Univiçosa – Centro Universitário de Viçosa
Cursos: Análise e Desenvolvimento de Sistemas e Engenharia de Computação
Disciplina: ADS101/ECO131 – Programação de Computadores 2

Aula 12 – Encapsulamento

1) Introdução

Encapsulamento é um dos pilares da programação orientada a objetos (POO) e tem a ver com **restringir o acesso direto aos dados de um objeto**, permitindo o controle desse acesso por meio de métodos específicos. Em C++, ele é implementado principalmente usando os **modificadores de acesso**: `public`, `protected` e `private`.

2) Conceito de Encapsulamento

1. **Proteção dos dados**: Com o encapsulamento, variáveis internas (atributos) de uma classe ficam protegidas, permitindo que apenas a própria classe controle como esses dados são acessados e modificados.

2. **Interfaces seguras**: Ao invés de expor diretamente os atributos de uma classe, criamos métodos que oferecem acesso controlado a esses dados. Isso garante que os dados sejam usados corretamente e que não ocorra acesso direto que possa comprometer o funcionamento do programa.

Imagine que queremos uma classe que represente um **retângulo**. Queremos garantir que a largura e a altura sempre sejam positivas. Vamos criar métodos para definir e acessar esses valores, aplicando o encapsulamento para proteger os atributos.

```
#include <iostream>

class Retangulo {
private:
    double largura;
    double altura;

public:
    // Construtor que inicializa largura e altura
    Retangulo(double larg, double alt) {
        setLargura(larg);
        setAltura(alt);
    }

    // Método para definir a largura, com verificação
    void setLargura(double larg) {
```

```
        if (larg > 0) {
            largura = larg;
        } else {
            std::cout << "Largura inválida. Deve ser positiva.\n";
            largura = 1.0; // valor padrão
        }
    }

    // Método para definir a altura, com verificação
    void setAltura(double alt) {
        if (alt > 0) {
            altura = alt;
        } else {
            std::cout << "Altura inválida. Deve ser positiva.\n";
            altura = 1.0; // valor padrão
        }
    }

    // Métodos para obter largura e altura
    double getLargura() const {
        return largura;
    }

    double getAltura() const {
        return altura;
    }

    // Método para calcular a área do retângulo
    double calcularArea() const {
        return largura * altura;
    }
};

int main() {
    Retangulo ret(5.0, 3.0);  // cria um retângulo com largura 5 e altura 3
    std::cout << "Largura: " << ret.getLargura() << "\n";
    std::cout << "Altura: " << ret.getAltura() << "\n";
    std::cout << "Área: " << ret.calcularArea() << "\n";

    ret.setLargura(-4.0); // tenta definir uma largura negativa
    std::cout << "Largura após ajuste: " << ret.getLargura() << "\n";
    std::cout << "Área após ajuste: " << ret.calcularArea() << "\n";

    return 0;
}
```

Explicando o código

1. **Atributos Privados**:
   - `largura` e `altura` são privados, então só podem ser modificados dentro da classe `Retangulo.`

2. **Métodos setLargura e setAltura**:
   - Esses métodos públicos permitem definir valores para `largura` e `altura;`
   - Eles garantem que o valor passado seja positivo. Caso contrário, definem um valor padrão (1.0) e mostram uma mensagem de erro.

3. **Métodos getLargura e getAltura**:
   - Esses métodos retornam os valores de `largura` e `altura`, sem permitir acesso direto aos atributos.

4. **Método calcularArea**:
   - Método para calcular a área do retângulo, usando os valores de `largura` e `altura` encapsulados.

Benefícios do Encapsulamento

Graças ao encapsulamento, garantimos que:

- `largura` e `altura` nunca terão valores negativos.
- Podemos confiar que `Retangulo` sempre estará em um estado válido, onde a área será calculada corretamente.

3) Aplicando o encapsulamento em um exemplo prático

Vamos criar uma classe `ContaBancaria` que armazena o saldo de uma conta bancária e usa encapsulamento para proteger esse saldo:

```
#include <iostream>
#include <string>

class ContaBancaria {
private:
    std::string nomeCliente;
    double saldo;

public:
    ContaBancaria(const std::string& nome, double saldoInicial)
        : nomeCliente(nome), saldo(saldoInicial) {}

    // Método para depositar dinheiro
    void depositar(double valor) {
```

```
        if (valor > 0) {
            saldo += valor;
            std::cout << "Depósito de " << valor << " realizado com sucesso!\n";
        } else {
            std::cout << "Valor de depósito inválido.\n";
        }
    }

    // Método para sacar dinheiro
    void sacar(double valor) {
        if (valor > 0 && valor <= saldo) {
            saldo -= valor;
            std::cout << "Saque de " << valor << " realizado com sucesso!\n";
        } else {
            std::cout << "Saldo insuficiente ou valor inválido para saque.\n";
        }
    }

    // Método para exibir o saldo (getter)
    double getSaldo() const {
        return saldo;
    }

    // Método para exibir o nome do cliente
    std::string getNomeCliente() const {
        return nomeCliente;
    }
};

int main() {
    ContaBancaria conta("João", 1000.0);

    // Exibindo informações iniciais
    std::cout << "Cliente: " << conta.getNomeCliente() << "\n";
    std::cout << "Saldo inicial: " << conta.getSaldo() << "\n\n";

    // Testando os métodos de depósito e saque
    conta.depositar(200.0);
    std::cout << "Saldo após depósito: " << conta.getSaldo() << "\n";

    conta.sacar(150.0);
    std::cout << "Saldo após saque: " << conta.getSaldo() << "\n";

    return 0;
}
```

Explicação do Código

1. **Atributos Privados**:
    - `nomeCliente` e `saldo` são declarados como `private`. Isso significa que eles só podem ser acessados ou modificados por métodos dentro da própria classe `ContaBancaria`, garantindo que ninguém possa acessar diretamente esses atributos de fora da classe.

2. **Métodos Públicos (Interface)**:
    - `depositar` e `sacar` são métodos públicos que fornecem uma interface segura para interagir com o saldo. Eles fazem verificações antes de modificar o saldo, o que ajuda a garantir que operações inválidas (como depositar um valor negativo ou sacar mais do que o saldo) não sejam realizadas;
    - `getSaldo` e `getNomeCliente` são métodos públicos que permitem acessar o saldo e o nome do cliente de forma controlada, sem dar acesso direto aos atributos.

3. **Como o Encapsulamento Protege a Classe**:
    - Imagine que `saldo` fosse um atributo público. Alguém poderia fazer `conta.saldo = -1000;`, o que não faz sentido para uma conta bancária. Mas com o encapsulamento, a única maneira de alterar `saldo` é por meio dos métodos `depositar` e `sacar`, que garantem que o saldo nunca seja alterado de forma incorreta.

Resumindo o encapsulamento

- **Encapsulamento** permite que apenas métodos específicos controlem os dados internos de uma classe.
- **Modificadores de acesso** (`private`, `protected`, `public`) ajudam a definir a visibilidade e o acesso aos membros de uma classe.
- O encapsulamento garante que os dados da classe sejam protegidos contra manipulação incorreta, o que ajuda a evitar bugs e tornar o código mais seguro e fácil de manter.

Encapsulamento é a base para criar classes seguras e bem projetadas!

4) Mais um exemplo prático

Um exemplo de encapsulamento que simula uma situação bem prática: um sistema de gerenciamento de temperatura em um ambiente, como em um termostato de ar-condicionado. Esse sistema pode monitorar a temperatura e ter métodos para ajustar o valor, mas a temperatura precisa estar em uma faixa segura, entre 16°C e 30°C, para evitar configurações incorretas.

Observe o código abaixo:

```
#include <iostream>
```

```
class Termostato {
private:
    double temperatura;

public:
    Termostato(double tempInicial) {
        setTemperatura(tempInicial); // inicializa com uma temperatura válida
    }

    // Método para definir a temperatura, com verificação de faixa
    void setTemperatura(double temp) {
        if (temp >= 16.0 && temp <= 30.0) {
            temperatura = temp;
            std::cout << "Temperatura ajustada para " << temperatura << "°C\n";
        } else {
            std::cout << "Erro: Temperatura fora da faixa permitida (16°C - 30°C).\n";
        }
    }

    // Método para obter a temperatura atual
    double getTemperatura() const {
        return temperatura;
    }

    // Método para aumentar a temperatura, dentro dos limites
    void aumentarTemperatura(double incremento) {
        setTemperatura(temperatura + incremento);
    }

    // Método para diminuir a temperatura, dentro dos limites
    void diminuirTemperatura(double decremento) {
        setTemperatura(temperatura - decremento);
    }
};

int main() {
    Termostato termostato(22.0); // inicializa o termostato com 22°C

    termostato.aumentarTemperatura(5.0); // aumenta a temperatura para 27°C
    std::cout << "Temperatura atual: " << termostato.getTemperatura() << "°C\n";

    termostato.diminuirTemperatura(10.0); // tenta diminuir para 17°C
    std::cout << "Temperatura atual: " << termostato.getTemperatura() << "°C\n";

    termostato.setTemperatura(35.0); // tenta ajustar para fora do limite
    std::cout << "Temperatura atual: " << termostato.getTemperatura() << "°C\n";

    return 0;
}
```

Explicando o código

1. **Encapsulamento da temperatura**:
    - `temperatura` é privada e só pode ser acessada ou alterada usando métodos públicos da classe.

2. **Método `setTemperatura`**:
    - Este método define a temperatura apenas se o valor estiver dentro da faixa de segurança (16°C a 30°C). Caso contrário, exibe uma mensagem de erro e mantém a temperatura atual.

3. **Métodos `aumentarTemperatura` e `diminuirTemperatura`**:
    - Estes métodos usam `setTemperatura` para ajustar a temperatura, garantindo que ela sempre permaneça nos limites de segurança.

4. **Controle do valor com `getTemperatura`**:
    - `getTemperatura` retorna o valor atual da temperatura de forma segura, sem permitir acesso direto ao atributo.

Benefícios no Dia-a-Dia

Em situações práticas, como controlar a temperatura de um ambiente, o encapsulamento ajuda a:

- **Evitar configurações incorretas**: Impede que a temperatura seja ajustada para valores fora da faixa segura.
- **Manter o controle centralizado**: Todas as operações de ajuste de temperatura passam por `setTemperatura`, o que torna o sistema mais confiável e fácil de modificar, caso os limites de temperatura mudem.
- **Melhorar a segurança e confiabilidade**: Garante que a temperatura esteja sempre dentro dos valores esperados, evitando erros e mantendo o ambiente seguro.

Este exemplo mostra como o encapsulamento torna o código robusto e confiável em situações práticas do dia-a-dia!

4) Considerações da aula

Nesta aula aprendemos sobre encapsulamento.

Bons estudos.