Herança

BCC 221 - Programação Orientada a Objectos(POO)

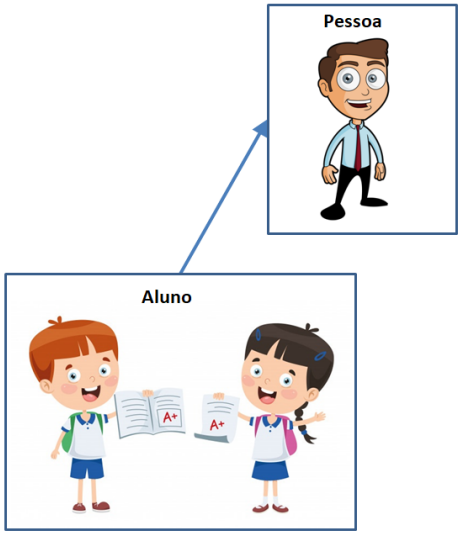
Guillermo Cámara-Chávez

Departamento de Computação - UFOP

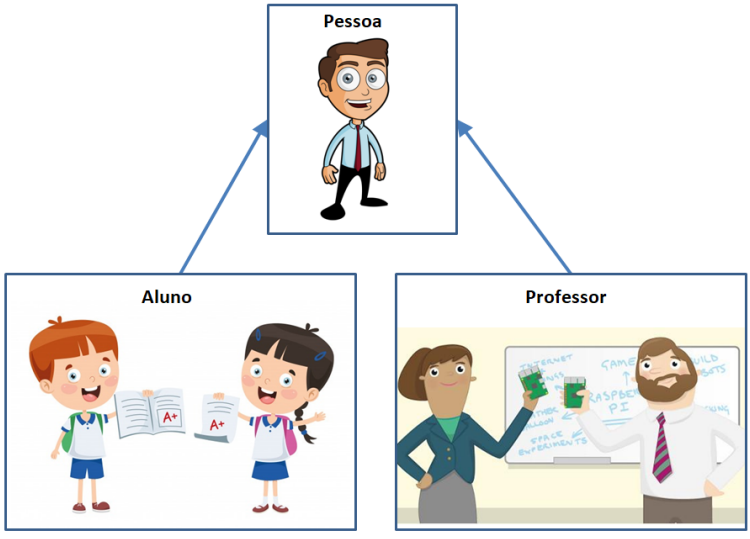
Introdução I

Introdução II

Introdução III



Introdução IV



Introdução V

- A herança é uma forma de **reuso de software**
  - O programador cria uma classe que **absorve os dados e o comportamento** de uma classe existente;
  - Ainda é possível **aprimorá-la com novas capacidades**.

Introdução VI

- Além de **reduzir o tempo de desenvolvimento**, o reuso de software **aumenta** a probabilidade de **eficiência** de um software
- Componentes já debugados e de qualidade provada contribuem para isto.

Introdução VII

- Uma **classe existente** e que será absorvida é chamada de **classe base** (ou **superclasse**);
- A **nova classe**, que absorve, é chamada de **classe derivada** (ou **subclasse**)
  - Considerada uma **versão especializada** da classe base.

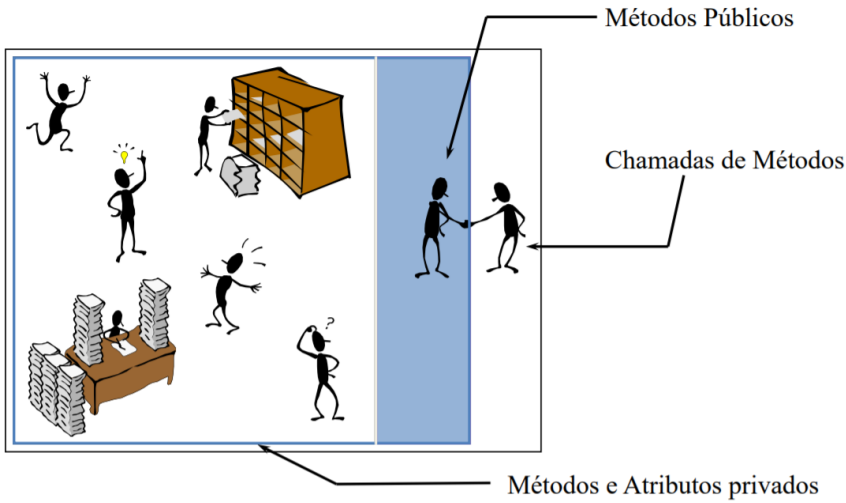
Introdução VIII

- É importante **identificar a diferença** entre **composição** e **herança**:
  - Na **herança**, um objeto da subclasse **"é um"** objeto da superclasse. Por exemplo, o carro **é um** veículo;
  - Enquanto que na **composição** um objeto **"tem um"** outro objeto. Por exemplo, o carro **tem uma** direção.

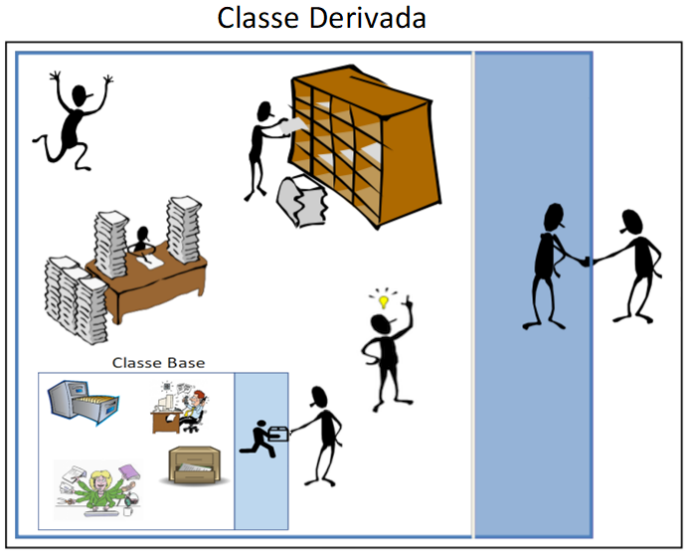
Herança I

- **Herança** é o **mecanismo** pelo qual **elementos mais específicos incorporam** a **estrutura e comportamento** de elementos **mais gerais**.
- Uma **classe derivada herda** a estrutura de **atributos e métodos** de sua **classe "base"**, mas pode seletivamente:
  - **adicionar** novos **métodos**
  - **estender** a estrutura de **dados**
  - **redefinir a implementação de métodos** já existentes

Herança II


Métodos Públicos
Chamadas de Métodos
Métodos e Atributos privados

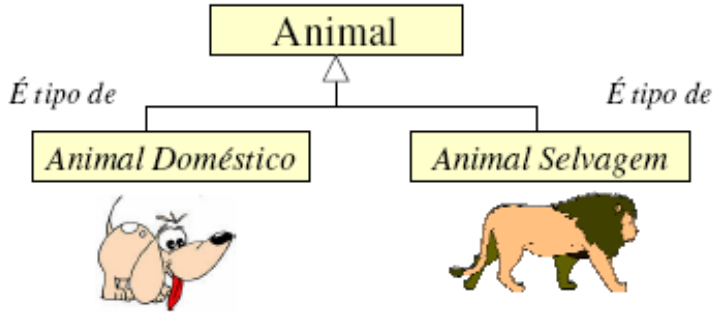
Herança III



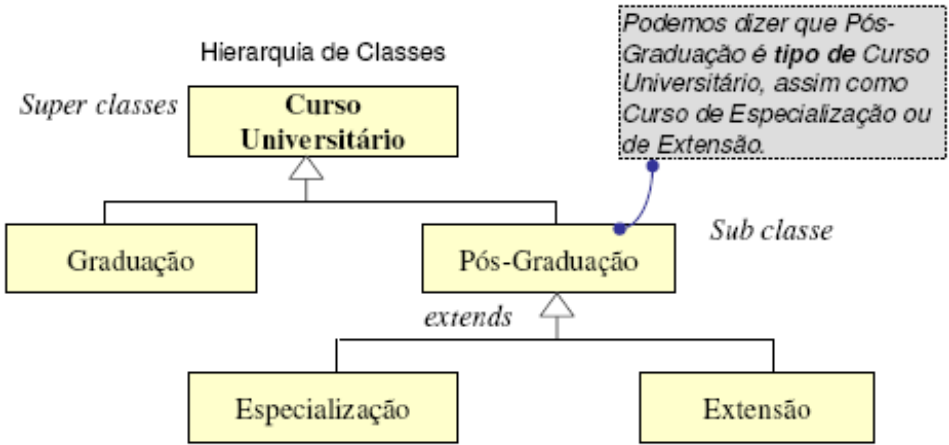
Herança IV

- Uma classe "pai" ou super classe proporciona a **funcionalidade que é comum a todas** as suas classes derivadas, filhas ou sub classe,

- Uma classe derivada proporciona a **funcionalidade adicional que especializa seu comportamento**.



Herança V


Hierarquia de Classes
Super classes
Curso Universitário
Podemos dizer que Pós-Graduação é tipo de Curso Universitário, assim como Curso de Especialização ou de Extensão.
Graduação
Pós-Graduação
Sub classe
extends
Especialização
Extensão

Herança VI

- Um **possível problema** com herança é ter que **herdar atributos ou métodos desnecessários** ou inapropriados
- É **responsabilidade do projetista** determinar se as **características da classe base** são **apropriadas** para herança direta e também para futuras classes derivadas.

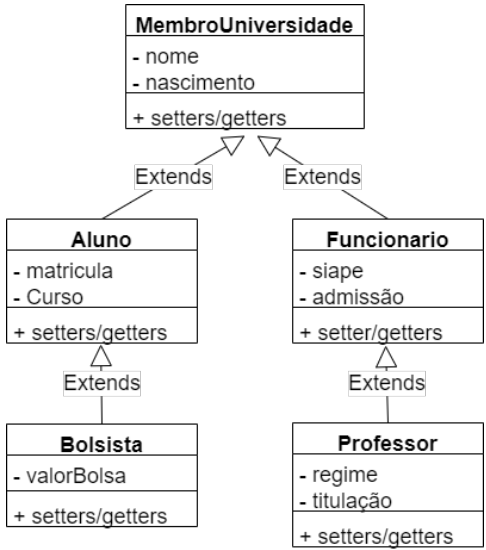
Herança VII

- Ainda, **é possível** que **métodos necessários não se comportem** de maneira especificamente necessária
- Nestes casos, é possível que a **classe derivada redefina** o método para que este tenha uma implementação específica.
- Um **objeto** de uma **classe derivada** pode ser **tratado** como um objeto da **classe base**.

Herança VIII

- Os métodos de uma classe derivada podem necessitar acesso aos métodos e atributos da classe base
  - **Somente** os **membros não privados** estão **disponíveis**;
  - **Membros** que **não** devem ser **acessíveis** através de herança devem ser **privados**;
  - Atributos privados poderão ser acessíveis por **getters** e **setters** públicos

Herança IX


MembroUniversidade
- nome
- nascimento
+ setters/getters
Extends
Extends
Aluno
- matricula
- Curso
+ setters/getters
Funcionario
- siape
- admissão
+ setter/getters
Extends
Extends
Bolsista
- valorBolsa
+ setters/getters
Professor
- regime
- titulação
+ setters/getters

Sintaxe em C++ I

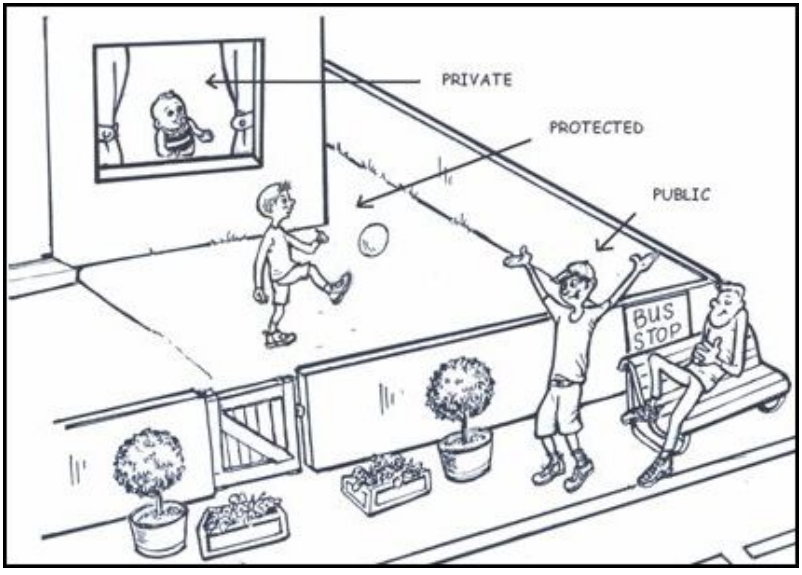
```
class classe-derivada : acesso classe-base
{
    corpo da nova clase
}
```

- O operador acesso é opcional, mas claro se presente tem de ser *public*, *private* ou *protected*.

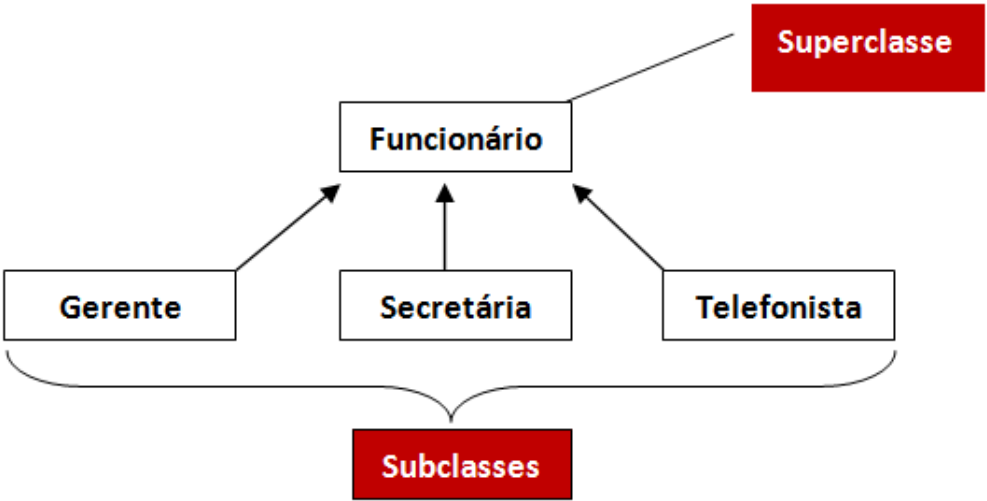
Sintaxe em C++ II

- Se o acesso for *public*
  - membros *public* da classe base: é como fazer copias dos métodos e colocássemos como *public* na classe derivada
  - membros *private*: não são passados, acessível apenas pelo métodos da classe base e funções amigas
  - membros *protected*: são "copiados" como *protected* na classe derivada
- **Funções amigas** (*friend*) **não são herdadas**.

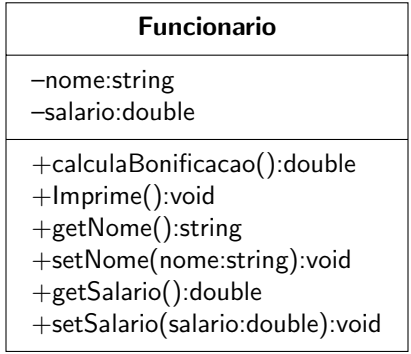
Sintaxe em C++ III



Exemplo I


Superclasse
Funcionário
Gerente
Secretária
Telefonista
Subclasses

Exemplo II



Os funcionários recebem uma bonificação do 10 % do salario. Implementar o imprime() usando sobrecarga do operados << .

Funcionário.h I

```
#include<string>
#include<iostream>
#include<iomanip>
using namespace std;
class Funcionario{
    string nome;
    double salario;
public:
    Funcionario(string="", double=0.0);
    ~Funcionario();
    string getNome() const;
    void setNome(string);
    double getSalario() const;
    void setSalario(double);
    double calculaBonificacao();
    friend ostream& operator<<(ostream&, const Funcionario&);
};
```

Funcionário.cpp I

```
#include "Funcionario.h"
Funcionario::Funcionario(string nome, double salario) :
        nome(nome), salario(salario){}

Funcionario::~Funcionario(){}

string Funcionario::getNome() const {
    return nome;
}
void Funcionario::setNome(string nome) {
    this->nome = nome;
}
double Funcionario::getSalario() const {
    return salario;
}
```

Funcionário.cpp II

```
void Funcionario::setSalario(double salario) {
    if (salario > 0)
        this->salario = salario;
    else
        this->salario = 0.0;
}
double Funcionario::calculaBonificacao() {
    return getSalario() * 0.1;
}
ostream& operator<<(ostream& out, const Funcionario& funcionario) {
    out << "\n nome: " << funcionario.nome
        << "\n salario: " << fixed
        << setprecision(2) << funcionario.salario;
    return out;
}
```

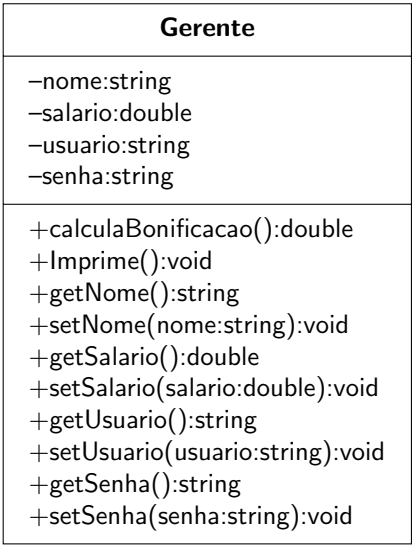
Main.cpp I

```
int main()
{
        Funcionario func1("Gustavo Costa", 12456.00);
        Funcionario func2("Manoel Travis", 10000.00);
        cout << func1 << func2;
        return 0;
}
```

Main.cpp II

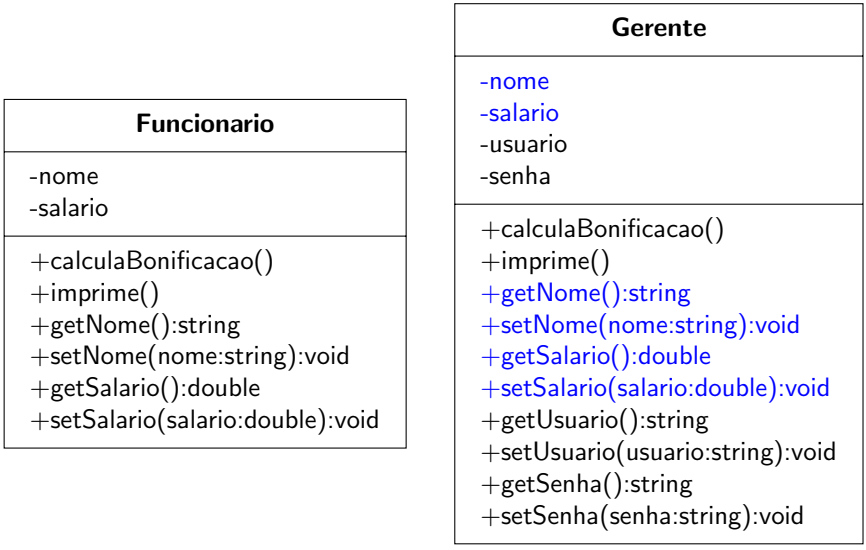
```
nome: Gustavo Costa
salario: 12456.00
nome: Manoel Travis
salario: 10000.00
```

Exemplo Gerente I



A bonificação do Gerente é igual ao 60 % do salario mais 100 reais.

Exemplo Gerente II



Exemplo Gerente III

- Ambas as classes compartilham vários dos atributos (privados);
- *getters* e *setters* também são compartilhados
- Quando a **redundância** entre **classes** acontece, **caracteriza-se** a necessidade de **herança**.

Exemplo Gerente IV

- A replicação de código pode resultar em replicação de erros:
- A **manutenção é dificultada**, pois cada cópia tem que ser corrigida;
- Define-se uma classe que absorverá os atributos e métodos redundantes
  - A classe base;
  - A manutenção dada na classe base se reflete nas classes derivadas automaticamente.

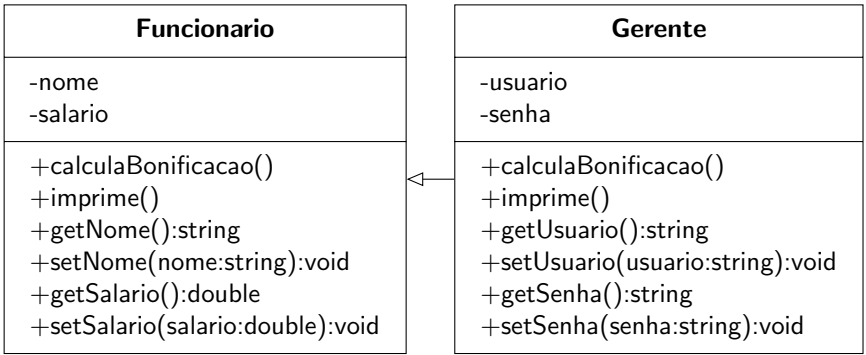
Exemplo Gerente V

- O próximo exemplo fixa a classe *Funcionario* como classe base
- Será utilizada a mesma definição desta classe do exemplo anterior;
- Os atributos continuam privados.
- A classe *Gerente* será a classe derivada

Exemplo Gerente VI

- Acrescentam-se o atributo *usuário* e *senha*;
- Acrescenta-se também *getters* e *setters* e redefinirá dos métodos.
- Qualquer tentativa de acesso aos membros privados da classe base gerará erro de compilação.

Classe Gerente I



Gerente.h I

```
#include"Funcionario.h"
class Gerente : public Funcionario
{
    string usuario;
    string senha;
public:
    Gerente(string="", double=0.0, string="", string="");
    ~Gerente();
    string getUsuario() const;
    void setUsuario(string);
    string getSenha() const;
    void setSenha(string);
    double calculaBonificacao();
    friend ostream& operator<<(ostream&, const Gerente&);
};
```

Gerente.h II

- O operador **:** define a herança;
- A herança é pública
- Todos os métodos públicos da classe base são também métodos públicos da classe derivada
  - Embora não os vejamos na definição da classe derivada, eles fazem parte dela.
- Foi definido um construtor específico.
- É necessário incluir o *header* da classe a ser herdada

Gerente.cpp I

```
#include "Gerente.h"

Gerente::Gerente(string nome, double salario,
    string usuario, string senha) :
    Funcionario(nome, salario), usuario(usuario), senha(senha) {}
Gerente::~Gerente(){}
string Gerente::getUsuario() const {
    return usuario;
}
void Gerente::setUsuario(string usuario) {
    this->usuario = usuario;
}
string Gerente::getSenha() const {
    return senha;
}
void Gerente::setSenha(string senha) {
    this->senha = senha;
}
```

Gerente.cpp II

```
double Gerente::calculaBonificacao() {
    return getSalario()*0.6 + 100;
}
ostream& operator<<(ostream& out, const Gerente& gerente) {
    out << "\n nome: " << gerente.getNome()
        << "\n salario: " << gerente.getSalario()
        << "\n usuario: " << gerente.usuario;
    return out;
}
```

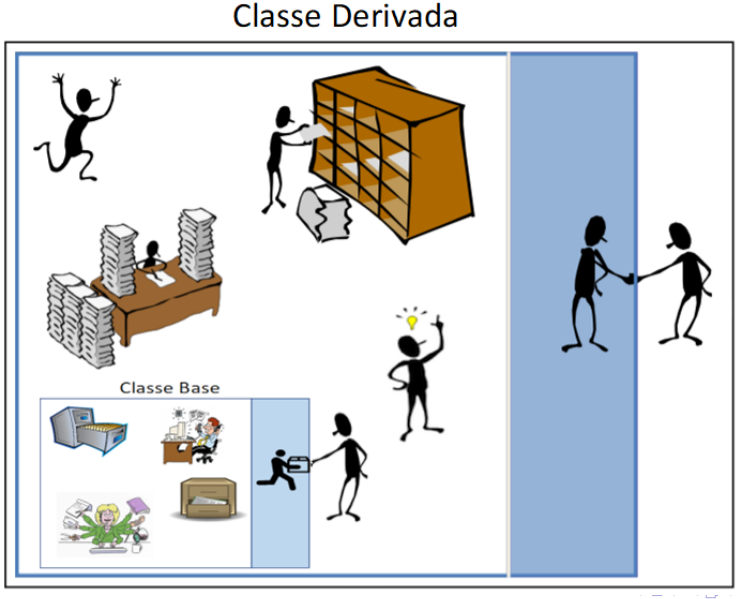
Gerente.cpp III

- O construtor da classe derivada chama explicitamente o construtor da classe base
- É necessário que a classe derivada tenha um construtor para que o construtor da classe base seja chamado;
- Se o construtor da classe base não for chamado explicitamente, o compilador chamará implicitamente o construtor *default* (sem argumentos) da classe base
- Se este não existir, ocorrerá um erro de compilação.

Gerente.cpp IV

- Acessar diretamente membros privados da classe base, não é permitido
- Utilizar os *getters* associados a tais atributos para evitar os erros de compilação
  - *getters* são públicos
- Os métodos podem ser redefinidos

Gerente.cpp V



Main.cpp I

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

int main()
{
    Gerente gerente("Sandra Sion", 30000.00, "ssion", "senh2");
    cout << gerente;
    cout << "\n Bonificacao: "
         << gerente.calculaBonificacao();
    return 0;
}
```

Main.cpp II

```
nome: Sandra Sion
salario: 30000.00
usuario: ssion
Bonificacao: 18010.00
```

Dados protegidos: protected I

```
class Funcionario{
    double salario;
    ...
};
class Gerente : public Funcionario{
    ...
    void calculaBonificacao(){
        return getSalario() * 0.6 + 100;
    }
};
```

Dados protegidos: protected II

```
class Funcionario{
protected:
    double salario;
    ...
};
class Gerente : public Funcionario{
    ...
    void calculaBonificacao(){
        return salario * 0.6 + 100;
    }
};
```

Dados protegidos: protected III

- Utilizar atributos protegidos nos dá uma leve melhoria de desempenho
- Não é necessário ficar chamando funções (*getters* e *setters*).

Dados protegidos: protected IV

- Por outro lado, também nos **gera dois problemas** essenciais:
  - Uma vez que não se usa getter e setter, pode haver inconsistência nos valores do atributo
  - Nada garante que o programador que herdar fará uma validação.
- Uma classe derivada deve depender do serviço prestado, e não da implementação da classe base.

Dados protegidos: protected V

- Quando devemos usar *protected* então?
  - Quando a classe base precisa fornecer um serviço (método) apenas para classes derivadas e funções amigas, ninguém mais.

Dados protegidos: protected VI

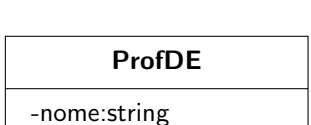
- **Declarar membros** como **privados** permite que a **implementação** da **classe base** seja **alterada sem implicar em alteração** da implementação da **classe derivada**;
- O **padrão mais utilizado** é **atributos privados** e *getters* e *setters* **públicos**.

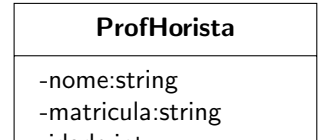
Exercício I

- Os professores de uma universidade dividem-se em 2 categorias
  - professores em dedicação exclusiva (DE)
  - professores horistas
- Professores em dedicação exclusiva possuem um salário fixo para 40 horas de atividade semanais.
- Professores horistas recebem um valor estipulado por hora.

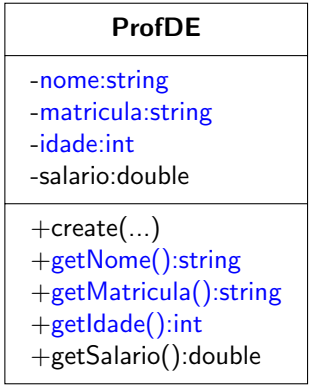
Exercício II

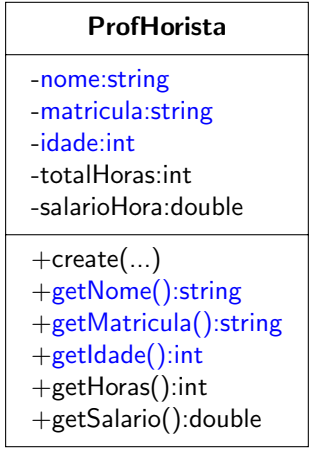
- Problema pode ser resolvido através da seguinte modelagem de classes:





Exercício III





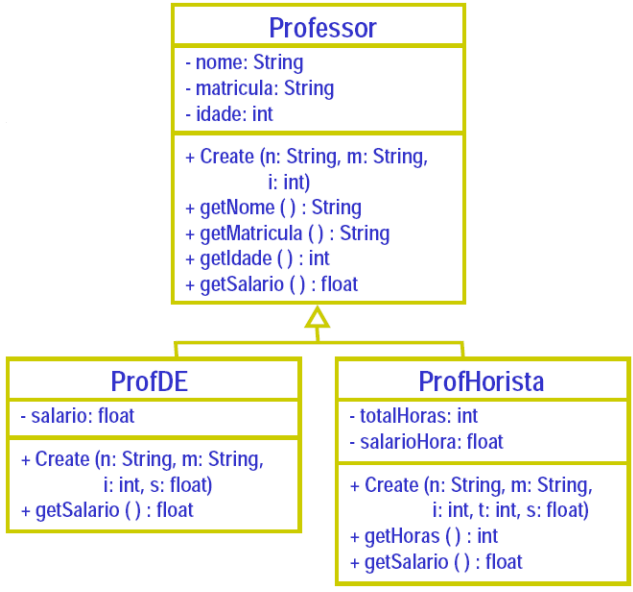
Exercício IV

- Analisando a solução:
  - Alguns atributos e métodos são iguais
  - Como resolver? **Herança!**
- Sendo assim:
  - Cria-se uma classe *Professor*, que contém os atributos e métodos comuns aos dois tipos de professor:
  - A partir dela, cria-se duas novas classes, que representarão os professores horistas e DE.
  - Para isso, essas classes deverão "**herdar**" os atributos e métodos declarados pela classe "pai", *Professor*.

Exercício V

- Solução

Exercício VI



FIM