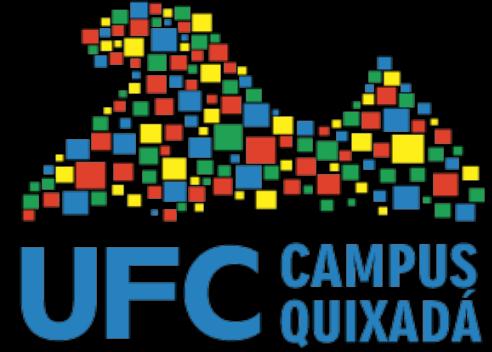




UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ



# Programação Orientada a Objetos

Prof. Thiago Werlley

2021.2



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS DE QUIXADÁ

# Na aula passada

- Classes
- Objetos
- Métodos
  - Construtores
  - Destrutores
  - Construtores Parametrizados e Vetores de Objetos
  - Objetos como Parâmetros de Métodos
  - Métodos que Retornam Objetos
- Separando a Interface da Implementação
- Composição: Objetos como Membros de Classes
- Funções Amigas
- Sobrecarga de Operadores
- O Ponteiro *This*

# Na aula de hoje

- Herança
  - Compilação
  - Redefinição de Métodos
  - Construtores e Destrutores em Classes Derivadas
- Herança Pública vs. Privada vs. Protegida
- Conversões de Tipo entre Base e Derivada
- Herança Múltipla
  - Classes Base Virtuais
  - Compilação
  - Construtores em Herança Múltipla

# Herança

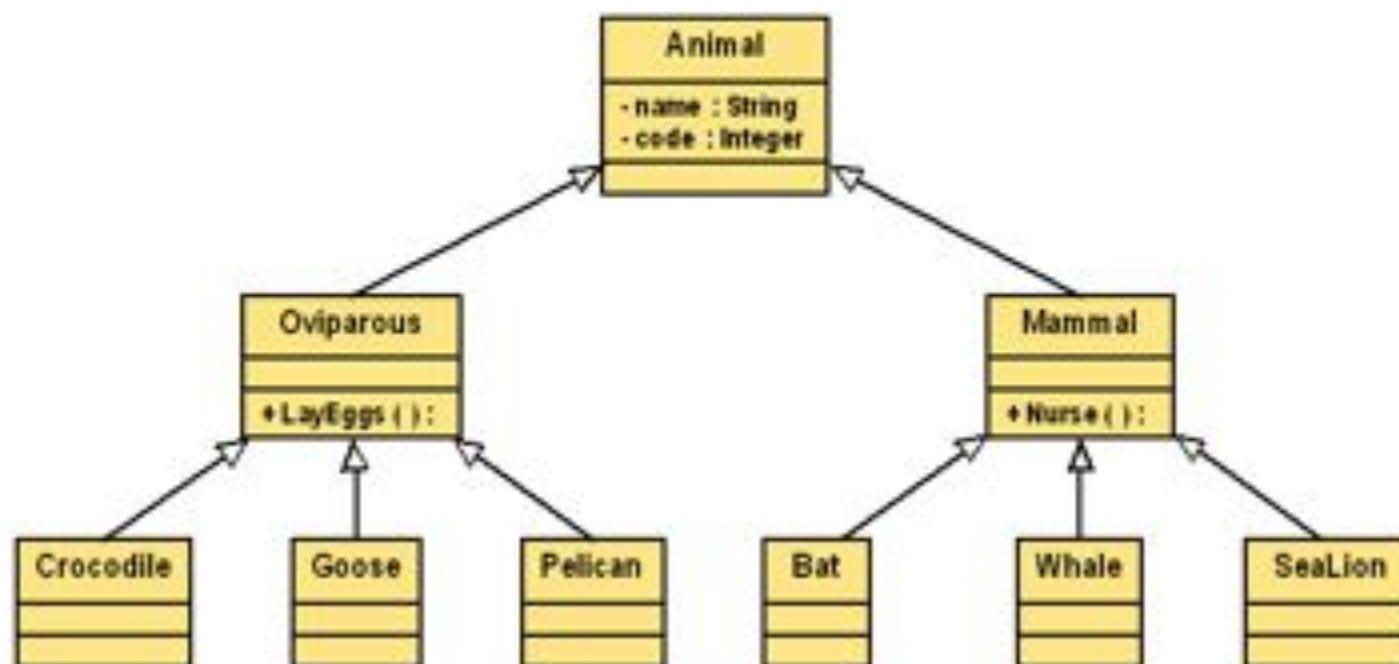
# Herança

- 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.
- 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.

# Herança

- 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**)
  - Que é considerada uma versão **especializada** da classe base.
- Uma **classe base direta** é herdada diretamente da classe derivada
  - É possível que haja uma **classe base indireta**, em que haja um ou dois níveis de distância em relação à classe derivada.
- A relação entre tais classes define uma **hierarquia de classes**.

# Herança



# Herança

- A herança define um relacionamento “é um”
  - Um carro é um veículo
    - Todas as propriedades de um veículo são propriedades de um carro.
  - Um objeto de uma classe derivada pode ser tratado como um objeto da classe base.
- Os métodos de uma classe derivada podem necessitar acesso aos métodos e atributos da classe base
  - Somente os membros não privados estão disponíveis;
  - Ou seja, membros que não devem ser acessíveis através de herança devem ser **privados**;
    - Poderão ser acessíveis por *getters* e *setters* públicos, por exemplo.

# Herança

- 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.
- 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.

# Exemplo 1

- Vamos criar uma hierarquia de classes para descrever a comunidade acadêmica:
  - A **base** será descrita por uma classe *CommunityMember*;
- As primeiras formas de especialização serão
  - Empregados (*Employee*);
  - Estudantes (*Students*);
  - Ex-Alunos (*Alumnus*);
- Entre os empregados, temos:
  - Os de departamentos ou escolas (*Faculty*);
  - Funcionários de apoio (*Staff*).

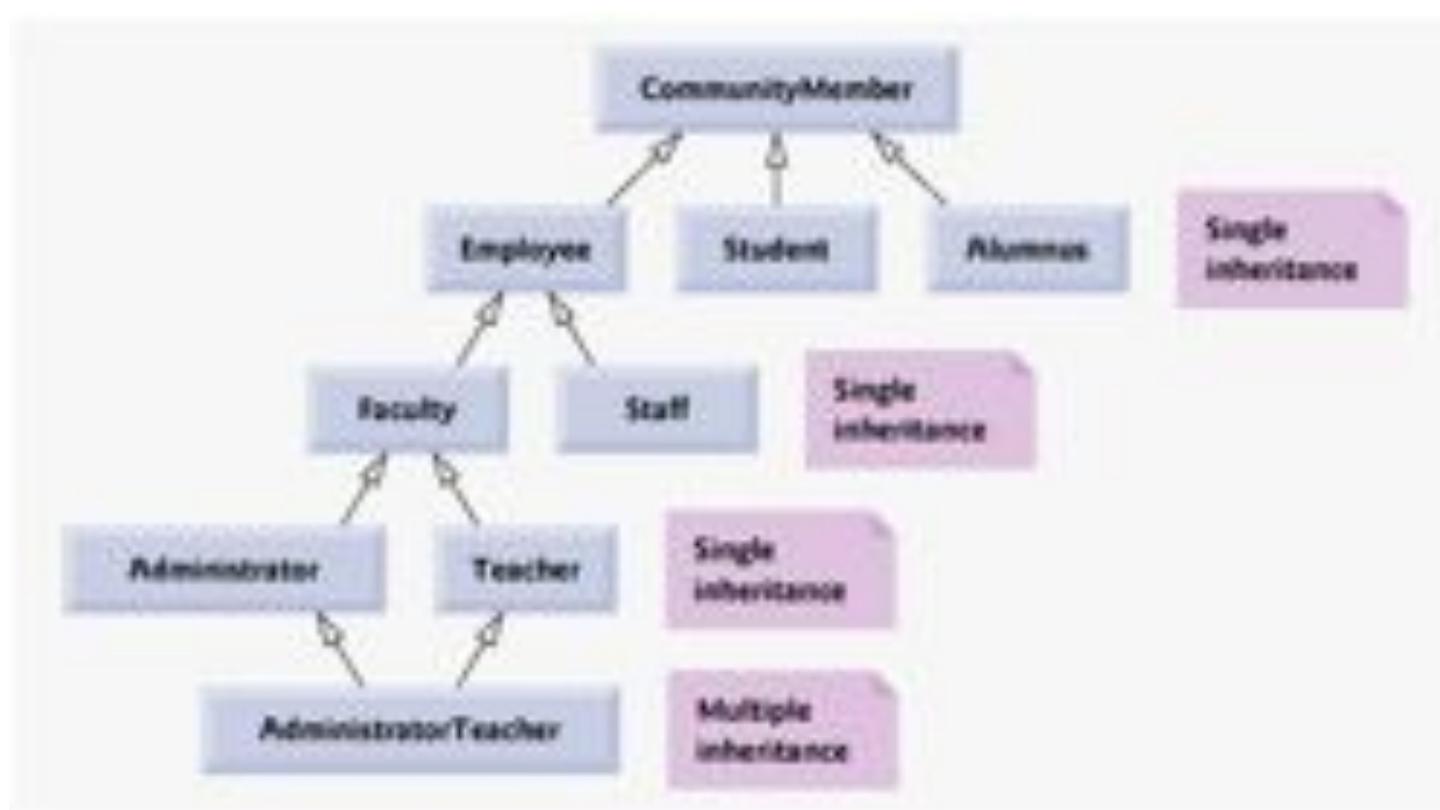
# Exemplo 1

- Entre os empregados dos departamentos e escolas (*Faculty*), temos:
  - Professores (*Teacher*);
  - Administradores (*Administrator*).
- Por fim, há uma sutileza
  - Os administradores são também professores
    - Como os chefes de departamento.

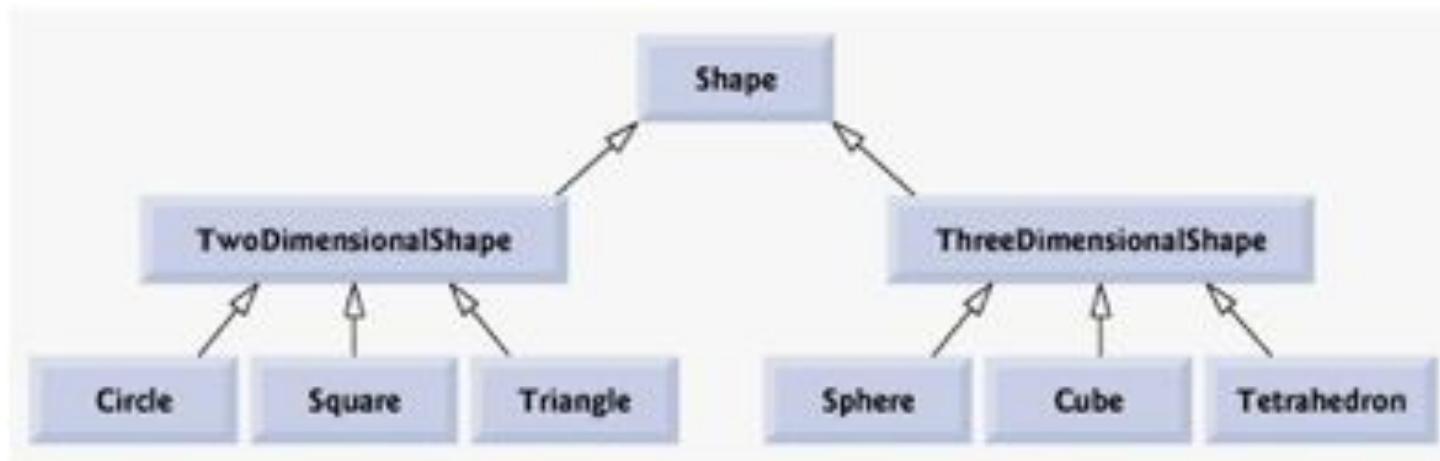
# Exemplo 1

- *CommunityMember*
  - *Alumnus*;
  - *Students*;
  - *Employees*
    - *Staff*;
    - *Faculty*
      - *Administrator*;
      - *Teacher*
        - *AdministratorTeacher*.
- Cada seta na hierarquia apresentada a seguir representa um relacionamento “é um”;
- *CommunityMember* é a base direta ou indireta de todas as classes da hierarquia.

# Exemplo 1



# Exemplo 2



- A cada nível da hierarquia o nível de especialização dos objetos aumenta
  - Objetos mais detalhados.
- Poderíamos ainda continuar a especializar os objetos
  - Elipse, Retângulo e Trapezóide por exemplo.

# Herança

- Definimos que a classe *TwoDimensionalShape* é derivada da classe *Shape* da seguinte forma em C++

```
class TwoDimensionalShape : public Shape
```

- Este é um exemplo de **herança pública**
  - A forma utilizada mais comumente;
  - Note o especificador de acesso;
  - Há também a herança **privada** e **protegida**.

# Exemplo 1

- Em todos os tipos de herança, os membros privados da classe base não são acessíveis a partir da classe derivada
  - Embora sejam herdados e considerados parte da classe derivada.
- Na herança pública, os membros mantêm sua visibilidade original
  - Membros públicos da classe base são membros públicos da classe derivada;
  - Membros protegidos da classe base são membros protegidos da classe derivada.
- Funções amigas não são herdadas.

# Especificadores de Acesso

- Revisando os especificadores de acesso (ou visibilidade)
  - *public*: acessível dentro da classe base e em qualquer parte do programa em que houver uma referência, ponteiro ou objeto da classe base;
  - *private*: acessível apenas ao código interno da classe base e a funções amigas.
  - *protected*: acessível ao código interno e funções amigas da classe base e aos membros e funções amigas de classes **derivadas**
    - Membros de classes derivadas podem chamar os métodos da classe base, somente invocando o nome.

# Herança

- Nosso exemplo de implementação considerará uma empresa e seus funcionários
  - Funcionários Comissionados são pagos com comissões sobre vendas;
  - Funcionários Assalariados Comissionados são pagos com um salário fixo e também recebem comissões sobre vendas.
- O primeiro tipo de funcionário será representado pela classe base, enquanto o segundo tipo será representado pela classe derivada.

# Herança

- A partir deste contexto, serão apresentados 4 exemplos:
  1. Classe *ComissionEmployee*, que representa funcionários comissionados
    - Membros Privados
  2. Classe *BasePlusCommissionEmployee*, que representa funcionários assalariados comissionados
    - Sem herança.
  3. Nova classe *BasePlusCommissionEmployee*
    - Usando herança;
    - Classe *ComissionEmployee* com membros *protected*.
  4. Nova herança, porém, classe *ComissionEmployee* com membros *private*.

# *CommissionEmployee.h*

```
#include <string> // classe string padrão C++
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee(const string &, const string &, const string &, double = 0.0, double = 0.0);
    void setFirstName( const string & ); // configura o nome
    string getFirstName() const; // retorna o nome
    void setLastName( const string & ); // configura o sobrenome
    string getLastName() const; // retorna o sobrenome
    void setSocialSecurityNumber( const string & ); // configura o SSN
    string getSocialSecurityNumber() const; // retorna o SSN
    void setGrossSales( double ); // configura a quantidade de vendas brutas
    double getGrossSales() const; // retorna a quantidade de vendas brutas
    void setCommissionRate( double ); // configura a taxa de comissão (porcentagem)
    double getCommissionRate() const; // retorna a taxa de comissão
    double earnings() const; // calcula os rendimentos
    void print() const; // imprime o objeto CommissionEmployee

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // vendas brutas semanais
    double commissionRate; // porcentagem da comissão
};
```

# *CommissionEmployee.h*

- Note os atributos são privados
  - Não acessíveis por classes derivadas.
- Os *getters* e *setters* desta classe são públicos
  - Acessíveis por classes derivadas;
  - Podem ser utilizados como meio para acessar os atributos
    - O que é uma boa prática de engenharia de *software*.
- Além disto, getters são declarados como *const*
  - Não podem alterar o valor de nenhum argumento.
- As *strings* recebidas pelo construtor também são declaradas como *referência constante*
  - Na prática não surte efeito, mas é uma boa prática;
  - Passagem por referência e prevenção de alteração.

# *CommissionEmployee.cpp*

```
#include <iostream>
using namespace std;
#include "CommissionEmployee.h" // Definição da classe CommissionEmployee

// construtor
CommissionEmployee::CommissionEmployee( const string &first, const string &last,
                                         const string &ssn, double sales, double rate)
{
    firstName = first; // deve validar
    lastName = last; // deve validar
    socialSecurityNumber = ssn; // deve validar
    setGrossSales( sales ); // valida e armazena as vendas brutas
    setCommissionRate( rate ); // valida e armazena a taxa de comissão
}

// configura o nome
void CommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // deve validar
}

// retorna o nome
string CommissionEmployee::getFirstName() const
{
    return firstName;
}
```

# *CommissionEmployee.cpp*

- Note que o construtor da classe base acessa e realiza atribuições aos atributos diretamente
  - Os atributos *sales* e *rate* são acessados através de métodos porque estes realizam testes de consistência nos valores passados por parâmetros;
  - Os demais atributos também poderiam ser checados quanto a sua consistência
    - Número correto de dígitos e comprimento máximo de *strings*.

# *CommissionEmployee.cpp*

```
// configura o sobrenome
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // deve validar
}

// retorna o sobrenome
string CommissionEmployee::getLastName() const
{
    return lastName;
}

// configura o SSN
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // deve validar
}

// retorna o SSN
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
}
```

# *CommissionEmployee.cpp*

```
// configura a quantidade de vendas brutas
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
}

// retorna a quantidade de vendas brutas
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
}

// configura a taxa de comissão
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
}

// retorna a taxa de comissão
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
}
```

# *CommissionEmployee.cpp*

```
// calcula os rendimentos
double CommissionEmployee::earnings() const
{
    return commissionRate * grossSales;
}

// imprime o objeto CommissionEmployee
void CommissionEmployee::print() const
{
    cout << "commission employee: " << firstName << ' ' << lastName
        << "\nsocial security number: " << socialSecurityNumber
        << "\ngross sales: " << grossSales
        << "\ncommission rate: " << commissionRate;
}
```

# *DriverCommissionEmployee.cpp*

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "CommissionEmployee.h" // Definição da classe CommissionEmployee

int main()
{
    // instancia um objeto CommissionEmployee
    CommissionEmployee employee("Sue", "Jones", "222-22-2222", 10000, .06 );

    // configura a formatação de saída de ponto flutuante
    cout << fixed << setprecision( 2 );

    // obtém os dados do empregado comissionado
    cout << "Employee information obtained by get functions: \n"
        << "\nFirst name is " << employee.getFirstName()
        << "\nLast name is " << employee.getLastName()
        << "\nSocial security number is "
        << employee.getSocialSecurityNumber()
        << "\nGross sales is " << employee.getGrossSales()
        << "\nCommission rate is " << employee.getCommissionRate() << endl;
```

# *DriverCommissionEmployee.cpp*

```
employee.setGrossSales( 8000 ); // configura vendas brutas
employee.setCommissionRate( .1 ); // configura a taxa de comissão

cout << "\nUpdated employee information output by print function: \n" << endl;

employee.print(); // exibe as novas informações do empregado

// exibe os rendimentos do empregado
cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;

return 0;
}
```

# Saída do Exemplo

Employee information obtained by get functions:

First name is Sue

Last name is Jones

Social security number is 222-22-2222

Gross sales is 10000.00

Commission rate is 0.06

Updated employee information output by print function:

commission employee: Sue Jones

social security number: 222-22-2222

gross sales: 8000.00

commission rate: 0.10

Employee's earnings: \$800.00

# Herança

- Vamos agora criar a classe *BasePlusCommissionEmployee*, que representa funcionários assalariados comissionados
  - Embora seja uma especialização da classe anterior, vamos definir completamente a classe.

# *BasePlusCommissionEmployee.h*

```
#include <string> // classe string padrão C++
using namespace std;

class BasePlusCommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, const string &,
                                double = 0.0, double = 0.0, double = 0.0 );
    void setFirstName( const string & ); // configura o nome
    string getFirstName() const; // retorna o nome
    void setLastName( const string & ); // configura o sobrenome
    string getLastName() const; // retorna o sobrenome
    void setSocialSecurityNumber( const string & ); // configura o SSN
    string getSocialSecurityNumber() const; // retorna o SSN
    void setGrossSales( double ); // configura a quantidade de vendas brutas
    double getGrossSales() const; // retorna a quantidade de vendas brutas
    void setCommissionRate( double ); // configura a taxa de comissão
    double getCommissionRate() const; // retorna a taxa de comissão
    void setBaseSalary( double ); // configura o salário-base
    double getBaseSalary() const; // retorna o salário-base
    double earnings() const; // calcula os rendimentos
    void print() const; // imprime o objeto BasePlusCommissionEmployee
```

# *BasePlusCommissionEmployee.h*

**private:**

```
string firstName;  
string lastName;  
string socialSecurityNumber;  
double grossSales; // vendas brutas semanais  
double commissionRate; // porcentagem da comissão  
double baseSalary; // salário-base  
};
```

# Herança

- Note que foram adicionados um atributo com respectivo *getter* e *setter*, além de um parâmetro adicional no construtor;
- O restante do código é basicamente redundante em relação à classe *ComissionEmployee*.

# *BasePlusCommissionEmployee.cpp*

```
#include <iostream>
using namespace std;

// Definição da classe BasePlusCommissionEmployee
#include "BasePlusCommissionEmployee.h"

// construtor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
    const string &first, const string &last, const string &ssn,
    double sales, double rate, double salary )
{
    firstName = first; // deve validar
    lastName = last; // deve validar
    socialSecurityNumber = ssn; // deve validar
    setGrossSales( sales ); // valida e armazena as vendas brutas
    setCommissionRate( rate ); // valida e armazena a taxa de comissão
    setBaseSalary( salary ); // valida e armazena salário-base
}

// configura o nome
void BasePlusCommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // deve validar
}
```

# *BasePlusCommissionEmployee.cpp*

```
// retorna o nome
string BasePlusCommissionEmployee::getFirstName() const
{
    return firstName;
}

// configura o sobrenome
void BasePlusCommissionEmployee::setLastName(const string &last)
{
    lastName = last; // deve validar
}

// retorna o sobrenome
string BasePlusCommissionEmployee::getLastName() const
{
    return lastName;
}

// configura o SSN
void BasePlusCommissionEmployee::setSocialSecurityNumber(const string &ssn)
{
    socialSecurityNumber = ssn; // deve validar
}
```

# *BasePlusCommissionEmployee.cpp*

```
// retorna o SSN
string BasePlusCommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
}

// configura a quantidade de vendas brutas
void BasePlusCommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
}

// retorna a quantidade de vendas brutas
double BasePlusCommissionEmployee::getGrossSales() const
{
    return grossSales;
}

// configura a taxa de comissão
void BasePlusCommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
}
```

# *BasePlusCommissionEmployee.cpp*

```
// retorna a taxa de comissão
double BasePlusCommissionEmployee::getCommissionRate() const
{
    return commissionRate;
}

// configura o salário-base
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
}

// retorna o salário-base
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
}

// calcula os rendimentos
double BasePlusCommissionEmployee::earnings() const
{
    return baseSalary + ( commissionRate * grossSales );
}
```

# *BasePlusCommissionEmployee.cpp*

```
// imprime o objeto BasePlusCommissionEmployee
void BasePlusCommissionEmployee::print() const
{
    cout << "base-salaried commission employee: " << firstName << ''
        << lastName << "\nsocial security number: " << socialSecurityNumber
        << "\ngross sales: " << grossSales
        << "\ncommission rate: " << commissionRate
        << "\nbase salary: " << baseSalary;
}
```

# *driverBasePlusCommissionEmployee.cpp*

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

int main()
{
    // instancia o objeto BasePlusCommissionEmployee
    BasePlusCommissionEmployee
        employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

    // configura a formatação de saída de ponto flutuante
    cout << fixed << setprecision( 2 );

    // obtém os dados do empregado comissionado
    cout << "Employee information obtained by get functions: \n"
        << "\nFirst name is " << employee.getFirstName()
        << "\nLast name is " << employee.getLastName()
        << "\nSocial security number is "
        << employee.getSocialSecurityNumber()
        << "\nGross sales is " << employee.getGrossSales()
        << "\nCommission rate is " << employee.getCommissionRate()
        << "\nBase salary is " << employee.getBaseSalary() << endl;
```

# *driverBasePlusCommissionEmployee.cpp*

```
employee.setBaseSalary( 1000 ); // configura o salário-base  
  
cout << "\nUpdated employee information output by print function: \n"  
     << endl;  
  
employee.print(); // exibe as novas informações do empregado  
  
// exibe os rendimentos do empregado  
cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;  
  
return 0;  
}
```

# Saída do Exemplo

Employee information obtained by get functions:

First name is Bob

Last name is Lewis

Social security number is 333-33-3333

Gross sales is 5000.00

Commission rate is 0.04

Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis

social security number: 333-33-3333

gross sales: 5000.00

commission rate: 0.04

base salary: 1000.00

Employee's earnings: \$1200.00

# Comparação

CommissionEmployee	BasePlusCommissionEmployee
-firstName	-firstName
-lastName	-lastName
-socialSecurityNumber	-socialSecurityNumber
-grossSales	-grossSales
-comissionRate	-comissionRate
+commissionEmployee()	+BasePlusCommissionEmployee()
+setFirstName()	+setFirstName()
+getFirstName()	+getFirstName()
+setLastName()	+setLastName()
+getLastname()	+getLastname()
+setSocialSecurityNumber()	+setSocialSecurityNumber()
+getSocialSecurityNumber()	+getSocialSecurityNumber()
+setGrossSales()	+setGrossSales()
+getGrossSales()	+getGrossSales()
+setCommissionRate()	+setCommissionRate()
+getCommissionRate()	+getCommissionRate()
+earnings()	+setBaseSalary()
+print()	+getBaseSalary()
	+earnings()
	+print()

# Herança

- Como pode ser notado:
  - Ambas as classes compartilham a maior parte dos atributos, que são privados;
  - Consequentemente, os *getters* e *setters* relacionados também são compartilhados
    - Foi acrescentado apenas um *getter* e um *setter* devido ao novo atributo.
  - Parte dos métodos foi adaptada para tratar o atributo extra na segunda classe
    - Construtor e *print*.
  - Literalmente o código original foi copiado e adaptado.
- Quando a redundância entre classes acontece, caracteriza-se a necessidade de herança.

# Herança

- A replicação de código pode resultar em replicação de erros:
  - A manutenção é dificultada, pois cada cópia tem que ser corrigida;
  - Desperdício de tempo.
- Imagine um sistema composto de várias classes parecidas divididas em diversos códigos replicados
  - Menos é mais.
- Definimos uma classe que absorverá os atributos e métodos redundantes
  - A classe base;
  - A manutenção dada na classe base se reflete nas classes derivadas automaticamente.

# Herança

- Nosso próximo exemplo fixa a classe *ComissionEmployee* como classe base
  - Note que utilizaremos a mesma definição desta classe do exemplo anterior;
  - Os atributos continuam privados.
- A classe *BasePlusCommissionEmployee* será a classe derivada
  - Acrescentará o atributo *baseSalary*;
  - Acrescentará também *getter* e *setter* e redefinirá dois métodos.
- Qualquer tentativa de acesso aos membros privados da classe base gerará erro de compilação.

# *BasePlusComissionEmployee.h*

```
#include <string> // classe string padrão C++
using namespace std;
#include "CommissionEmployee.h" // Declaração da classe CommissionEmployee

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, const string &,
                                double = 0.0, double = 0.0, double = 0.0 );
    void setBaseSalary( double ); // configura o salário-base
    double getBaseSalary() const; // retorna o salário-base
    double earnings() const; // calcula os rendimentos
    void print() const; // imprime o objeto BasePlusCommissionEmployee

private:
    double baseSalary; // salário-base
};
```

# Herança

- O operador **:** define a herança;
- Note que 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.
  - Note que o construtor não foi herdado
    - Foi definido um construtor específico.
- É necessário incluir o *header* da classe a ser herdada
  - Permite a utilização do nome da classe, determina o tamanho dos objetos e garante que a interface será respeitada.

# *BasePlusComissionEmployee.cpp*

```
#include <iostream>
using namespace std;
// Definição da classe BasePlusCommissionEmployee
#include "BasePlusCommissionEmployee.h"

// construtor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(const string &first,
const string &last, const string &ssn, double sales, double rate, double salary)
    // chama explicitamente o construtor da classe básica
    : CommissionEmployee( first, last, ssn, sales, rate )
{
    setBaseSalary( salary ); // valida e armazena salário-base
}
// configura o salário-base
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
}
// retorna o salário-base
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
}
```

# *BasePlusComissionEmployee.cpp*

```
// calcula os rendimentos
double BasePlusCommissionEmployee::earnings() const
{
    // a classe derivada não pode...
    return baseSalary + ( commissionRate * grossSales );
}

// imprime o objeto BasePlusCommissionEmployee
void BasePlusCommissionEmployee::print() const
{
    // a classe derivada não pode...
    cout << "base-salaried commission employee: " << firstName << ''
        << lastName << "\nsocial security number: " << socialSecurityNumber
        << "\ngross sales: " << grossSales
        << "\ncommission rate: " << commissionRate
        << "\nbase salary: " << baseSalary;
}
```

# Herança

- Neste exemplo, o construtor da classe derivada chama explicitamente o construtor da classe base
  - Sintaxe inicializadora 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.

# Herança

- Os métodos *earnings* e *print()* deste exemplo gerarão erros de compilação
  - Ambos tentam acessar diretamente membros privados da classe base, o que não é permitido
    - Mesmo para classes intimamente relacionadas.
  - Poderíamos utilizar os *getters* associados a tais atributos para evitar os erros de compilação
    - Uma vez que os *getters* são públicos.
  - Os métodos podem ser redefinidos sem causar erros de compilação, como veremos em breve.

# Herança

- Vamos modificar o primeiro exemplo (classe *ComissionEmployee*) para que seus atributos sejam ***protected***
  - O modificador de acesso *protected* (# em UML) permite que um membro seja acessível por:
    - Membros e funções amigas da classe base;
    - Membros e funções amigas das classes derivadas.
  - Desta forma o segundo exemplo (classe *BasePlusComissionEmployee*) compilará sem erros
    - Poderá acessar os atributos diretamente.

# *ComissionEmployee.h*

```
#include <string> // classe string padrão C++
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee( const string &, const string &, const string &,
                        double = 0.0, double = 0.0 );

    void setFirstName( const string & ); // configura o nome
    string getFirstName() const; // retorna o nome

    void setLastName( const string & ); // configura o sobrenome
    string getLastName() const; // retorna o sobrenome

    void setSocialSecurityNumber( const string & ); // configura SSN
    string getSocialSecurityNumber() const; // retorna SSN

    void setGrossSales( double ); // configura a quantidade de vendas brutas
    double getGrossSales() const; // retorna a quantidade de vendas brutas
```

# *ComissionEmployee.h*

```
void setCommissionRate( double ); // configura a taxa de comissão
double getCommissionRate() const; // retorna a taxa de comissão

double earnings() const; // calcula os rendimentos
void print() const; // imprime o objeto CommissionEmployee
```

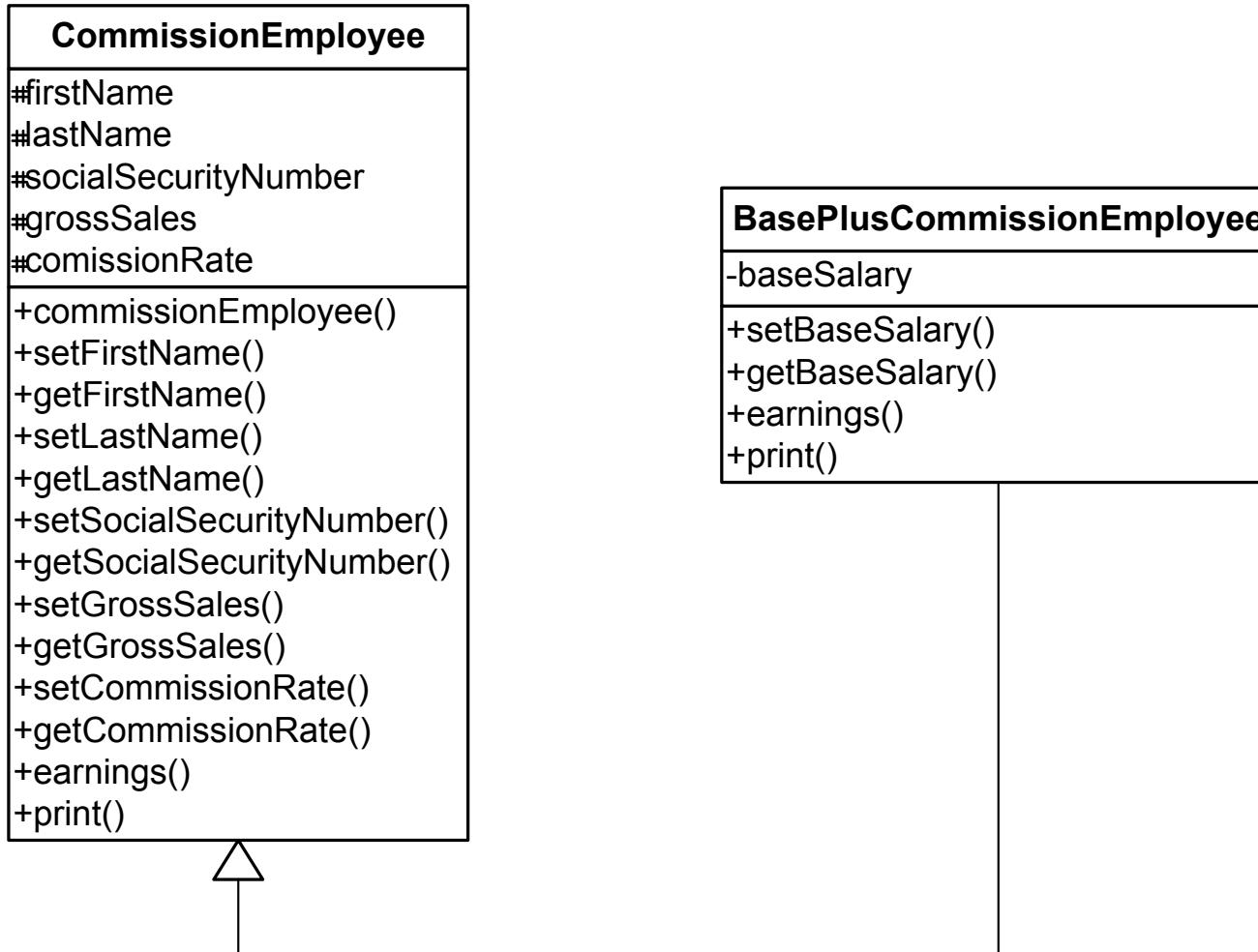
## **protected:**

```
string firstName;
string lastName;
string socialSecurityNumber;
double grossSales; // vendas brutas semanais
double commissionRate; // porcentagem da comissão
};
```

# Herança

- A implementação da classe *ComissionEmployee* não muda em rigorosamente nada
  - Internamente à classe base, nada muda se um membro é privado ou protegido.
- A implementação da classe *BasePlusComissionEmployee* também não muda
  - Na verdade, apenas herdando da nova classe base é que ela funciona!

# Diagrama de Classes



# Considerações Sobre Membros *Protected*

- Utilizar atributos protegidos nos dá uma leve melhoria de desempenho
  - Não é necessário ficar chamando funções (*getters* e *setters*).
- 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.
  - Qualquer alteração de nomenclatura de um atributo invalida toda a classe que herda seu código
    - Uma classe derivada deve depender do serviço prestado, e não da implementação da classe base.

# Considerações Sobre Membros *Protected*

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

# Herança

- Vamos adequar nossas classes dos exemplos anteriores ao padrão proposto
  - De acordo com as boas práticas de engenharia de *software*.
- A classe base volta a ter atributos privados
  - A implementação também passa a utilizar *getters* e *setters* internamente.
- A classe derivada não acessa os atributos diretamente
  - Utiliza *getters* e *setters*;
  - Somente a implementação da classe é alterada.

# *CommissionEmployee.h*

```
#include <string> // classe string padrão C++
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee(const string &, const string &, const string &, double = 0.0, double = 0.0);
    void setFirstName( const string & ); // configura o nome
    string getFirstName() const; // retorna o nome
    void setLastName( const string & ); // configura o sobrenome
    string getLastName() const; // retorna o sobrenome
    void setSocialSecurityNumber( const string & ); // configura o SSN
    string getSocialSecurityNumber() const; // retorna o SSN
    void setGrossSales( double ); // configura a quantidade de vendas brutas
    double getGrossSales() const; // retorna a quantidade de vendas brutas
    void setCommissionRate( double ); // configura a taxa de comissão (porcentagem)
    double getCommissionRate() const; // retorna a taxa de comissão
    double earnings() const; // calcula os rendimentos
    void print() const; // imprime o objeto CommissionEmployee

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // vendas brutas semanais
    double commissionRate; // porcentagem da comissão
};
```

# *CommissionEmployee.cpp*

```
CommissionEmployee::CommissionEmployee(const string &first, const string  
    &last, const string &ssn, double sales, double rate )  
: firstName( first ), lastName( last ), socialSecurityNumber( ssn )  
{  
    setGrossSales( sales ); // valida e armazena as vendas brutas  
    setCommissionRate( rate ); // valida e armazena a taxa de comissão  
}  
  
double CommissionEmployee::earnings() const  
{  
    return getCommissionRate() * getGrossSales();  
}  
  
// imprime o objeto CommissionEmployee  
void CommissionEmployee::print() const  
{  
    cout << "commission employee: "  
        << getFirstName() << ' ' << getLastName()  
        << "\nsocial security number: " << getSocialSecurityNumber()  
        << "\ngross sales: " << getGrossSales()  
        << "\ncommission rate: " << getCommissionRate();  
}
```

# Herança

- Note que o construtor inicializa os atributos em seu próprio cabeçalho
  - Após a assinatura, utilizamos : e em seguida o nome do parâmetro seguido do valor a ser atribuído, entre parênteses;
  - Forma alternativa para o construtor;
  - Executado antes do próprio construtor;
  - Em alguns casos, é obrigatório
    - Constantes e referências.

# *BasePlusCommissionEmployee.cpp*

```
// calcula os rendimentos
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
}

// imprime o objeto BasePlusCommissionEmployee
void BasePlusCommissionEmployee::print() const
{
    cout << "base-salaried ";

    // invoca a função print de CommissionEmployee
    CommissionEmployee::print();

    cout << "\nbase salary: " << getBaseSalary();
}
```

# Saída do Método *print()*

Employee information obtained by get functions:

```
First name is Bob  
Last name is Lewis  
Social security number is 333-33-3333  
Gross sales is 5000.00  
Commission rate is 0.04  
Base salary is 300.00
```

Updated employee information output by print function:

```
base-salaried commission employee: Bob Lewis  
social security number: 333-33-3333  
gross sales: 5000.00  
commission rate: 0.04  
base salary: 1000.00
```

**Employee's earnings: \$1200.00**

# Compilação

# Compilação

- Compilamos primeiro a classe base
  - g++ -c ComissionEmployee.cpp
- Compilamos então a classe derivada
  - g++ -c BasePlusComissionEmployee.cpp
- Compilamos o programa driver utilizando os arquivos .o das duas compilações anteriores
  - g++ ComissionEmployee.o BasePlusComissionEmployee.o driver.cpp –o programa
- Uma outra maneira é compilar todos os arquivos .cpp das classes
  - g++ -c \*Employee.cpp
  - g++ \*.o driver.cpp –o programa



**Continua na  
próxima aula...**

# Redefinição de Métodos

# Redefinição de Métodos

- Classes derivadas podem redefinir métodos da classe base
  - A assinatura pode até mudar, embora o nome do método permaneça;
  - A precedência é do método redefinido na classe derivada
    - Na verdade, este **substitui** o método da classe base na classe derivada;
    - Em caso de assinaturas diferentes, pode haver erro de compilação caso tentemos chamar o método da classe base.
- A classe derivada pode chamar o método da classe base
  - Desde que use o operador de escopo :: precedido pelo nome da classe base.

# Redefinição de Métodos

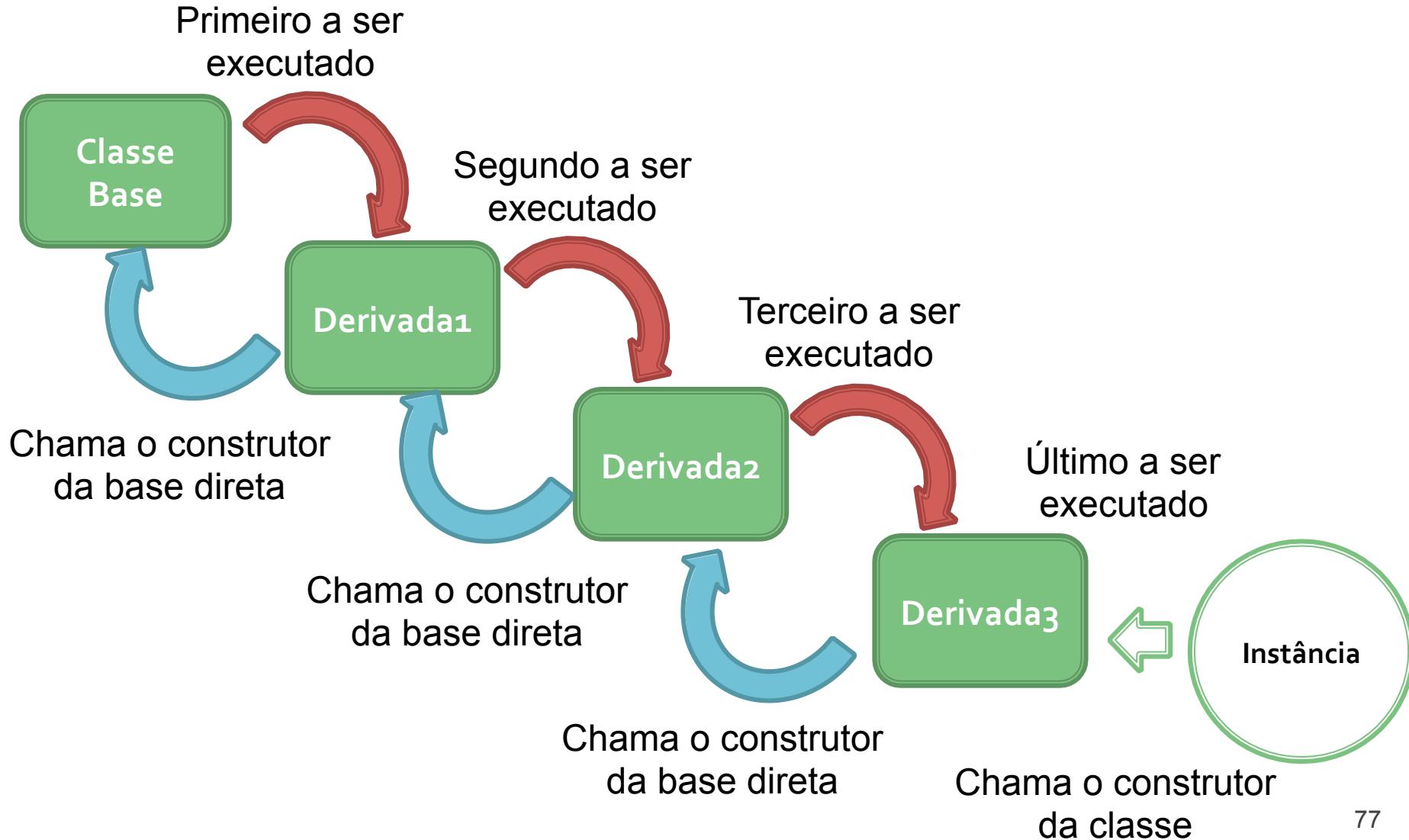
- É comum que métodos redefinidos chamem o método original dentro de sua redefinição e acrescentem funcionalidades
  - Como no exemplo anterior, em que frases adicionais são impressas na redefinição do método *print()*.

# Construtores e Destrutores em Classes Derivadas

# Construtores e Destrutores em Classes Derivadas

- Como vimos no exemplo, instanciar uma classe derivada inicia uma cadeia de chamadas a construtores
  - O construtor da classe derivada chama o construtor da classe base antes de executar suas próprias ações;
  - O construtor da classe base é chamada direta ou indiretamente (construtor *default*).
- Considerando um hierarquia de classes mais extensa:
  - O primeiro construtor **chamado** é a última classe derivada
    - E é o último a ser **executado**.
  - O último construtor **chamado** é o da classe base
    - E é o primeiro a ser **executado**.

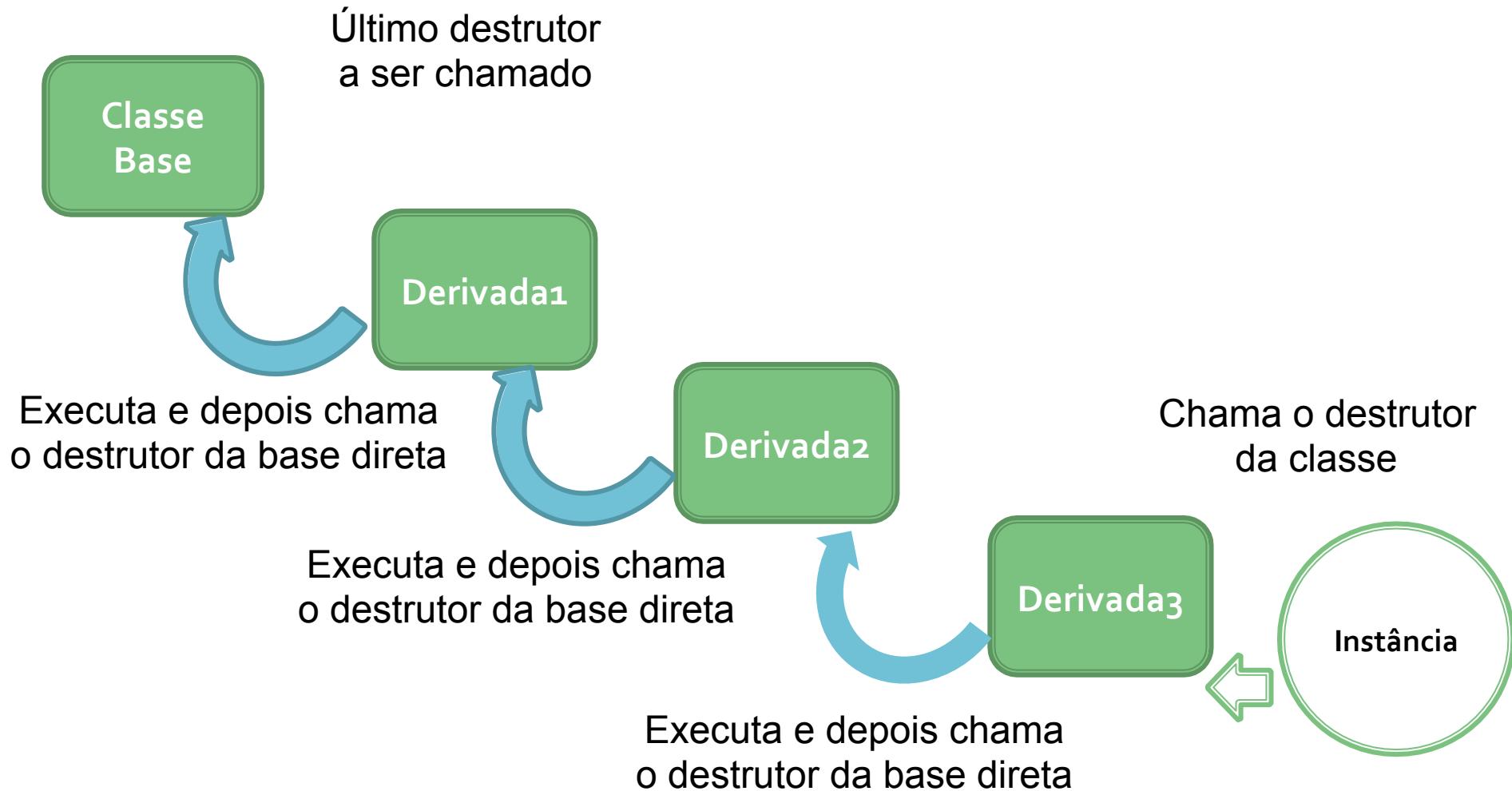
# Construtores e Destrutores em Classes Derivadas



# Construtores e Destrutores em Classes Derivadas

- O contrário acontece em relação aos destrutores
  - A execução começa pela classe derivada e é propagada até a classe base.
- Considerando um hierarquia de classes mais extensa:
  - O primeiro destrutor **chamado** é a última classe derivada
    - E é também o primeiro a ser **executado**.
  - O último construtor **chamado** é o da classe base
    - E é também o último a ser **executado**.

# Construtores e Destrutores em Classes Derivadas



# Construtores e Destrutores em Classes Derivadas

- Caso um atributo de uma classe derivada seja um objeto de outra classe:
  - Seu construtor será o primeiro a ser executado;
  - Seu destrutor será o último a ser executado.
- **Atenção!**
  - Construtores e destrutores de uma classe base não são herdados;
  - No entanto, podem ser chamados pelos construtores e destrutores das classes derivadas.

# Herança PÚblica vs. Privada vs. Protegida

# Herança Pública vs. Privada vs. Protegida

## ■ Herança Pública

- Os membros públicos da classe base serão membros públicos da classe derivada
  - Poderão ser acessados por objetos da classe derivada.
- Os membros protegidos da classe base serão membros protegidos da classe derivada;

## ■ Herança Privada

- Tantos os membros públicos quanto os protegidos da classe base serão membros privados da classe derivada;
- Acessíveis aos membros da classe derivada, mas não aos seus objetos.

## ■ Herança Protegida

- Tantos os membros públicos quanto os protegidos da classe base serão membros protegidos da classe derivada
  - Disponíveis aos membros da classe derivada e as que herdarem dela;
  - Um objeto da classe derivada não terá acesso a nenhum membro da classe derivada.

# Herança

## Pública vs. Privada vs. Protegida

```
class Base
{
    protected:
        int protegido;
    private:
        int privado;
    public:
        int publico;
};
```

# Herança

## Pública vs. Privada vs. Protegida

```
class Derivada1: public Base
{
    private:
        int a, b, c;
    public:
        Derivada1()
        {
            a = protegido;
            b = privado; //ERRO: Não acessível
            c = publico;
        }
};
```

# Herança

## Pública vs. Privada vs. Protegida

```
class Derivada2: private Base
{
    private:
        int a, b, c;
    public:
        Derivada2()
    {
        a = protegido;
        b = privado; //ERRO: Não acessível
        c = publico;
    }
};
```

# Herança

## Pública vs. Privada vs. Protegida

```
class Derivada3:protected Base
{
    private:
        int a, b, c;
    public:
        Derivada3()
        {
            a = protegido;
            b = privado;//ERRO: Não acessível
            c = publico;
        }
};
```

# Herança

## Pública vs. Privada vs. Protegida

```
int main()
{
    int x;
    Derivada1 HPublica;
    x = HPublica.protegido;           //ERRO: Não acessível
    x = HPublica.privado;            //ERRO: Não acessível
    x = HPublica.publico;           //OK

    Derivada2 HPrivada;
    x = HPrivada.protegido;          //ERRO: Não acessível
    x = HPrivada.privado;            //ERRO: Não acessível
    x = HPrivada.publico;           //ERRO: Não acessível

    Derivada3 HProtegida;
    x = HProtegida.protegido;         //ERRO: Não acessível
    x = HProtegida.privado;           //ERRO: Não acessível
    x = HProtegida.publico;           //ERRO: Não acessível

    return 0;
}
```

# Herança

## Pública vs. Privada vs. Protegida

Base-class member- access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
public	public in derived class.  Can be accessed directly by member functions, friend functions and nonmember functions.	protected in derived class.  Can be accessed directly by member functions and friend functions.	private in derived class.  Can be accessed directly by member functions and friend functions.
protected	protected in derived class.  Can be accessed directly by member functions and friend functions.	protected in derived class.  Can be accessed directly by member functions and friend functions.	private in derived class.  Can be accessed directly by member functions and friend functions.
private	Hidden in derived class.  Can be accessed by member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class.  Can be accessed by member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class.  Can be accessed by member functions and friend functions through public or protected member functions of the base class.

# Conversões de Tipo entre Base e Derivada

# Herança

- Se necessário, podemos converter um objeto de uma classe derivada em um objeto da classe base
  - A classe base pode conter menos atributos que a classe derivada
    - Os atributos em comum são copiados;
    - Os atributos extras são desperdiçados.
- O contrário não pode ser feito
  - Conversão de um objeto da classe base para um objeto da classe derivada.
- A conversão de tipos está intimamente relacionada com o Polimorfismo, que veremos em breve.
- Para nosso exemplo, consideraremos duas classes:
  - *Base* e *Derivada*;
  - Em que a segunda herda da primeira.

# Conversões de Tipo entre Base e Derivada

```
int main()
{
    Base obj1;
    Derivada obj2;

    obj1.get();           //“Preenche” o objeto.
    obj2.get();           //“Preenche” o objeto.

    obj1 = obj2;          //OK.
    obj2 = obj1;          //ERRO! Não pode ser convertido.

    return 0;
}
```

# Herança Múltipla

# Herança Múltipla

- Uma classe pode ser derivada a partir de mais que uma classe base
  - Caracterizando assim a **herança múltipla**.
- A complexidade relacionada à herança múltipla diz respeito ao projeto do relacionamento das classes
  - A sintaxe é similar à herança simples;
  - Após o cabeçalho da classe derivada, usamos : e listamos as classes a serem herdadas, separadas por vírgula e com modificadores de acesso individuais.

**class** Derivada: **public** Base1, **public** Base2

# Herança Múltipla

- Considere para nosso exemplo que queremos modelar o registro de imóveis de uma imobiliária que trabalha apenas com vendas
  - Já possuímos uma classe **Cadastro** que utilizamos para outros fins;
  - Já possuímos também uma classe **Imóvel**, que armazena os dados de um imóvel;
  - Decidimos então não modificar as classes, e criar a classe **Venda**
    - Vamos incorporar o cadastro do dono e do imóvel a partir das nossas classes já existentes.
  - Resolvemos criar uma classe **Tipo**, que determina se o imóvel é residencial, comercial ou galpão.
  - Como último detalhe, os dados sobre comprador não são interessantes para a classe **Venda**.

# *Cadastro.h e Cadastro.cpp*

```
#include<string>
using namespace std;

class Cadastro
{
    private:
        string nome, fone;
    public:
        void setNome();
        void setFone();
        void print();
};
```

```
#include<iostream>
using namespace std;

#include "Cadastro.h"

void Cadastro::setNome()
{
    cout<<"Nome: "<<endl;
    getline(cin, nome);
}

void Cadastro::setFone()
{
    cout<<"Telefone: "<<endl;
    getline(cin, fone);
}

void Cadastro::print()
{
    cout<<"Nome: "<<nome<<endl
        <<"Telefone:"<<fone
        <<endl;
}
```

# *Imovel.h*

```
#include<string>
using namespace std;

class Imovel
{
    private:
        string endereco, bairro;
        float areaUtil, areaTotal;
        int quartos;
    public:
        void setEndereco();
        void setBairro();
        void setAreaUtil();
        void setAreaTotal();
        void setQuartos();
        void print();
};
```

# *Imovel.cpp*

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

void Imovel::setEndereco()
{
    cout<<"Endereço:"<<endl;
    getline(cin, endereco);
}

void Imovel::setBairro()
{
    cout<<"Bairro:"<<endl;
    getline(cin, bairro);
}

void Imovel::setAreaUtil()
{
    cout<<"Área Útil:"<<endl;
    cin>>areaUtil;
}

void Imovel::setAreaTotal()
{
    cout<<"Área Total:"<<endl;
    cin>>areaTotal;
}
```

```
void Imovel::setQuartos()
{
    cout<<"Número de Quartos:"
        <<endl;
    cin>>quartos;
}

void Imovel::print()
{
    cout <<"Endereço: "
        <<endereco<<endl
        <<"Bairro: "<<bairro
        <<endl<<"Área Útil: "
        <<areaUtil<<endl
        <<"Área Total: "
        <<areaTotal<<endl
        <<"Quartos: "<<quartos
        <<endl;
}
```

# *Tipo.h e Tipo.cpp*

```
#include<string>
using namespace std;

class Tipo
{
    private:
        string tipoImovel;
    public:
        void setTipoImovel();
        void print();
};
```

```
#include<iostream>
using namespace std;

#include "Tipo.h"

void Tipo::setTipoImovel()
{
    cout<<"Tipo do Imóvel:"<<endl;
    getline(cin, tipoImovel);
    cin.ignore(10, '\n');
}

void Tipo::print()
{
    cout<<"Tipo do Imóvel: "
        <<tipoImovel<<endl;
}
```

# Venda.h

```
#include<iostream>
using namespace std;

#include"Cadastro.h"
#include"Imovel.h"
#include"Tipo.h"

class Venda: private Cadastro, Imovel, Tipo
{
    private:
        float valor;
    public:
        void set();
        void print();
};
```

# Herança Múltipla

- A classe **Venda** realiza herança privada
  - Os métodos da classe base não são acessíveis aos objetos.
- **Atenção!**
  - Herdando da classe **Cadastro**, a classe **Venda** só terá espaço para os dados de um único cadastro;
  - Se quiséssemos cadastrar o comprador, não seria possível
    - Um objeto da classe Cadastro resolveria este “problema”.
  - Não é possível herdar duas vezes da mesma classe.

# Venda.cpp

```
#include<iostream>
using namespace std;

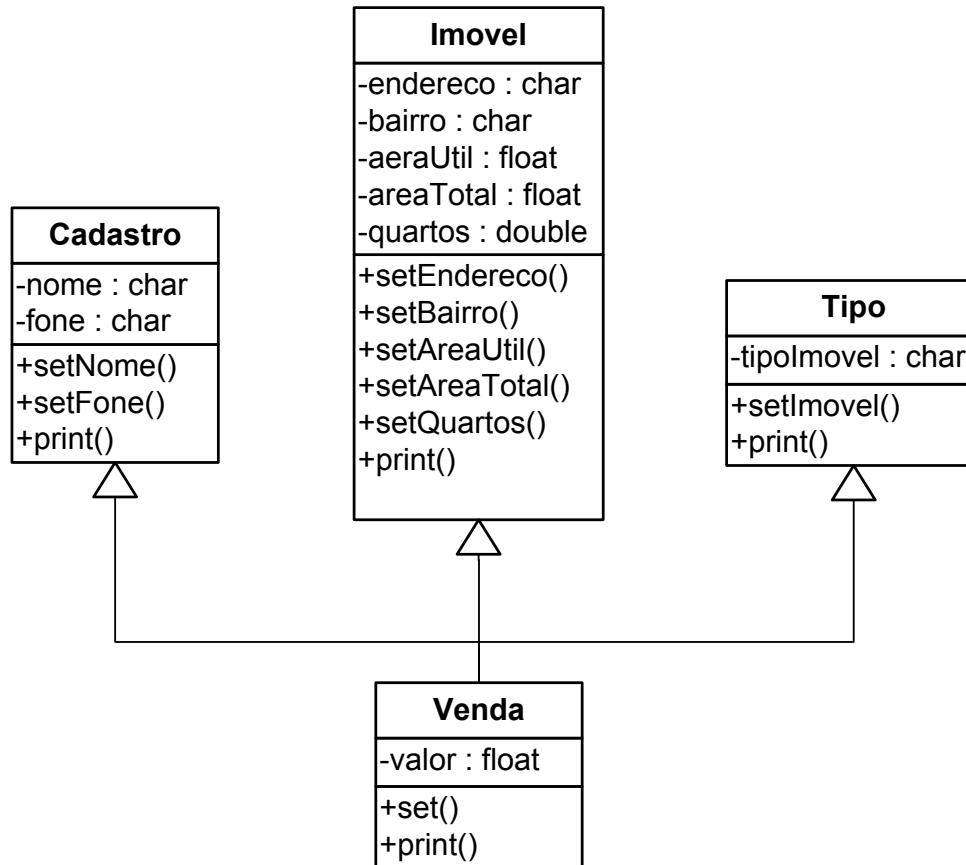
#include"Venda.h"

void Venda::set()
{
    cout<<"Proprietário:"<<endl;
    Cadastro::setNome();
    Cadastro::setFone();
    cout<<"Imóvel:"<<endl;
    Imovel::setEndereco();
    Imovel::setBairro();
    Imovel::setAreaTotal();
    Imovel::setAreaUtil();
    Tipo::setTipoImovel();

    cout<<"Valor de Venda:"<<endl;
    cin>>valor;
    cin.ignore(10, '\n');
}
```

```
void Venda::print()
{
    cout<<"Proprietário:"<<endl;
    Cadastro::print();
    cout<<"Imóvel:"<<endl;
    Imovel::print();
    Tipo::print();
    cout<<"Valor: $"<<valor<<endl;
}
```

# Diagrama de Classes



# Ambiguidade em Herança Múltipla

- Note que todas as classes base possuem um método *print()*
  - Dentro da classe derivada, devemos informar ao compilador a qual classe estamos nos referindo;
  - Utilizamos o nome da classe, o operador de escopo e o nome do método para que fique claro.  
**Base::metodo();**
- No caso de herança pública, o objeto que chame o método também deve usar o nome da classe e o operador de escopo  
**obj.Base::metodo();**

# *driverVenda.cpp*

```
#include<iostream>
using namespace std;

#include"Venda.h"

int main()
{
    Venda galpao;

    galpao.set();
    galpao.print();
}
```

# Saída do Método *print()*

Proprietário:

Nome: marco

Telefone: 3552-1663

Imóvel:

Endereço: Campus Morro do Cruzeiro

Bairro: Bauxita

Área Útil: 8

Área Total: 8

Quartos: 0

Tipo do Imóvel: Sala

Valor: R\$0

# Compilação

# Compilação

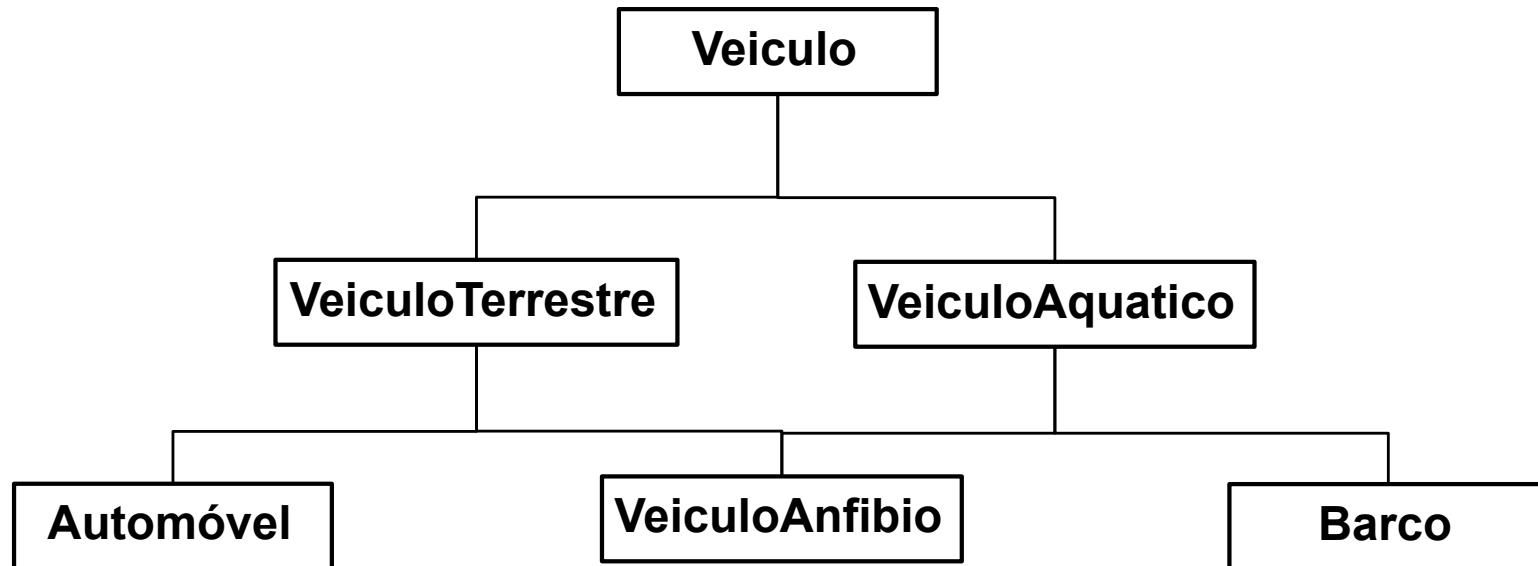
- Compilamos primeiro cada classe base
  - g++ -c Cadastro.cpp
  - g++ -c Imovel.cpp
  - g++ -c Tipo.cpp
- Compilamos então a classe derivada
  - g++ -c Venda.cpp
- Compilamos o programa driver utilizando os arquivos .o das duas compilações anteriores
  - g++ Cadastro.o Imovel.o Tipo.o Venda.o driver.cpp -o programa
- Uma outra maneira é compilar todos os arquivos .cpp das classes
  - g++ -c \*.cpp
  - g++ \*.o driver.cpp -o programa

# O Problema do Diamante (Classes Base Virtuais)

# O Problema do Diamante

- A herança múltipla fornece facilidades para reuso de *software*
  - O poder de combinar classes.
- No entanto, seu uso indiscriminado pode gerar problemas em uma estrutura de herança
  - Por exemplo, o problema do diamante, em que uma classe derivada herda de uma classe base indireta mais de uma vez.

# O Problema do Diamante



- Neste exemplo, a classe **VeiculoAnfibio** herda indiretamente duas vezes da classe **Veiculo**
  - Uma através da classe **VeiculoTerrestre**;
  - Uma através da classe **VeiculoAquatico**.

# O Problema do Diamante

- Considerando o exemplo anterior, suponhamos que desejamos utilizar um objeto da classe VeiculoAnfibio para invocar um método da classe Veiculo
  - Qual seria o caminho para atingir a classe Veiculo?
    - Qual dos membros seria utilizado?
  - Haveria ambiguidade, portanto, o código não compilaria.

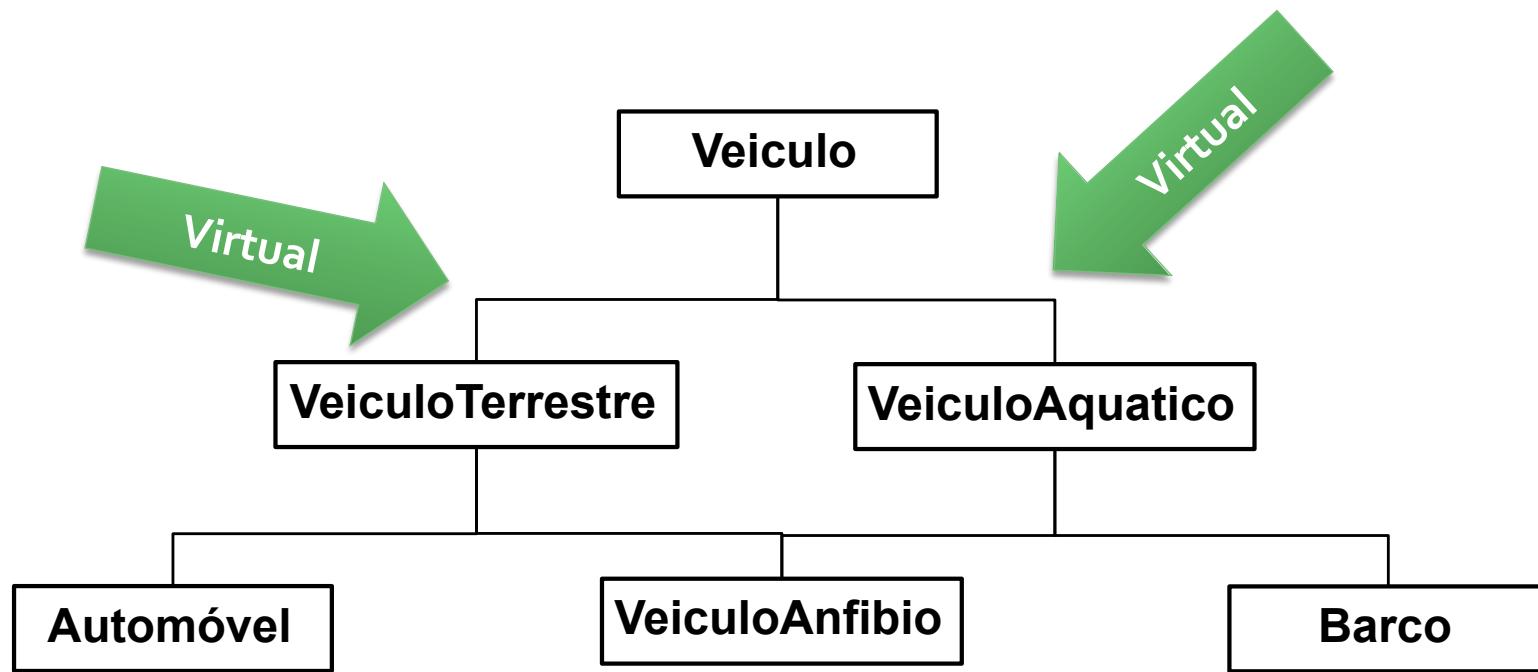
# Classe Base Virtual

- O problema de subobjetos duplicados pode ser resolvido com herança de classe base virtual
  - Quando uma classe base é definida como virtual, apenas um subobjeto aparece na classe derivada.

# Classe Base Virtual

- A única alteração necessária em um código com o problema do diamante é a definição de uma classe base virtual
  - Basta adicionar a palavra **virtual** antes do nome da classe na especificação da herança;
  - A adição deve ser feita um nível antes da herança múltipla
    - Porém, o efeito ocorrerá quando houver herança múltipla.

# Classe Base Virtual



# Classe Base Virtual

- Os cabeçalhos das classes VeiculoTerrestre e VeiculoAquatico passam de

```
class VeiculoTerrestre: public class Veiculo
```

```
class VeiculoAquatico: public class Veiculo
```

- para

```
class VeiculoTerrestre: virtual public class Veiculo
```

```
class VeiculoAquatico: virtual public class Veiculo
```

# Construtores em Herança Múltipla

# Construtores em Herança Múltipla

- No exemplo anterior, as classes base não possuam construtores
  - Por motivos de simplicidade da listagem dos códigos.
- No entanto, caso tivessem, a sintaxe é parecida com a utilizada para herança simples
  - O construtor de cada classe base é chamado explicitamente
    - Sintaxe inicializadora da classe base.
  - Separados por vírgula, e enviando os parâmetros necessários.

# Construtores em Herança Múltipla

- É necessário que a classe derivada tenha um construtor para que os construtores das classes base sejam chamados;
- Se o construtor de uma classe base não for chamado explicitamente, o compilador chamará implicitamente o construtor *default* (sem argumentos) de tal classe
  - Se este não existir, ocorrerá um erro de compilação.
- Vamos supor que nosso exemplo anterior possui o construtor *default* para cada classe base e para a classe derivada.

# Construtores em Herança Múltipla

```
Venda::Venda(): Cadastro(), Imovel(), Tipo(), valor(0.0)
```

```
{  
    //construtor default  
}
```

```
Venda::Venda(string n, string f, string e, string b, float au, float at, int q,  
            string t, float v):
```

```
    Cadastro(n, f), Imovel(e, b, au, at, q), Tipo(t), valor(v)  
{  
    //construtor parametrizado  
}
```

# Na próxima aula

- Arquivos
- Prova
- Polimorfismo
  - Funções Virtuais
  - Resolução Dinâmica
  - Classes Abstratas
    - Funções Virtuais Puras
  - Conversão de Tipos
  - Destrutores Virtuais