Aula 07:

Herança

ECOP13A - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Rev. maio 25

Herança e Classes Derivadas

Da linguagem Simula, C++ emprestou algumas ideias como **classes** e **hierarquia de classes**. Além disso, emprestou também a ideia de que classes devem ser utilizadas para modelar **conceitos** dos universos da aplicação e do programador.

> C++ provê ferramentas que implementam estas visões: A utilização de ferramentas da linguagem para apoiar ideias conceituais é o que distingue o uso efetivo da linguagem C++.

Utilizar construções da linguagem apenas como suporte de notação para tipos tradicionais de programação, é desperdiçar a força do C++.

Herança e Classes Derivadas

Um conceito (ideia, noção, etc.) não existe de forma isolada. Ele coexiste com outros conceitos relacionados e obtém muito de seu poder de seus relacionamentos com outros conceitos.

Por exemplo, tente explicar o que é um carro. Em breve você estará apresentando novos conceitos como volantes, motores, motoristas, caminhões, ambulâncias, estradas, combustível...

Como nós criamos classes para representar conceitos, a questão que emerge é: **como representar relacionamentos entre classes?**

Não podemos representá-los arbitrariamente;
C++ nos apresenta com algumas ferramentas nesse sentido.

Herança e Classes Derivadas

A noção de uma classe derivada e seus mecanismos de linguagem associados são providos para expressar **relacionamentos hierárquicos**, onde há comunalidade entre as classes. Vejamos um exemplo:

> Os conceitos de *círculo* e *triângulo* estão relacionados, pois ambos são *formas geométricas*. Eles têm esta característica comum. Portanto, podemos definir uma classe **Circulo** e outra **Triangulo** com tendo uma classe **Forma** em comum.

Neste caso, **Forma** torna-se uma **classe base** (ou superclasse), e Circulo e Triangulo são **classes derivadas** (ou subclasses).

Superclasses são classes mais gerais, das quais outras classes mais específicas *herdam* os atributos e métodos que realizam um trabalho comum. Como regra, **elas são mais simples** que as **subclasses**, e não possuem superpoderes! Este tipo de nomenclatura é mais comum em linguagens como Java e C#, mas também pode ser utilizada em C++.

Nomenclaturas

SuperClasse	SubClasse
Classe Base	Classe Derivada
Classe Mãe	Classe Filha

Herança e Classes Derivadas

Herança é um relacionamento entre classes em que é criada uma classe (derivada) que **absorve dados e comportamentos** de uma classe existente (base), acrescenta *novas capacidades* e *personaliza* outras. A classe derivada criada continua tendo as características de sua base, mas é mais ***específica***.

> Por isso o nome herança: uma classe **derivada** **herda** os atributos e comportamentos de uma classe **base**.

Em C++ podemos ter tanto **herança simples**, onde uma classe herda de apenas uma única outra, ou **herança múltipla**, onde a classe herda de várias outras classes.

Herança e Classes Derivadas

Distinguimos entre relacionamentos do tipo “**é um**” e “**tem um**”.

> O relacionamento “**é um**” é a **herança**. Neste caso, um objeto de uma classe derivada também pode ser tratado como um objeto de sua classe base, da mesma maneira como um carro também pode ser tratado também como simplesmente veículo (o contrário não é verdade: nem todo veículo é um carro).

O relacionamento “**tem um**” representa **composição**. Neste caso, uma classe tem objetos de outras classes com seus membros de dados. Um carro **tem** um volante, um motor, pedais, etc.

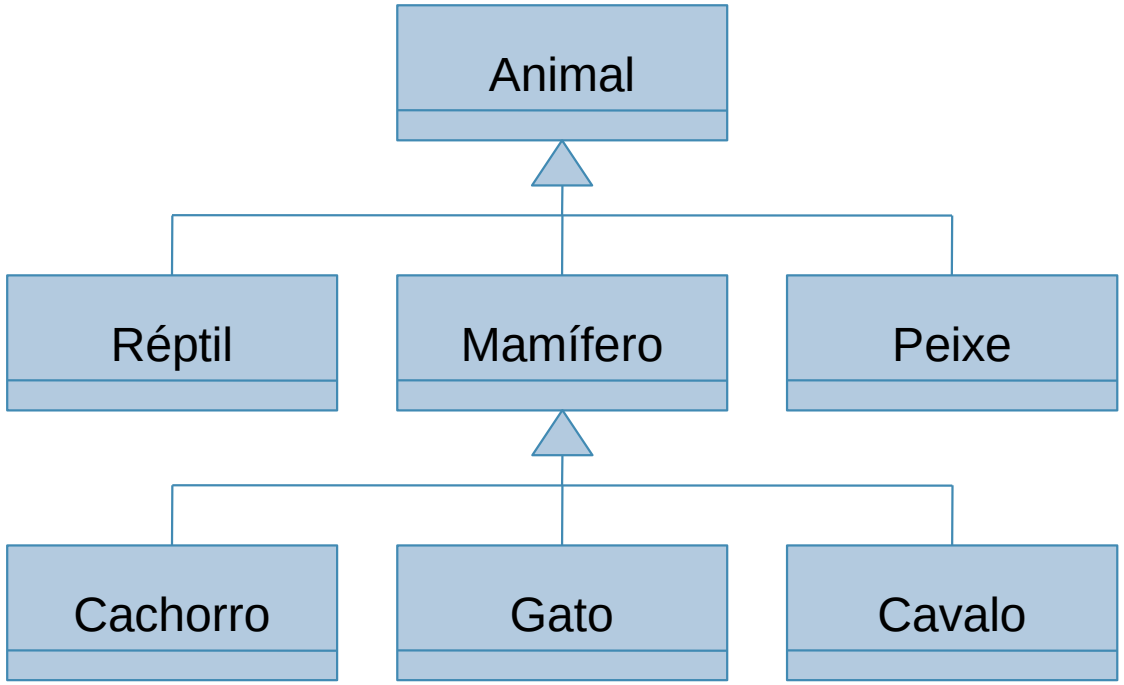
Resumo

Herança é uma forma de **reutilização de software** na qual novas classes são criadas a partir de classes já existentes:

- Absorvendo seus recursos e comportamentos;
- Adicionando ou modificando funcionalidades;
- Tornando-a mais específica para sua finalidade.

Ao identificar a necessidade de uma classe que ***especialize*** outra, não é necessário implementá-la do zero. Ela deve herdar os atributos e métodos da classe mais ***geral***, tomando a outra como ponto de partida.

Veja o exemplo de uma **hierarquia de classes**:



Peixe “**é um**” Animal

Gato “**é um**” Mamífero e também “**é um**” Animal

Podemos portanto dizer que:

- Réptil, Mamífero e Peixe “herdam” de Animal
- Gato, Cachorro e Cavalo “herdam” de Mamífero.

Herança é uma maneira de

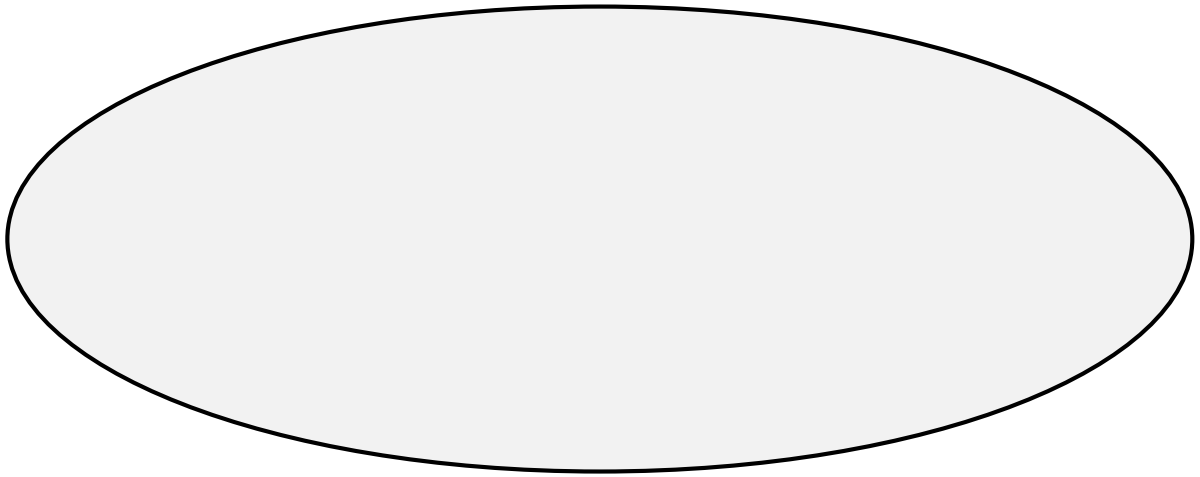
- Agrupar classes similares;
- Modelar similaridades entre classes;
- Organizar melhor a informação e;
- Criar uma taxonomia de objetos.

Taxonomia, ou categorização, tem sido uma importante área de pesquisa em psicologia, na área da cognição humana: a mente humana organiza naturalmente seu conhecimento do mundo em tais sistemas.

Exemplo

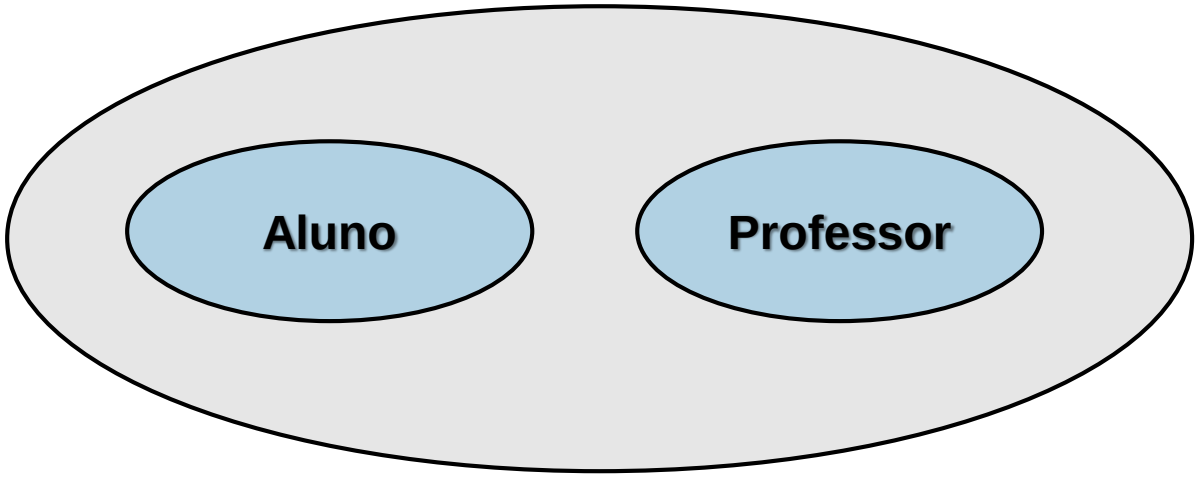
Uma classe define um tipo de dado.

Imagine um sistema que modele as pessoas que fazem parte da nossa universidade, por exemplo para um aplicativo de controle acadêmico.



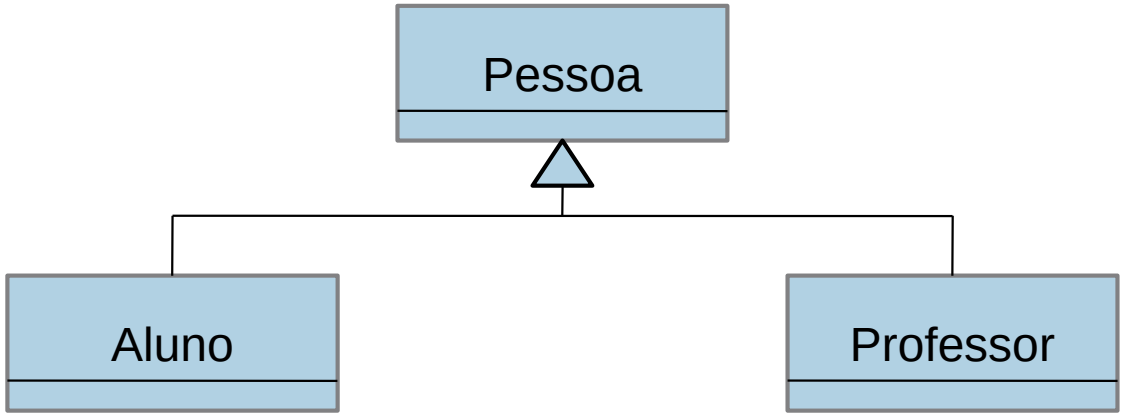
Pessoas na UNIFEI

Algumas pessoas do ambiente universitário podem possuir algumas particularidades que as fazem diferentes umas das outras: *Alunos* e *Professores* "**são**" *Pessoas* mas possuem características que os distinguem, ou seja representam um papel diferente no sistema UNIFEI.



Pessoas na UNIFEI

Em **UML** (*Unified Modeling Language*), este tipo de relacionamento entre classes é chamado de **generalização**. Relaciona um item **geral** a um tipo mais **específico** desse item.



Que características e comportamentos as pessoas na UNIFEI possuem em comum?

- Nome, Endereço
- Mudar Endereço, Imprimir Informações do perfil

- O que há de especial com um Aluno?
  - Sigla do curso, número de matrícula.
  - Mudar de curso.

- O que há de especial com um Professor?
  - SIAPE, categoria, instituto a que pertence.
  - Ser promovido, mudar de instituto.

Uma classe derivada herda os membros de dados e funções membro de sua classe base.
Por exemplo, um **Aluno** terá:

Membros de dados	**Funções membro**
Nome (string)	Mudar Endereço
Endereço (string)	Imprimir Perfil
Sigla do Curso (string)	Mudar de Curso
Número de Matricula (int)	

Enquanto **Professor** terá:

Membros de dados	**Funções membro**
Nome (string)	Mudar Endereço
Endereço (string)	Imprimir Perfil
SIAPE (int)	Ser Promovido
Categoria (int)	Mudar de Instituto
Instituto (string)	

Agora sim! Vamos ver um pouco de código.

```
#ifndef PESSOA_H
#define PESSOA_H

#include <string>
using namespace std;

// classe base para hieraquia
// contém info básica para qualquer PESSOA
class Pessoa {
    // um novo identificador: o que significa?
    protected:
        string nome;
        string endereco;
    public:
        Pessoa(string, string);
        ~Pessoa() {}
        void muda_endereco(string);
        void imprime_perfil();
};

#endif
```

Classe base: **Pessoa**

```cpp
#include "pessoa.h"
#include <iostream>
using namespace std;

// Repare que toda a implementação da classe é normal.
// Qualquer classe pode servir de base para outra.

Pessoa::Pessoa(string n, string e)
    : nome{n}, endereco{e} { }

void Pessoa::muda_endereco(string e)
{
    endereco = e;
}

void Pessoa::imprime_perfil()
{
    cout << "Nome: " << nome << "\n";
    cout << "Endereco: " << endereco;
}
```

Exemplo:

A primeira coisa que percebemos em nosso exemplo, e talvez a única novidade da classe Pessoa é a utilização do novo especificador de acesso **protected**.

> O uso do acesso *protected* provê um nível intermediário de acesso entre *public* e *private*. Membros *protected* de uma classe-base continuam sendo inacessíveis fora da classe (a não ser para *friends*), mas são acessíveis para todas as classes derivadas dela e *friends* das classes derivadas.

Resumindo: membros **protected** de uma classe base são *public* para as classes derivadas e *private* para todo o resto.

Exemplo:

A utilização de membros de dados *protected* melhora ligeiramente o desempenho do programa, pois possibilita às classes derivadas o acesso aos membros sem a necessidade do *overhead* gerado por chamadas de função membro.

No entanto, é recomendável manter os dados como private sempre que possível, encorajando uma engenharia de software apropriada e deixando o trabalho de otimização de código para o compilador.

No exemplo, vocês verão que as classes derivadas não acessarão diretamente nenhum dos membros *protected* da classe base. Não haverá necessidade.

```
#ifndef ALUNO_H
#define ALUNO_H

#include "pessoa.h"

// classe Aluno, derivada de Pessoa
class Aluno : public Pessoa {
    private:
        string curso;
        int matricula;
    public:
        Aluno(string, string, string, int);
        ~Aluno() {}
        void mudar_curso(string);
};

#endif
```

Classe derivada: **Aluno**

Exemplo:

Veja que é muito simples especificar uma classe derivada. Seguimos a sintaxe:

```
class Classe_derivada : public Classe_base { ... };
```

Mas o que significa o **public**?

A herança do tipo *public* é a mais utilizada. Com ela, todos os membros da classe base mantêm seu acesso na classe derivada, ou seja, membros *public* continuam *public* e membros *protected* continuam *protected* (lembre-se de que os membros *private* continuam **inacessíveis**).

Vejamos como ficaria o acesso para herança ***private*** e ***protected***.

Exemplo:

Vejamos como ficaria o acesso para herança private e protected.

	Herança protected	**Herança private**
Membros **public**	São protected na classe derivada.	Tornam-se private na classe derivada.
Membros **protected**	Continuam protected na classe derivada.	Também tornam-se private na derivada
Membros **private**	Oculto na classe derivada.	Oculto na classe derivada.

Utilizaremos como “padrão” a herança pública, que é mais comum e que modela o relacionamento de maneira mais clara, possibilitando a criação de hierarquias.

Exemplo:

Entenda melhor o que quer dizer reutilização de código:

```
#ifndef ALUNO_H
#define ALUNO_H

#include "pessoa.h"

// classe Aluno, derivada de Pessoa
class Aluno : public Pessoa {
    private:
        string curso;
        int matricula;
    public:
        Aluno(string, string, string, int);
        ~Aluno() {}
        void mudar_curso(string);
};

#endif
```

c/ herança

```
#ifndef ALUNO_H
#define ALUNO_H

// classe Aluno
class Aluno {
    private:
        string nome;
        string endereco;
        string curso;
        int matricula;
    public:
        Aluno(string, string, string, int);
        ~Aluno() {}
        void mudar_curso(string);
        void muda_endereco(string);
        void imprime_perfil();
};

#endif
```

s/ herança

Não peca a chance de modelar relacionamentos óbvios de herança. Economiza-se código e faz-se boa engenharia de software.

Exemplo:

Agora, a implementação da classe Aluno, derivada de Pessoa:

```
#include "aluno.h"

// Podemos chamar o construtor da classe base
// para inicializar os membros herdados.
// Por isso, Pessoa{n,e} já inicializa a parte
// de Aluno que é simplesmente uma Pessoa.
Aluno::Aluno(string n, string e, string c, int m)
    : Pessoa{n,e}, curso{c}, matricula{m} {}

// Somente implementamos as funções membro da classe Aluno
void Aluno::mudar_curso(string c)
{
    curso = c;
}
```

```
#ifndef PROFESSOR_H
#define PROFESSOR_H

#include "pessoa.h"

class Professor : public Pessoa {
    private:
        int siape;
        int categoria;
        string instituto;
    public:
        Professor(string, string, int, int, string);
        ~Professor() {}
        int promover() { return ++categoria; } // inline
        void mudar_instituto();
        void imprime_perfil(); // opa, uma redefinição!
};

#endif
```

Classe derivada: **Professor**

Exemplo:

A classe Professor segue o mesmo padrão de Aluno, mas acrescenta suas próprias particularidades a seu tipo base Pessoa.

> Entretanto, para esta classe derivada, o programador decidiu **redefinir** o método de impressão de perfil. Neste caso, ao ser chamada de uma referência do tipo Professor, a função imprime_perfil() irá executar a nova versão ao invés da presente na classe Pessoa.

Neste caso, o programador pode escolher se a nova função faz algo totalmente diferente ou simplesmente complementa a função da classe base, como faremos neste exemplo:

```cpp
#include "professor.h"
#include <iostream>
using namespace std;

// Construtor da classe Professor, derivada de Pessoa
Professor::Professor(string n, string e, int s, int c, string i)
    : Pessoa{n, e}, siape{s}, categoria{c}, instituto{i} {}

void Professor::imprime_perfil()
{
    // Chama a função imprime_perfil() da classe base
    // que desempenha seu papel normalmente, imprimindo
    // os valores para nome e endereco.
    Pessoa::imprime_perfil();

    // complementa a impressao com valores da classe derivada
    cout << "\nsiape: " << siape;
    cout << "\ncategoria: " << categoria;
    cout << "\ninstituto: " << instituto;
}
```

Utilize o operador de escopo para chamar métodos da classe base, que sejam redefinidos na derivada. Assim, sabemos que é a função *imprime_perfil* de Pessoa.

```cpp
#include <iostream>
#include "aluno.h"
#include "professor.h"

int main()
{
    Pessoa p{"Joao", "Itajuba"};
    Aluno a{"Paulo", "Campinas", "ECO", 11984};
    Professor pf{"Pedro", "SJC", 132515, 2, "IESTI"};

    cout << "Pessoa: \n";
    p.imprime_perfil();
    cout << "\n\nAluno: \n";
    a.imprime_perfil();
    cout << "\n\nProfessor: \n";
    pf.imprime_perfil();

    Pessoa p2 = pf; // Pode isso? SIM. Mas não o contrário.
    cout << "\n\nSegunda Pessoa: \n";
    p2.imprime_perfil();
}
```

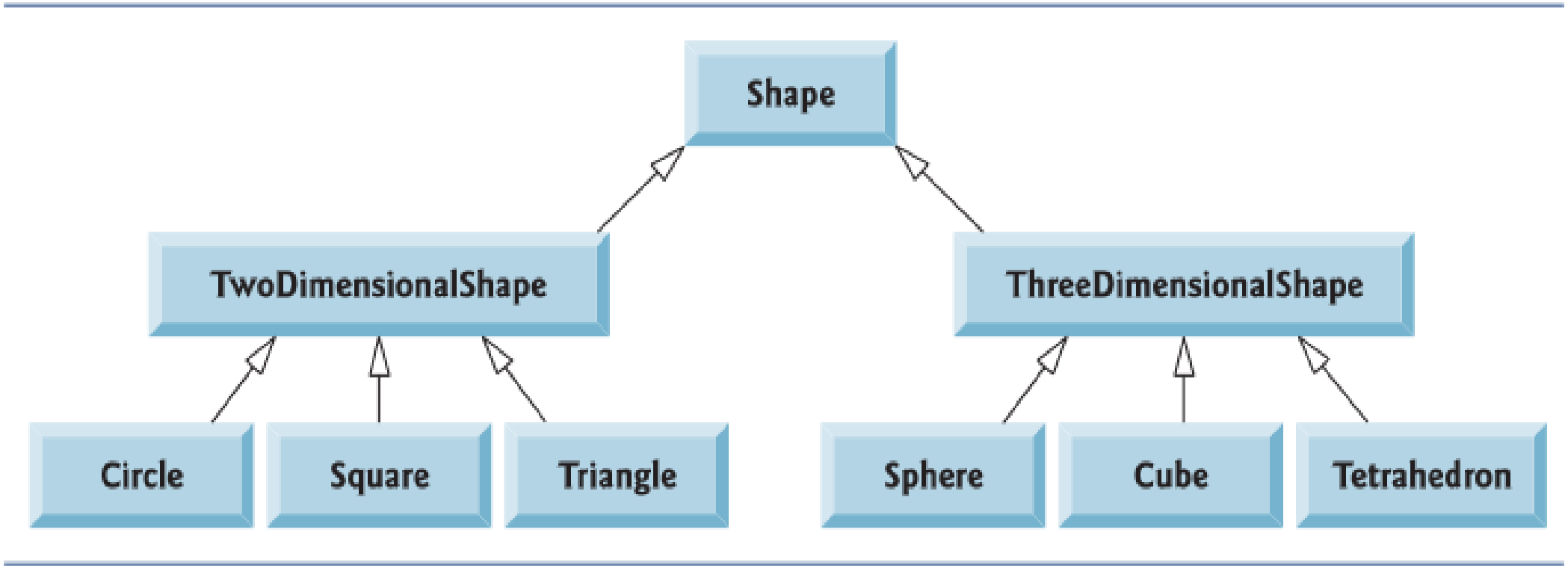
Teste na função **main**

```
C:\Users\JoãoPaulo\Desktop\Aulas\ECOP03\7 - Herança\code>principal
Pessoa:
Nome: Joao
Endereco: Itajuba

Aluno:
Nome: Paulo
Endereco: Campinas

Professor:
Nome: Pedro
Endereco: SJC
siape: 132515
categoria: 2
instituto: IESTI

Segunda Pessoa:
Nome: Pedro
Endereco: SJC
```

Repare que cada objeto foi impresso com seus próprios dados.

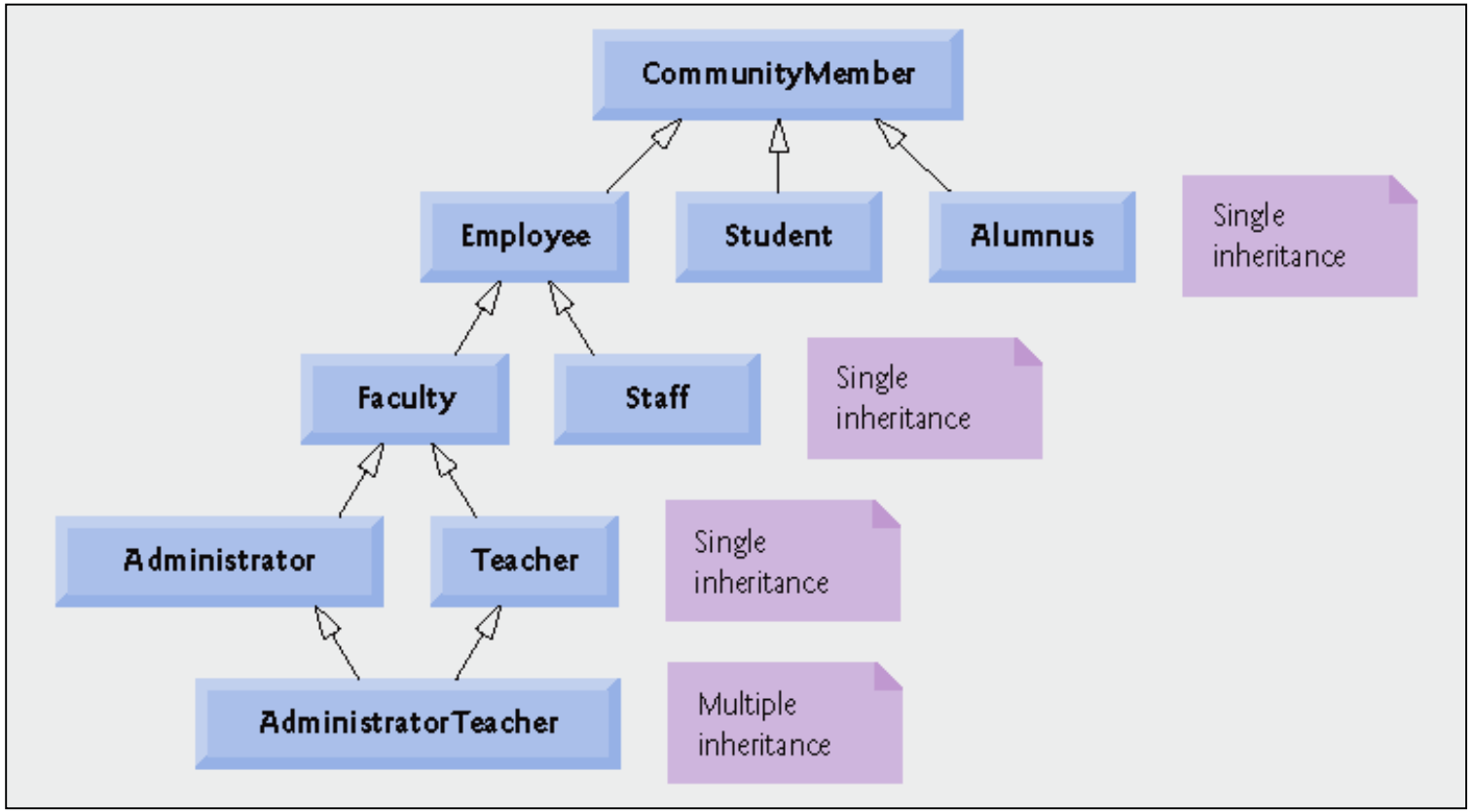
Como é possível referenciar Alunos e Professores através de uma variável do tipo Pessoa, a função membro chamada será a do tipo específico da variável. Por isso os dados da segunda pessoa, que é um professor, foram impressos apenas como o de uma Pessoa.

Hierarquia Formas



Herança Múltipla

A linguagem C++ provê suporte para um tipo de herança que não é comum em muitas outras linguagens. Nela, é possível fazer com que uma determinada classe herde membros de várias outras, derivando não somente de uma classe base, mas de várias. Veja um exemplo retirado do livro "C++, como programar" (Deitel, 2011) :



Herança Múltipla

Criemos um tipo em nossa hierarquia pessoa-Aluno-Professor que é um professor que também é aluno de algum curso:

```
class ProfessorAluno : public Professor, public Aluno {
    private:
        bool esta_ativo;
    public:
        ProfessorAluno(string n, string e, int m, int s, int cat, string c, string i)
            : Professor{n, e, s, cat, i}, Aluno{n, e, c, m}
        {
            esta_ativo = true;
        }

        void imprime_perfil()
        {
            // é necessário implementar, para que não haja ambiguidade,
            // dado que tanto Aluno quanto Professor possuem funções com mesmo nome
            Professor::imprime_perfil();
        }
};
```

Lembre-se de que **ProfessorAluno** herda de duas classes que estão relacionadas pela classe Pessoa. **Não precisa ser assim**. Poderíamos ter **ProfessorPesquisador**, que herda de uma classe Pesquisador que nada tem a ver com nossa hierarquia anterior

Referências

- https://cplusplus.com/reference/

- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.