UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Programação Orientada a Objetos (Encapsulamento)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

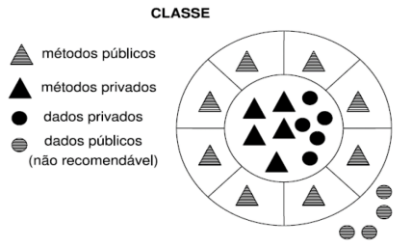
Introdução

- Abstração
  - Simplificação de um problema difícil
  - É o ato de representar as características essenciais sem incluir os detalhes por trás
- Ocultação de dados
  - Informações desnecessárias devem ser escondidas do mundo externo
  - Usuário do TAD x Programador do TAD

Introdução

- Encapsulamento
  - Mecanismo que coloca juntos os dados e suas funções associadas, mantendo-os controlados em relação ao seu nível de acesso
- Proporciona abstração
  - Separa a visão externa da visão interna
  - Protege a integridade dos dados do Objeto

Introdução



Introdução

- Escopo de uma variável
  - Região de um programa dentro da qual uma variável pode ser referenciada pelo seu nome
- Relação com a memória?
  - O escopo define quando o sistema aloca e libera memória para armazenar uma variável
  - Variáveis alocadas no heap continuam lá mesmo fora do escopo (se não forem desalocadas)

Introdução

```
class ClasseTeste {
  public:
    int var1;
    std::string var2;

    void metodo(int param) {
      int x = 1;
      int y = 9;

      if (param % 2 == 0) {
        int result = 12;
      }

    }
};
```

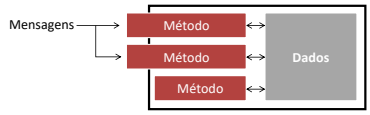
- Quais escopos?
  - Classe, Método, If
- O que acontece quando:
- Entra/sai do if?
  - `result` entra/sai da pilha
- O método termina?
  - `param`, `x` e `y` saem da pilha
- A classe é destruída?
  - `var1` e `var2` saem da pilha

Encapsulamento

- Encapsulamento ocorre nas classes
- O comportamento e a interface de uma classe são definidos pelos seus membros
  - Atributos
  - Métodos
- Fazem uso dos modificadores de acesso

Encapsulamento

- Informações encapsuladas em uma Classe
  - Estado (dados)
  - Comportamento (métodos)



Encapsulamento
Benefícios

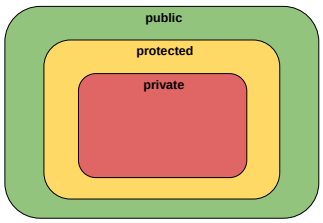
- Desenvolvimento
  - Melhor compreensão de cada classe
  - Facilita a realização de testes
- Manutenção/Evolução
  - Impacto reduzido ao modificar uma classe
  - Interface deve ser o mais constante possível

Encapsulamento
C++

- Modificadores de acesso
  - Public
  - Protected
  - Private
- Membros declarados após o modificador
- Checagem em tempo de compilação

https://en.cppreference.com/w/cpp/language/access

Encapsulamento
Modificadores de acesso



Encapsulamento
Modificadores de acesso – Public

- Permite que os membros públicos sejam acessados de qualquer outra parte do código
- Mais liberal dos modificadores
  - Fazem parte (definem) o contrato da classe
  - Deve ser usado com responsabilidade
  - Não é recomendado
    - Por que?

Encapsulamento
Modificadores de acesso – Public

Membros públicos.

```
class Ponto {

  public:
    int x;
    int y;

    Ponto(int x, int y) {
      this->x = x;
      this->y = y;
    }

};
```


Encapsulamento
Modificadores de acesso – Protected

- Permite que os membros possam ser acessados apenas por outras classes que
  - Fazem parte da hierarquia (derivadas)
  - Classes "amigas"
    - Algo bem específico em C++

Encapsulamento
Modificadores de acesso – Protected

```
class Base {
  protected:
    int i = 99;
};

class Derived : public Base {
  public:
    int f() {
      i++;
      return i;
    }
};
```

```
int main() {
  Base b;
  cout << b.i << endl;

  Derived d;
  cout << d.f() << endl;

  return 0;
}
```

Erro!

https://wandbox.org/permlink/I91umA4d5fxhaxbb

Encapsulamento
Modificadores de acesso – Private

- Permite que os membros privados sejam acessados por métodos da mesma classe
- O mais restritivo dos modificadores
  - Deve ser empregado sempre que possível
  - Utilizar métodos auxiliares de acesso
- Quando não há declaração explícita
  - Nível padrão (implícito)
  - Structs → Padrão é público

Encapsulamento
Modificadores de acesso – Private

```
class Ponto {

  private:
    int _x;
    int _y;

  public:
    Ponto(int x, int y) : _x(x), _y(y) {}

};
```


Encapsulamento
Modificadores de acesso – Private (Exemplo 1)

```
class Base {
  int i = 99;
};

class Derived : public Base {
  public:
    int f() {
      i++;
      return i;
    }
};
```

```
int main() {
  Base b;
  cout << b.i << endl;

  Derived d;
  cout << d.f() << endl;

  return 0;
}
```

Encapsulamento
Modificadores de acesso – Private (Exemplo 2)

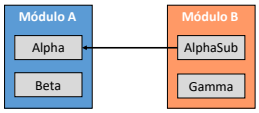
```
class Ponto {

  private:
    int _x;
    int _y;

    Ponto(int x, int y) : _x(x), _y(y) {}

};
```

Encapsulamento
Modificadores de acesso – Private (Exemplo 3)

```
class Ponto {

  private:
    class EstruturaPonto {
      public:
        double x;
        double y;
    };

    EstruturaPonto p;

  public:
    Ponto(int x, int y) {
      p.x = x;
      p.y = y;
    }
};
```


Encapsulamento
Modificadores de acesso

- Considere a visibilidade dos membros da **Alpha** de acordo com o modificador de acesso



	Alpha	AlphaSub	Beta	Gamma
Public	■	■	■	■
Protected	■	■	■	■
Private	■	■	■	■

Encapsulamento
Modificadores de acesso

	Classe	Subclasse	Mundo
Public	■	■	■
Protected	■	■	■
Private	■	■	■

Encapsulamento
Acessando e modificando atributos

- Evitar a manipulação direta de atributos
  - Acesso deve ser o mais restrito possível
  - De preferência todos devem ser private
- Sempre utilizar métodos auxiliares
  - Melhor controle das alterações
  - Acesso centralizado

Encapsulamento
Getters e Setters

- Convenção de nomenclatura dos métodos
- Get
  - Os métodos que permitem apenas o acesso de consulta (obter) devem possuir o prefixo get
- Set
  - Os métodos que possibilitam a alteração (definir) devem possuir o prefixo set

Encapsulamento
Getters e Setters

```
class Ponto {
   private:
      double _x;
      double _y;

   public:
      Ponto(double x, double y) : _x(x), _y(y) {}

      void setX(double x) { this->_x = x; }
      void setY(double y) { this->_y = y; }

      double getX() { return this->_x; }
      double getY() { return this->_y; }
};
```


Encapsulamento
Getters e Setters

- Todos os atributos devem possuir get e set
- Nomenclatura alternativa
  - Atributos booleanos devem utilizar o prefixo "is" ao invés do prefixo get
  - Melhora a legibilidade e entendimento
- E uma coleção, possui 'setColecao'?
  - Não!
  - Métodos auxiliares: adicionar, remover, …

Encapsulamento
Getters e Setters

```
class Cliente {
  private:
    string _nome;
    bool _ativo;

  public:
    Cliente(string nome, bool ativo) : _nome(nome), _ativo(ativo) {}

    void setNome(string nome) { this->_nome = nome; }
    void setAtivo(bool ativo) { this->_ativo = ativo; }

    string getNome() { return this->_nome; }
    bool isAtivo() { return this->_ativo; }
};
```


Exercício

- Modelar uma conta bancária
  - Quais atributos devem existir?
  - Quais métodos devem existir?

Exercício

```
class Conta {

  public:
    int agencia;
    int numero;
    double saldo;

    Conta(int agencia, int numero) : agencia(agencia), numero(numero){}

};
```


Exercício

```
class Conta {

  private:
    int _agencia;
    int _numero;
    double _saldo = 0;

  public:

    Conta(int agencia, int numero) : _agencia(agencia), _numero(numero){}

    void setAgencia(int ag) { this->_agencia = ag; }
    void setNumero(int num) { this->_numero = num; }
    void setSaldo(double saldo) { this->_saldo = saldo; }

    int getAgencia() { return this->_agencia; }
    int getNumero() { return this->_numero; }
    double getSaldo() { return this->_saldo; }
};
```

Exercício

```
class Conta {

   private:
      int _agencia;
      int _numero;
      double _saldo = 0;

   public:

      {...}

      void depositar(double valor) {
         this->_saldo += valor;
      }

      void sacar(double valor) {
         this->_saldo -= valor;
      }
};
```


Exercício

```
class Conta {

   {...}

   public:

      {...}

      void depositar(double valor) {
         this->_saldo += valor;
         this->_saldo -= 0.25;
      }

      void sacar(double valor) {
         this->_saldo -= valor;
         this->_saldo -= 0.25;
      }
};
```


Exercício

```
class Conta {

  {...}

  public:

    {...}

    void depositar(double valor) {
      this->_saldo += valor;
      descontarTarifa();
    }

    void sacar(double valor) {
      this->_saldo -= valor;
      descontarTarifa();
    }

    void descontarTarifa() {
      this->_saldo -= 0.25;
    }
};
```


Exercício

```
class Conta {

  private:
    {...}

    void _descontarTarifa() {
      this->_saldo -= 0.25;
    }

  public:
    {...}

    void depositar(double valor) {
      this->_saldo += valor;
      descontarTarifa();
    }

    void sacar(double valor) {
      this->_saldo -= valor;
      descontarTarifa();
    }
};
```


Exercício

```
class Conta {

  private:
    {...}

    double const _TARIFA = 0.25;

    void _descontarTarifa() {
      this->_saldo -= TARIFA ;
    }

  public:
    {...}

};
```


Exercício

```
class Conta {

  private:
    {...}

    static double constexpr _TARIFA = 0.25;

    void _descontarTarifa() {
      this->_saldo -= TARIFA ;
    }

  public:
    {...}

};
```

https://wandbox.org/permlink/gMs86xBtaTO0CLJj

https://en.cppreference.com/w/cpp/language/constexpr