

Linguagem de Programação I

Aula 22 - Introdução a Classes

Objetivos da aula

- Introduzir os conceitos de classes e objetos
- Para isto, estudaremos:
 - Criar tipos compostos através de classes
 - Manipular estes tipos em linguagem C++
- Ao final da aula espera-se que o aluno seja capaz de:
 - Entender os conceitos básicos essenciais do paradigma orientado a objetos
 - Implementar programas utilizando classes e objetos em C++

Contexto

- Registros (`struct`) em `C` e `C++` são tipos de dados compostos
 - Podem ser definidos pelo usuário
 - Refletem os conceitos do paradigma estruturado
- Exemplo: Modelagem de um retângulo e funções para manipular este retângulo em `C`

```
1. struct Retangulo
2. {
3.     int largura;
4.     int altura;
5. };
```

```
1. int area( struct Retangulo *rect )
2. {
3.     return rect->largura * rect->altura;
4. }
5.
6. int perimetro( struct Retangulo *rect )
7. {
8.     return 2 * ( rect->largura + rect->altura );
9. }
```


Orientação a objetos

- **Objetos** são tipos compostos definidos pelo usuário
- Eles podem conter
 - **Atributos**: informações que um objeto armazena
 - **Métodos**: funções que determinam o comportamento do objeto



Classes (ou Classes de Objetos)

- **Classes** são definições compostas de membros a partir das quais os objetos podem ser criados
- Elas determinam quais são os atributos e métodos de um objeto

Sintaxe

```
1. class nomeDaClasse
2. {
3.     corpoDaClasse;
4. };
```

Exemplo do modelo de um Retângulo em C++

```
1. class Retangulo
2. {
3.     int largura;
4.     int altura;
5.
6.     int area()
7.     {
8.         return largura * altura;
9.     }
10.
11.    int perimetro()
12.    {
13.        return 2 * ( largura + altura );
14.    }
15.};
```

Categorias de permissão

- **Membros** (atributos e métodos) de uma classe podem ser:
 - `public`
 - Podem ser acessados a partir de qualquer lugar (dentro ou fora da classe)
 - `private`
 - Só podem ser acessados por membros da própria classe
 - `protected`
 - Podem ser acessados apenas por membros da própria classe ou das suas sub-classes (herança)
- Por **default**, todo membro de uma classe é considerado `private`

Exemplo 1

- Modelo de um Retângulo em C++ com categorias de permissão

```
1. class Retangulo
2. {
3.     int largura;
4.
5. private:
6.     int altura;
7.
8. public:
9.     int area()
10.    {
11.        return largura * altura;
12.    }
13.
14. protected:
15.     int perimetro()
16.     {
17.         return 2 * ( largura + altura );
18.     }
19. };
```

```
1. // Exemplos de acesso
2. int main()
3. {
4.     Retangulo rect;
5.
6.     rect.altura = 10;      // Incorreto
7.     rect.largura = 40;    // Incorreto
8.
9.     int a = rect.area();  // Correto
10.    a = rect.perimetro();  // Incorreto
11.
12.    return 0;
13. }
```

Classes e registros

- C++ permite a criação de registros (`struct`) com métodos membro
- Os registros são praticamente idênticos às classes, porém todos os seus membros são `public` por *default*

```
1. struct Retangulo
2. {
3.     int getLargura() { return largura; } // public por default
4.     int getAltura () { return altura; } // public por default
5.
6. private:
7.     int largura;
8.     int altura;
9. };
```


Operador ::

- Quando implementamos um método dentro de uma classe, o compilador copia e cola o código sempre que o método é chamado
 - O método é dito `inline`
 - Isto torna o código executável mais rápido
 - Mas deixa o código executável bem maior
 - Portanto, somente deve ser usado para métodos muito curtos
- Qual é a solução correta para implementar métodos mais longos?
 - Implementar o método fora da classe
 - Utilizar o operador `::`

Operador ::

- O operador `::` permite a implementação de métodos fora da classe
- A classe passa a possuir apenas o protótipo do método, o corpo podendo ficar no mesmo arquivo ou em outro separado
- Sintaxe:
 - `nomeDaClasse::nomeDoMembro`
- Caso queira implementar métodos curtos fora da classe, também pode-se usar o modificador `inline` juntamente com o operador `::` para tornar o método `inline`

Exemplo 2

```
1. class Retangulo
2. {
3. private:
4.     int largura;
5.     int altura;
6.
7. public:
8.     int area();
9.     int perimetro();
10. };
11.
12. int inline Retangulo::area() // força o método a ser inline
13. {
14.     return largura * altura;
15. }
16.
17. int Retangulo::perimetro()
18. {
19.     return 2 * ( largura + altura );
20. }
```


Construtor

- Método especial que é chamado quando um novo objeto é criado
- Deve possuir o mesmo nome da classe
- Não possui retorno
- É utilizado para inicializar os atributos da classe

```
1.  class Retangulo
2.  {
3.  private:
4.      int largura;
5.      int altura;
6.  public:
7.      Retangulo( int l, int a ); // construtor com dois parâmetros
8.  };
9.
10. Retangulo::Retangulo( int l, int a )
11. {
12.     largura = l;
13.     altura = a;
14. }
```

Destrutor

- Método especial que é chamado automaticamente quando um objeto está prestes a ser liberado da memória
- Deve ter o mesmo nome da classe precedido por um ~
- Assim como o construtor, o destrutor não possui retorno
- Além disso, ele não pode ter parâmetros

```
1. class Retangulo
2. {
3. private:
4.     int largura;
5.     int altura;
6.
7. public:
8.     Retangulo() {} // construtor padrão
9.     ~Retangulo() {} // destrutor padrão
10.};
```

Métodos de acesso **get** e **set**

- Método **get**
 - Serve para ter acesso aos atributos encapsulados de uma classe
 - Exemplo: `int getLargura() { return largura; }`
- Método **set**
 - Útil para permitir a modificação dos atributos encapsulados de uma classe
 - Exemplo: `void setLargura(int l) { largura = l; }`

Exemplo 3

```
1.  class Retangulo
2.  {
3.  private:
4.      int largura;
5.      int altura;
6.
7.  public:
8.      int getLargura() { return largura; }
9.      int getAltura () { return altura; }
10.
11. protected:
12.     void setLargura( int l )
13.     {
14.         if( l > 0 ) // evita um valor inválido
15.             largura = l;
16.     }
17.     void setAltura( int a )
18.     {
19.         if( a > 0 ) // evita um valor inválido
20.             altura = a;
21.     }
22. };
```

Métodos membros constantes

- Quando não é necessário modificar dados membro de uma classe através de um método membro, uma boa prática consiste em declarar este método como constante (`const`)
- Métodos membro constantes não permitem
 - Modificar dados membros de uma classe
 - Fazer chamada a outros métodos membro não constantes
- Isto protege o programa contra alterações acidentais
 - Exemplo: `int getAltura() const { return altura; }`

Exemplo 4

```
1. class Retangulo
2. {
3.     private:
4.         int largura;
5.         int altura;
6.
7.     public:
8.         int getLargura() const { return largura; }
9.         int getAltura () const { return altura; }
10.
11.     protected:
12.         void setLargura( int l )
13.         {
14.             if( l > 0 ) // evita um valor inválido
15.                 largura = l;
16.         }
17.         void setAltura( int a )
18.         {
19.             if( a > 0 ) // evita um valor inválido
20.                 altura = a;
21.         }
22.     };
```


Membros estáticos

- Objetos são ótimos pois cada instância deles armazena suas próprias informações de uma classe, criando uma identidade única
- Mas se quisermos armazenar informações de uma classe como um todo, por exemplo o número total de instâncias criadas para ela?
- Para isso, podemos usar o que chamamos de **membros estáticos**
 - Uma única instância destes membros existirão para todos os objetos criados
 - Exemplo:
 - Dado: `static int total;`
 - Método: `static int getTotal() { return total; }`

Exemplo 5

```
1. class Retangulo
2. {
3.     private:
4.         int largura;
5.         int altura;
6.         static int total;
7.
8.     public:
9.         int getLargura() const { return largura; }
10.        int getAltura () const { return altura; }
11.        static int getTotal() const { return total; }
12.
13.    protected:
14.        void setLargura( int l )
15.        {
16.            if( l > 0 ) // evita um valor inválido
17.                largura = l;
18.        }
19.        void setAltura( int a )
20.        {
21.            if( a > 0 ) // evita um valor inválido
22.                altura = a;
23.        }
24.    };
```

Simulando um animal de estimação virtual

- Jogos com animais de estimação colocam o jogador no comando de seus próprios animais de estimação virtuais
 - O jogador é completamente responsável por manter o seu animal contente, o que não é sempre fácil
 - Ele pode alimentar e brincar com o animal para mantê-lo de bom humor
 - Ele também pode ouvir o animal para saber o quanto ela está contente, o que pode variar de feliz a triste



Simulando um animal de estimação virtual (versão 1 - como uma struct)

```
1. // definição de uma nova classe
2. class Animal
3. {
4. public:
5.     int nivelFome;    // dado membro
6.
7.     void situacao(); // método membro
8. };
9.
10. // definição do método membro
11. void Animal::situacao()
12. {
13.     std::cout << "Ola!"
14.               << "Eu sou seu animal! "
15.               << "Meu nivel de fome eh "
16.               << nivelFome << std::endl;
17. }
```

```
1. #include <iostream>
2. int main()
3. {
4.     Animal gato;
5.     Animal cachorro;
6.
7.     gato.nivelFome = 9;
8.     std::cout << "Fome do gato eh "
9.               << gato.nivelFome
10.              << std::endl;
11.
12.     cachorro.nivelFome = 3;
13.     std::cout << "Fome do cachorro eh "
14.               << cachorro.nivelFome
15.               << std::endl;
16.
17.     gato.situacao();
18.     cachorro.situacao();
19.
20.     return 0;
21. }
```

Simulando um animal de estimação virtual (versão 2 - com construtor)

```
1. class Animal
2. {
3. public:
4.     int nivelFome;    // dado membro
5.
6.     Animal( int fome = 0 ); // construtor
7.     void situacao(); // método membro
8. };
9.
10. Animal::Animal( int fome )
11. {
12.     std::cout << "Um novo animal nasceu!" << std::endl;
13.     nivelFome = fome;
14. }
15.
16. void Animal::situacao()
17. {
18.     std::cout << "Ola!"
19.               << "Eu sou seu animal! "
20.               << "Meu nivel de fome eh "
21.               << nivelFome << std::endl;
22. }
```

```
1. #include <iostream>
2. int main()
3. {
4.     Animal gato( 7 );
5.     Animal cachorro( 10 );
6.
7.     gato.situacao();
8.     cachorro.situacao();
9.
10.    return 0;
11. }
```

Simulando um animal de estimação virtual (versão 3 - com privacidade)

```
1. class Animal
2. {
3. public: // início da seção pública
4.     Animal( int fome = 0 );
5.     int getNivelFome() const;
6.     void setNivelFome( int fome );
7.
8. private: // início da seção privada
9.     int nivelFome;
10. };
11.
12. Animal::Animal( int fome )
13. {
14.     std::cout << "Um novo animal nasceu!"
15.               << std::endl;
16.     nivelFome = fome;
17. }
18.
19. int Animal::getNivelFome() const
20. {
21.     return nivelFome;
22. }
```

```
1. void Animal::setNivelFome( int fome )
2. {
3.     if( nivelFome < 0 )
4.         std::cout << "Valor incorreto!"
5.                   << std::endl;
6.     else
7.         nivelFome = fome;
8. }
9.
10. #include <iostream>
11. int main()
12. {
13.     Animal gato( 5 );
14.     std::cout << gato.getNivelFome();
15.
16.     gato.setNivelFome( -1 );
17.     std::cout << gato.getNivelFome();
18.
19.     gato.setNivelFome( 9 );
20.     std::cout << gato.getNivelFome();
21.     return 0;
22. }
```


Simulando um animal de estimação virtual (versão 4 - com membros static)

```
1. class Animal
2. {
3. public:
4.     // dado membro estático
5.     static int total;
6.     Animal( int fome = 0 );
7.     // método membro estático
8.     static int getTotal() const;
9.
10. private:
11.     int nivelFome;
12. };
13.
14. // inicialização do dado membro estático
15. int Animal::total = 0;
16.
17. Animal::Animal( int fome )
18. {
19.     std::cout << "Um novo animal nasceu!"
20.               << std::endl;
21.     nivelFome = fome;
22.     ++total;
23. }
```

```
1. int Animal::getTotal() const
2. {
3.     return total;
4. }
5.
6. #include <iostream>
7. int main()
8. {
9.     std::cout << "O total de animais eh "
10.               << Animal::total
11.               << std::endl;
12.
13.     Animal gato, cachorro, coelho;
14.
15.     std::cout << "O total de animais eh "
16.               << Animal::getTotal()
17.               << std::endl;
18.
19.     return 0;
20. }
```

Resumo da aula

- Objetos são tipos compostos de atributos e métodos definidos pelo usuário
- Classes são definições compostas a partir das quais os objetos podem ser criados
- As classes incorporam diversos conceitos do paradigma orientado a objetos
- Elas são a base para o aprendizado deste paradigma que tenta simular os objetos do mundo real e portanto facilita bastante o desenvolvimento de programas