2;                  // construtor padrão
    vector<bool> v3(50, false);         // construtor com tamanho e valor padrão
    vector<vector<int>> matrix(100, vector<int>(100, 1)); // matriz 100 x 100

    return 0;
}
```

String

String

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

```
#include <bits/stdc++.h>

int main()
{
    std::string s;
}
```

String

- Alguns métodos mais utilizados
  - **`size()`**: retorna o número de elementos (`O(1)`)
  - **`push_back(c)`**: insere o char `c` ao final da string (`O(1)` ???)
  - **`pop_back()`**; remove o elemento do final da string (`O(1)`)
  - **`clear()`**: limpa a string (`O(n)`)
  - **`c_str()`**: obtém a string em C equivalente (`O(1)`)
  - **`==`**: compara duas strings (`O(n)`)
  - **`=`**: copia uma string (`O(n)`)
  - **`+`**: concatena duas strings (`O(n)`)

String

```
int main()
{
    std::string r;              // r == ""
    std::string s = "abra";     // s == "abra"
    s.push_back('c');           // s == "abrac"
    s.push_back('a');           // s == "abraca"
    s.pop_back();               // s == abrac
    s.resize(3);                // s == abr
    s[1];                       // retorna b
    s = s + "acadabra";         // s == abracadabra;
    s.clear();                  // s == ""
    s == r;                     // true;

    return 0;
}
```

String - 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
  - Similar ao `scanf`
- Para ler uma linha inteira, pode-se usar o getline:
  - **`getline(cin,str);`**
  - O **`'\n'`** não é inserido ao final de str

Range Based Loops

Iteradores

- Um **iterador** é um objeto que pode iterar em elementos em uma estrutura de dados complexa (container) e fornecer acesso a elementos individuais
- Os contêineres da biblioteca padrão do C++ fornecem iteradores de modo que todos os algoritmos possam acessar seus elementos de maneira padrão sem precisar se preocupar com o tipo do contêiner em que os elementos estão armazenados

Iteradores

```
int main()
{
    vector<int> v = {1, 2, 3, 4, 5};
    vector<int>::iterator it;

    for(it = v.begin(); it != v.end(); it++) {
        cout << *it << endl;
    }

    return 0;
}
```

Iteradores

```
int main()
{
    vector<vector<int>> v = {{1, 2, 3}, {4, 5, 6}};
    vector<vector<int>>::iterator it;
    vector<int>::iterator it2;
    for(it = v.begin(); it != v.end(); it++) {
        for(it2 = it->begin(); it2 != it->end(); it2++) {
            cout << *it2 << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Auto

- Ao invés de explicitamente colocar o tipo da variável ou objeto, é possível utilizar a palavra **auto**, onde o compilador irá inferir o tipo em tempo de compilação
  - Surgiu a partir do C++11
  - Precisa sempre ser seguido de uma atribuição
  - Facilita a escrita de iteradores mais complexos

```
auto x = 5;          // int
auto y = 5.0;        // double
auto z = "IDP";      // const char *
```

Auto

```
int main()
{
    vector<vector<int>> v = {{1, 2, 3}, {4, 5, 6}};

    for(auto it = v.begin(); it != v.end(); it++) {
        for(auto it2 = it->begin(); it2 != it->end(); it2++) {
            cout << *it2 << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Range Based Loops

- Além do **`auto`**, é possível iterar sobre uma coleção de itens utilizando range based loops
  - Sintaxe:

```
for (auto x: V) {
    // ...
}
```

- A cada iteração, **x** recebe o próximo valor da coleção **V**
  - Esse valor é uma cópia

Range Based Loops

- É possível utilizar range based loops alterando o conteúdo do container, por meio de referências:

```
for (auto &x: V) {
    // ...
}
```

Range Based Loops

- Se a intenção for evitar a cópia, e ainda assim acessar o conteúdo original sem modificá-lo, é possível utilizar **`const`**

```
for (const auto &x: V) {
    // ...
}
```

Pares e Tuplas

Pares e Tuplas

- O intuito de pares e tuplas é agregar múltiplos valores sob um único identificador
  - Não é necessário que os objetos sejam do mesmo tipo
  - Permitem atribuir a variáveis individuais via biding (C++17)
    - Structured Binding
  - Acesso pode ser feito utilizando o seguinte:
    - `get<pos>(estrutura)`

Pares e Tuplas

- Pares
  - Pares permitem agregar dois objetos em uma única estrutura
    - Acesso aos membros é feito por .first e .second

```
pair<int, int> p1 = {1, 2};
pair<int, double> p2 = {1, 2.5};
pair<double, string> p3 = make_pair(2.5, "IDP");
pair<vector<int>, pair<int, double>> p4 = {{1, 2, 3}, p2};

cout << p1.first << " - " << p1.second << endl;                // 1 - 2
cout << p3.first << " - " << get<1>(p3) << endl;               // 2.5 - IDP
cout << p4.first.size() << " - " << p4.second.first << endl;   // 3 - 1
```

Pares e Tuplas

- Tuplas
  - São uma generalização de pares para 2 ou mais elementos

```
pair<int, double> p1 = {1, 2.5};
tuple<int, double, char, string> t1 = {1, 2.5, 'a', "IDP"};
tuple<int, double, char, string> t2 = make_tuple(1, 2.5, 'a', "IDP");

auto [r, s, t, u] = t1;
auto [i, d] = p1;
```

Funções Lambda

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

```cpp
bool eh_primo(int n)
{
    if (n <= 1) return false;
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) return false;
    }
    return true;
}

int main()
{
    cout << eh_primo(99991) << endl;
    return 0;
}
```

```cpp
int main()
{
    auto eh_primo = [](int n) {
        if (n <= 1) return false;
        for (int i = 2; i <= sqrt(n); i++) {
            if (n % i == 0) return false;
        }
        return true;
    };

    cout << eh_primo(99991) << endl;

    return 0;
}
```

Funções Lambda

- Funções lambda tem utilidade maior, quando combinadas com funções que recebem funções como parâmetros
  - A biblioteca <algorithms> possui diversas dessas funções
    - Exemplo:
      - **`for_each`**: percorre um container e aplica a função a cada elemento do container

Funções Lambda

```
int main()
{
    vector<int> tab = {1, 2, 3, 4, 5};

    for_each(tab.begin(), tab.end(), [](int &x) {x = x * 3;});

    for (auto x: tab) {
        cout << x << endl;
    }

    return 0;
}
```

Alocação Dinâmica

Alocação Dinâmica

- Em C++, a alocação e liberação dinâmica de memória são realizadas pelas operações new e delete
- A sintaxe é a seguinte:
  - tipo *p = new { tipo | tipo(valor_inicial) | tipo[tamanho] }
- O operador new retorna (caso ok), um ponteiro para o elemento ou vetor de elementos requisitado
  - Em caso de falha, retorna exceção std::bad_alloc
  - Além disso, não é necessário coerção (como em C)

Alocação Dinâmica

- Finalizado o uso da memória alocada, ela deve ser liberada através da chamada do operador delete, cuja sintaxe é
  - delete p;
  - delete [] array;
- Os operadores new e delete podem ser sobrescritos para operar da maneira desejada pelo programador

Alocação Dinâmica

```
int main()
{
    int *vet = new int[10];

    for (int i = 0; i < 10; i++) vet[i] = i * i;

    for (int i = 0; i < 10; i++) cout << vet[i] << endl;

    delete[] vet;

    return 0;
}
```

Conclusão