Classes

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP
Baseado nos slides do Prof. Marco Antônio Carvalho

Introdução I

- Estamos acostumados a criar programas que:
  - Apresentam mensagens ao usuário;
  - Obtêm dados do usuário;
  - Realizam cálculos e tomam decisões.
- Todas estas ações eram delegadas à função *main* ou outras;
- A partir de agora, nossos programas terão uma função *main* e uma ou mais classes
- Cada classe consistindo de **dados e funções**.

Introdução II

- Suponhamos que queremos **dirigir um carro e acelerá-lo**, de modo que ele fique mais veloz;
- Antes disto, alguém precisa projetar e construir o carro;
- O projeto do carro tipicamente começa com desenhos técnicos
  - Que incluem o pedal do acelerador que utilizaremos.

Introdução III

- De certa forma, o pedal do acelerador esconde os mecanismos complexos que fazem com que o carro acelere
  - Isto permite que pessoas sem conhecimento de mecânica acelerem um carro;
  - Acelerar é uma "interface" mais amigável com a mecânica do motor.
- Acontece que não podemos dirigir os desenhos técnicos
  - É necessário que alguém construa o carro, com o pedal.

Introdução IV

- Depois de construído, o carro não andará sozinho
- Precisamos apertar o pedal de acelerar.
- Vamos fazer uma analogia com programação orientada a objetos.

Introdução V

- Realizar uma tarefa em um programa requer uma função
  - O *main*, por exemplo.
- A função descreve os mecanismos que realizam a tarefa
  - Escondendo toda a complexidade do processo, assim como o acelerador.
- Começaremos pela criação de uma classe, que abriga uma função

Introdução VI

- Uma **função** pertencente a uma classe é **chamada método**
  - São utilizadas para desempenhar as tarefas de uma classe.
- Da mesma forma que não é possível dirigir o projeto de um carro, você não pode "dirigir" uma classe;
- É necessário antes construir o carro para dirigí-lo;
- É necessário criar um objeto de uma classe antes para poder executar as tarefas descritas por uma classe.

Introdução VII

- Ainda, **vários carros podem ser criados** a partir do projeto inicial
  - Vários objetos podem ser criados a partir da mesma classe.
- Quando dirigimos um carro, pisar no acelerador **manda uma mensagem** para que o carro desempenhe uma tarefa
  - "Acelere o carro".

Introdução VIII

- Similarmente, enviamos mensagens aos objetos
  - As chamadas aos métodos;
- Além das competências de um carro, ele possui diversos atributos
  - Número de passageiros;
  - Velocidade atual;
  - Nível do tanque de combustível, etc.

Introdução IX

- Assim como as competências, os atributos também são representados no projeto de um carro
  - Cada carro possui seus **atributos próprios**;
  - Um carro **não conhece os atributos de outro**.
- Da mesma forma ocorre com os objetos
  - Cada objeto tem os seus próprios atributos

Classes I

- Vejamos um exemplo de uma classe que descreve um "diário de classe"
  - Utilizado para manter dados sobre a avaliação de alunos.
- Vejamos também como criar objetos desta classe.

Classes II

- Definir uma classe é dizer ao compilador quais métodos e atributos pertencem à classe
  - A definição começa com a palavra class;
  - Em seguida, o nome da classe
- Por padrão, a primeira letra de cada palavra no nome é maiúscula.
- O corpo da classe é delimitado por e ;
- Não se esqueça do ; no final.

Classes III

- Declaração de uma classe

```
class Nome_Classe{

    atributos ..

public:

    métodos ..

}; // fim da classe
```

Classes IV

```
#include <iostream>
using namespace std;

// Definição da classe DiarioClasse
class DiarioClasse{
    public:
    // método que exibe uma mensagem de boas-vindas
    void mostraMensagem(){
        cout << "Seja Bem-vindo ao Diario de Classe"
             << endl;
    }
};
int main(){
    // cria um objeto da classe DiarioClasse
    DiarioClasse meuDiario;
    // chama ao método mostraMensagem
    meuDiario.mostraMensagem();
    return 0;
}
```

Classes V

- Dentro da classe definimos os métodos e os atributos
  - Separados pelo especificador de acesso (visibilidade)
  - Basicamente, público (public) , privado (private) e protegido (protected).

Classes VI

- O método *mostraMensagem()* é público, pois está declarado depois deste especificador
  - Ou seja, pode ser chamado por outras funções do programa e por métodos de outras classes.
- Especificadores de acesso são sempre seguidos de :

Classes VII

- A definição de um método se assemelha a definição de uma função
  - Possui valor de retorno;
  - Assinatura do método
  - Lista de parâmetros

Classes VIII

- **Assinatura:** por padrão, começa com uma letra minúscula e todas as palavras seguintes começam com letras maiúsculas.
- **Lista de parâmetros:** va entre parênteses (tipo e identificador). Eventualmente, vazia Além disso, o corpo de um método também é delimitado por e .

Classes IX

- O corpo de um método contém as instruções que realizam uma determinada tarefa
- Neste exemplo, simplesmente apresenta uma mensagem

Classes X

```
#include <iostream>
using namespace std;

class DiarioClasse{
    public:
    void mostraMensagem(){
        cout << "Seja Bem-vindo ao Diario de Classe"
             << endl;
    }
};
int main(){
    // cria um objeto da classe DiarioClasse
    DiarioClasse meuDiario;
    // chama ao método mostraMensagem
    meuDiario.mostraMensagem();
    return 0;
}
```

Classes XI

- Para utilizarmos a classe *DiarioClasse* em nosso programa, precisamos criar um objeto
  - Há uma exceção em que é possível usar métodos sem criar objetos.
- Para criarmos um objeto, informamos o nome da classe como um tipo, e damos um identificador ao objeto;
- Uma vez criado o objeto, podemos utilizar seus métodos;
- Note que também podemos criar vetores e matrizes de objetos.

Classes XII

- Para utilizar um método, utilizamos o operador **.**
  - Informamos o nome do objeto, seguido por **.** e o nome do método e eventuais parâmetros
- O método possui acesso aos atributos do objeto que o chamou
  - Não possui acesso aos atributos de outros objetos, a não ser que estes sejam passados por parâmetro;

Classes XIII

- Como seria o diagrama de classe UML para a classe DiarioClasse?

Classe *string* I

- Existem várias classes prontas, com métodos úteis para várias finalidades
- Uma delas é a classe *string*, utilizada para manipulação de strings.
- Vamos incorporar a classe *string* em nosso exemplo anterior
  - Como um parâmetro para o método do exemplo

Classe *string* II

```
#include <iostream>
#include <string>
using namespace std;
class DiarioClasse{
    public:
    void mostraMensagem(string nomeDisciplina){
        cout << "Seja Bem-vindo ao Diario de Classe de"
             << nomeDisciplina << endl;
    } // fim da função
}; // fim da classe
int main(){
    string nome;
    DiarioClasse meuDiario;
    cout << "Digite o nome da disciplina: ";
    getline(cin, nome);
    meuDiario.mostraMensagem(nome);
    return 0;
}
```

Classe *string* III

- Para utilizarmos a classe *string*, precisamos incluir o arquivo <string>
  - Depois podemos criar um objeto desta classe.
- Para lermos uma *string* com espaços em branco, utilizamos a instrução *getline*.

Classe *string* IV

- Existem duas sintaxes para a instrução getline
  - Uma lê caracteres até que seja encontrado o final da linha

```
getline (cin, nameOfCourse);
```

  - A outra lê caracteres até que seja encontrado um caractere de terminação especificado por nós

```
getline (cin, nameOfCourse, 'A');
```

    Neste exemplo, a instrução lê caracteres até achar um 'A', que não será incluído na *string*.

Classe *string* V

- Métodos podem receber parâmetros assim como as funções
  - Basta definir o tipo do parâmetro e seu identificador entre parênteses
  - Se não houver parâmetros, deixamos os parênteses sem conteúdo.

Classe *string* VI

- Como fica o diagrama de classe UML para a classe *diarioClasse* agora que temos um parâmetro para o método *mostraMensagem*?

Atributos I

- Nossa classe não possui atributos, apenas um método;
- Atributos são representados como variáveis na definição de uma classe
  - Declarados dentro da classe
  - Porém, fora dos métodos.
- Cada objeto da classe possui sua própria cópia dos atributos.

Atributos II

```
#include <iostream>
#include <string>
using namespace std;
class DiarioClasse{
    string nomeDaDisciplina;
    public:
    void mostraMensagem(string nomeDisciplina){
        cout << "Seja Bem-vindo ao Diario de Classe de"
            << nomeDisciplina << endl;
    } // fim da função
}; // fim da classe
int main(){
    string nome;
    DiarioClasse meuDiario;
    cout << "Digite o nome da disciplina: ";
    getline(cin, nome);
    meuDiario.mostraMensagem(nome);
    return 0;
}
```

Getters e Setters I

- Atributos são definidos como privados
- Só podem ser alterados dentro da própria classe
- Tentar acessá-los fora da classe causará erro.
- Ocultação de informação;

Getters e Setters II

- Um método que altere o valor de um atributo é chamado de **setter**
  - Por padrão, a nomenclatura é set+[nome do atributo].
- Um método que retorne o valor de um atributo é chamado de getter
  - Por padrão, a nomenclatura é get+[nome do atributo].

Getters e Setters III

- Uma vez que nossa classe possui *getters* e *setters*, os atributos só devem ser acessados e alterados por eles
- **Mesmo dentro de outros métodos** que porventura necessitem acessar/alterar os atributos.
- Vamos alterar nosso exemplo anterior para que a *string* utilizada seja agora um atributo
- Com *getter* e *setter*.

Getters e Setters IV

```
#include <iostream>
#include <string>
using namespace std;
class DiarioClasse{
    string nomeDaDisciplina;
    public:
    void setNomeDaDisciplina(string name){
        nomeDaDisciplina = name;
    }
    string getNomeDaDisciplina(){
        return nomeDaDisciplina;
    }
    void mostraMensagem(){
        cout << "Seja Bem-vindo ao Diario de Classe de"
             << getNomeDaDisciplina() << endl;
    } // fim da função
}; // fim da classe
```

Getters e Setters V

```
int main(){
    string nome;
    DiarioClasse meuDiario;
    cout << "Digite o nome da disciplina: ";
    getline(cin, nome);
    meuDiario.setNomeDaDisciplina(nome);
    ...
    meuDiario.mostraMensagem();

    return 0;
}
```

Getters e Setters VI

- Para garantir a consistência dos valores dos atributos, convém realizar **validação de dados** nos getters e setters
  - O valor passado como parâmetro é adequado em relação ao tamanho ou faixa de valores?
  - Temos que definir se permitiremos valores negativos, tamanho máximo para vetores, etc.
  - Se um parâmetro for inadequado, devemos atribuir um valor padrão ao atributo
  - Zero, um, NULL, ”, etc.

Getters e Setters VII

- Notem um detalhe interessante:
  - Quando usamos a classe string, podemos fazer atribuição direta entre os objetos, utilizando o operador de atribuição
  - De fato, podemos fazer atribuição direta entre quaisquer objetos de uma mesma classe
  - **Cuidado:** pode causar erros se entre os atributos possuirmos ponteiros para memória alocada dinamicamente.
  - Diferentemente do que ocorre com vetores de caracteres, que precisam da função strcpy.

Getters e Setters VIII

- Como fica o diagrama de classe UML para a classe *DiarioClasse* agora que temos um getter e um setter?

Exercicio I

Implemente a classe Retângulo e as operações para calcular a área e perímetro.

Exercicio II

```
#include <iostream>
using namespace std;
class Retangulo{
    double altura, base;
    public:
    void setAltura(double m_altura){
        altura = m_altura;
    }
    void setBase(double m_base){
        base = m_base;
    }
    double getAltura(){return altura;}
    double getBase(){return bas;}
    double calculaArea(){
        return getAltura() * getBase();}
    double calculaPerimetro(){
        return 2 * getAltura() + 2 * getBase();}
}; // fim da classe
```

Exercicio III

```
int main(){
    Retangulo R;
    double m_base, m_altura;
    cout << "Digite a base e altura do retangulo: ";
    cin >> m_base >> m_altura;
    R.setBase(m_base);
    R.setAltura(m_altura);
    cout << "Area: " << R.calculaArea() << endl;
    cout << "Perimetro: " << R.calculaPerimetro() << endl;
    return 0;
}
```

FIM