# Introdução à Programação


## *Encapsulamento e Herança*

# Tópicos da Aula

- Hoje, aprenderemos conceitos mais avançados de Orientação a Objetos
  - Encapsulamento
    - Usando modificadores de acesso em C++
  - Herança
    - Importância
    - Utilização em C++
    - Herança múltipla

# Encapsulamento *(Information Hiding)*

- Um objeto pode ser visto de duas formas diferentes:
  - Internamente
    - Detalhes de atributos e métodos da classe que o define
  - Externamente
    - Serviços que um objeto fornece e como este objeto interage com o resto do sistema (a interface do objeto)

# Encapsulamento *(Information Hiding)*

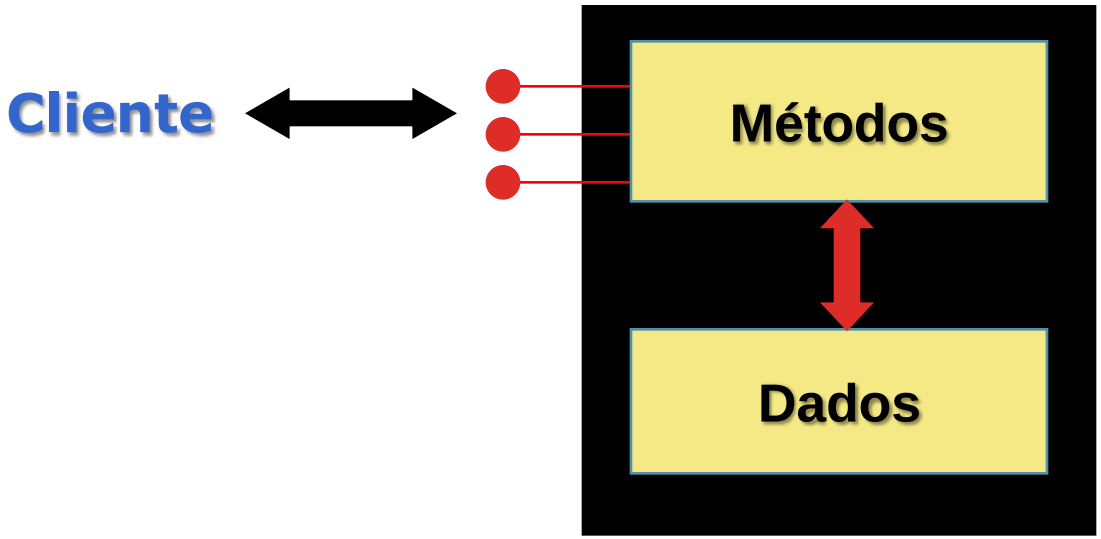
- A visão externa de um objeto **encapsula** o modo como são fornecidos os serviços
- Isto é, esconde os detalhes (internos) de implementação do objeto (**information hiding**)

# Encapsulamento *(Information Hiding)*

- Um objeto (chamado neste caso de cliente) pode usar os serviços fornecidos por um outro objeto
  - Contudo, um cliente não deve precisar saber os detalhes de implementação destes métodos
- Mudanças no estado (atributos) de um objeto devem ser feitas pelos métodos do objeto
- Para permitir uma maior independência entre os objetos,o acesso direto aos atributos de um objeto por um outro objeto deve ser restrito ou quase impossível

# Encapsulamento *(Information Hiding)*

- Um objeto pode ser visto como uma “caixa preta”, onde os detalhes internos são escondidos dos clientes
- Clientes acessam o estado do objeto, através dos métodos oferecidos



# Modificadores de Acesso

- Em C++, o encapsulamento é possível através do uso apropriado de **modificadores de acesso**
  - Modificadores são palavras reservadas que especificam características particulares de um conjunto de métodos ou de atributos
- Modificadores de acesso variam de acordo com a visibilidade que se quer oferecer ao cliente
- Podem ser:
  - **`public`**,**`protected`**,**`private`**,**`friend`**

# Modificadores de Acesso

- Membros da classe que recebem o modificador **`public`**, podem ser acessados por qualquer outra classe
  - Devem ser utilizados para métodos que definem a interface da classe
  - Não deve ser utilizado para os atributos, excetuando-se o caso onde queremos declarar uma constante
- Membros que recebem o modificador **`private`**, só podem ser acessados por membros da classe ou classes (ou funções) “amigas” (*friends*)
  - Devem ser utilizados para atributos e métodos auxiliares

# Classe Conta e Modificadores de Acesso

```
class Conta {
    private:
       string  numero;
       string agencia;
       double saldo;

    public:
     void creditar (double valor);
     void debitar(double valor);
     double getSaldo();
     string getNumero();
     string getAgencia();
     Conta(string num, string ag);

}
```

Parte privada

Parte pública

# Usando Classe Conta

```
#include "Conta.h"
using namespace std;
int main() {
   Conta*  ca;
   ca = new Conta("21.342-7","123-4");
   ca->creditar(500);
   ca->debitar(300);

   ca->saldo = 1000;
   cout << ca->getSaldo()<<endl;
}
```

Parte pública

Parte privada
Acesso ilegal
(Erro de compilação)

# Modificadores de Acesso

| | `public` | `private` |
|---|---|---|
| **Atributos** | **Violam encapsulamento** | **Preservam encapsulamento** |
| **Métodos** | **Fornecem serviços para os clientes** | **Auxiliam outros métodos da classe** |

# Modificador de Acesso protected

```
class Conta {
    private:
       string  numero;
       string agencia;
       double saldo;

    public:
     void creditar (double valor);
     void debitar(double valor);
     double getSaldo();
     string getNumero();
     string getAgencia();
     Conta(string num, string ag);

    protected:
     void setSaldo(double valor);
}
```

Parte privada

Parte pública

Parte protegida

# Modificador de Acesso `protected`

- Membros de uma superclasse que recebem o modificador **`private`**, não podem ser acessados nem pelas subclasses
  - Se colocar **`public`** e o membro for um atributo, o princípio do encapsulamento é violado
- Membros com o modificador **`protected`** são visíveis pelas subclasses e pelas classes (ou funções) "amigas"

# Modificador de Acesso `friend`

- Em C++, o modificador **`friend`** pode ser colocado antes de um método ou (uma classe) declarado dentro de uma classe
  - O método “amigo” (ou classe “amiga”) não pertence a classe na qual está declarado
  - Informa que o método “amigo” (ou classe “amiga”) pode acessar os membros privados e protegidos da classe na qual está declarado

# Classe Conta e Modificador de Acesso friend

```
class Conta {
    private:
       string  numero;
       string agencia;
       double saldo;

    public:
     void creditar (double valor);
     ...
    friend void creditarBonus(Conta* c,double v);
};
```

Parte privada

Função (método) amiga

```
void creditarBonus(Conta* c, double v) {
      if (c->saldo > 10000)
            c->saldo = c->saldo + v;
}
```

Função amiga que não pertence à classe acessa parte privada da classe

# Modificadores de Acesso

- Membros de uma classe, que não recebem modificador de acesso, tem visibilidade **`private`**

- Na redefinição de métodos herdados, o modificador de acesso não deve ser trocado por um mais restrito
  - No entanto, podem ser trocados por modificadores menos restritos

# Classes de Poupanças e Contas: Descrições

```
class PoupancaD {
    private:
       string  numero;
       string agencia;
       double saldo;
    public:
      void creditar (double valor);
      void debitar(double valor);
     void renderJuros(double taxa);
}
```

Partes idênticas das descrições

Parte diferente das descrições

```
class Conta{
  private:
     string  numero;
     string agencia;
     double saldo;
     public:
        void creditar (double valor);
        void debitar(double valor);
}
```

# Classe de Bancos: Assinatura

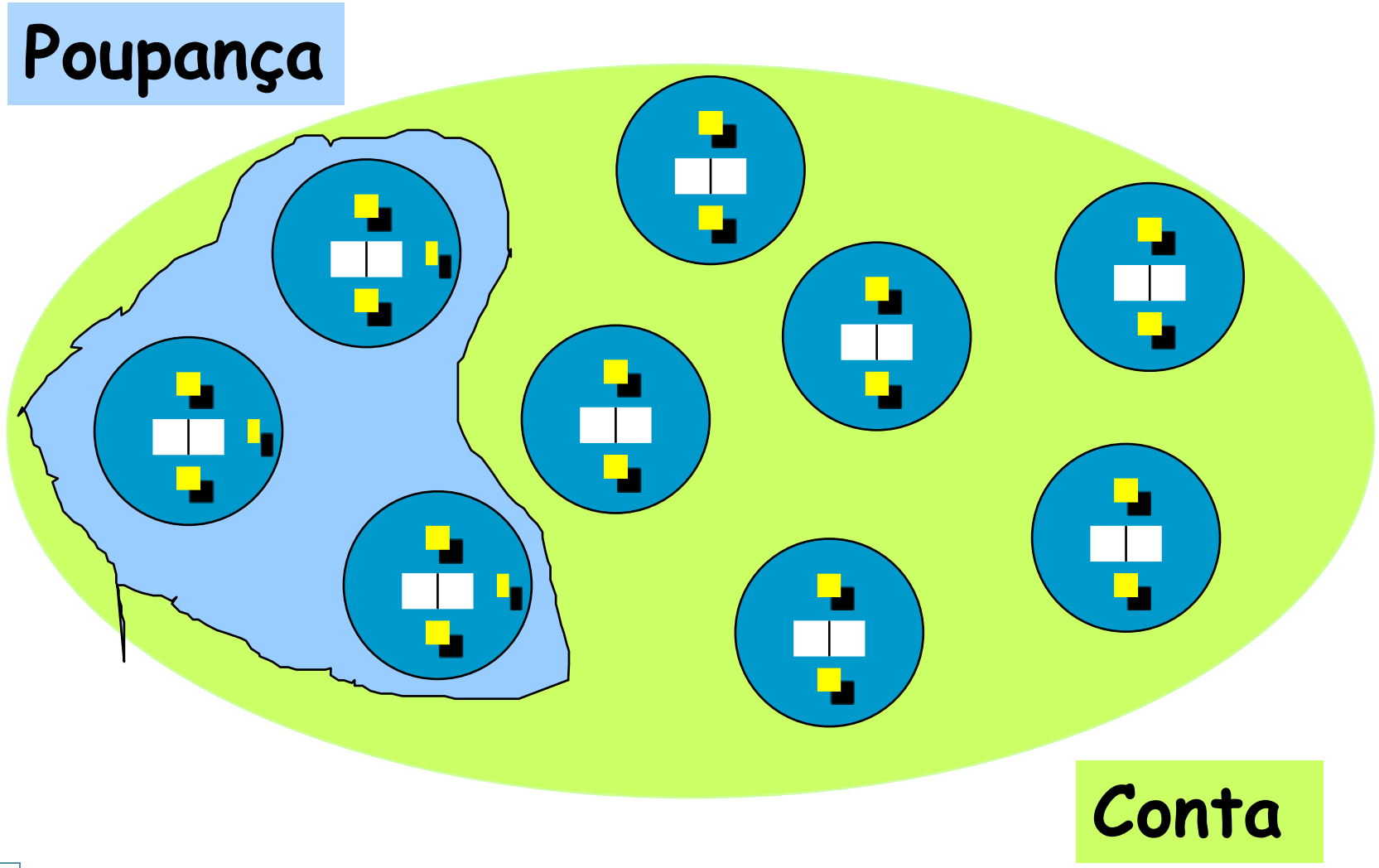
```
class BancoD {

  public:
       void cadastrarConta(Conta* c){}
       void cadastrarPoupanca(PoupancaD* p){}
        ...
}
```

Métodos diferentes para manipular contas e poupanças

# Problemas

- Duplicação desnecessária de código:
  - A definição de PoupançaD é uma simples extensão da definição de Conta
  - Clientes de Conta que precisam trabalhar também com PoupançaD terão que ter código especial para manipular poupanças
- Falta refletir relação entre tipos do "mundo real"

# Subtipos e Subclasses



# Herança

- Necessidade de estender classes
  - Alterar classes já existentes e adicionar propriedades ou comportamentos para representar outra classe de objetos
  - Criar uma hierarquia de classes que “herdem” propriedades e comportamentos de outra classe e definem novas propriedades e comportamentos

# Herança

- ***Herança*** permite que novas classes possam ser derivadas de classes existentes
- A classe existente é chamada de classe pai (mãe) ou superclasse
- A classe derivada é chamada de classe filha ou subclasse
- Subclasse *herda* as características da superclasse
  - Herda os atributos e métodos
- Estabelece a relação de ***é- um***
  - A subclasse é uma versão especializada da superclasse

# Importância de Herança

- Comportamento
  - Objetos da subclasse comportam-se como os objetos da superclasse
- Substituição
  - Objetos da subclasse podem ser usados no lugar de objetos da superclasse
- Reuso de Código
  - Descrição da superclasse pode ser usada para definir a subclasse
- Extensibilidade
  - algumas operações da superclasse podem ser redefinidas na subclasse

# Classe de Poupanças: Assinatura

Indica que a classe Poupanca herda de Conta

```
#include "Conta.h"
class Poupanca: public Conta {
  public:
    void renderJuros(double taxa);
    Poupanca(string num,string ag):Conta(num,ag){}
}
```

Indica que o construtor de Poupanca utiliza o construtor de Conta para inicializar atributos

# Classes de Poupanças: Implementação

```
#include "Poupanca.h"
void Poupanca::renderJuros(double taxa) {
        creditar(getSaldo() * taxa);

}
```

So precisa implementar o método `renderJuros`, o resto é herdado!

# Herança e Construtores

- Construtores não são herdados
  - Embora, freqüentemente, precisamos do construtor da superclasse para a definição dos construtores das subclasses
  - Necessários para inicializar os atributos que são herdados da superclasse
- Devemos portanto definir construtores para as subclasses

# Usando Classe Poupanca

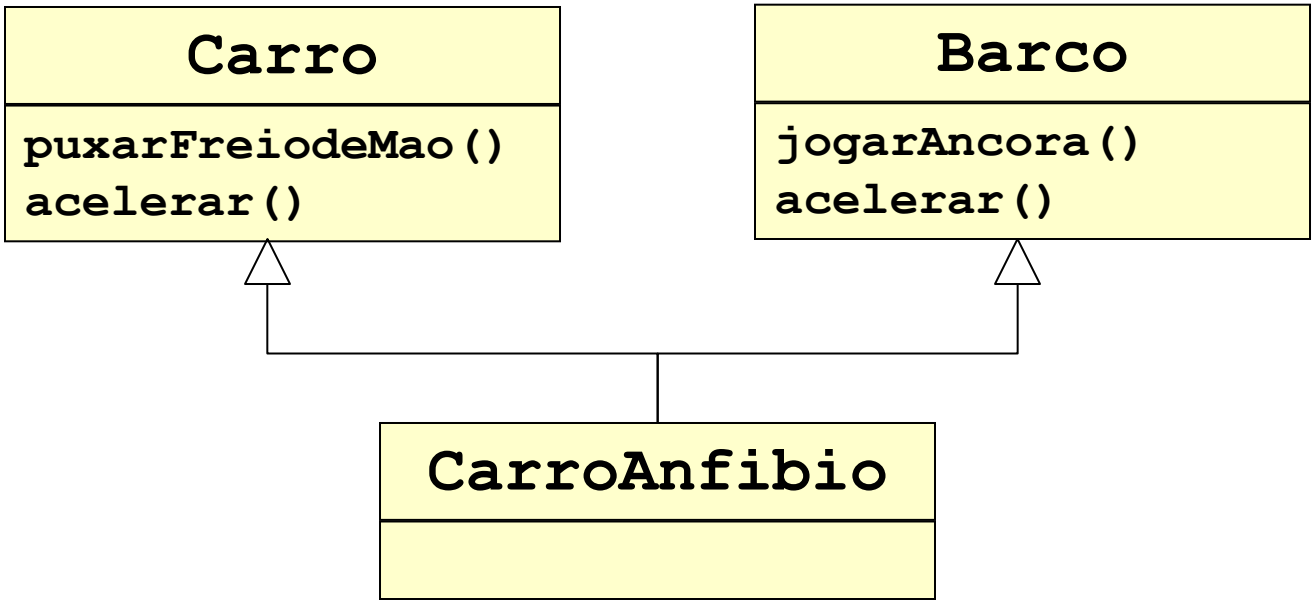
```
#include "Poupanca.h"
using namespace std;
int main() {
   Poupanca*  p;
   p = new Poupanca("21.342-7","123-4");
   p->creditar(500);
   p->debitar(300);

   p->renderJuros(0.01);
   cout << p->getSaldo()<<endl;
}
```

Métodos herdados de Conta

Método existente somente em Poupanca

# Herança Múltipla

- Herança múltipla permite que uma classe seja derivada de mais de uma classe




**Qual versão do método `acelerar` é herdada por `CarroAnfibio`?**

# Herança Múltipla e C++

- Herança múltipla faz com que subclasses herdem atributos e métodos das suas superclasses
  - Pode haver colisões de nomes de atributos e métodos
- Ao contrário de Java, C++ **dá** suporte a herança múltipla
  - Na hora de utilizar o método, usamos o **operador de resolução de escopo** para dizer a qual método estamos nos referindo

# Herança Múltipla em C++

```
class CarroAnfibio: public Carro,Barco {
   ...
};
```

Herda das superclasses Carro e Barco

```
int main() {
      CarroAnfibio* ca;
      ca = new CarroAnfibio();
      ca->Carro::acelerar();
      ca->Barco::acelerar();
      ...
}
```

Usando operador de resolução de escopo para determinar qual versão do método acelerar deve ser usado