

CLASSES DERIVADAS

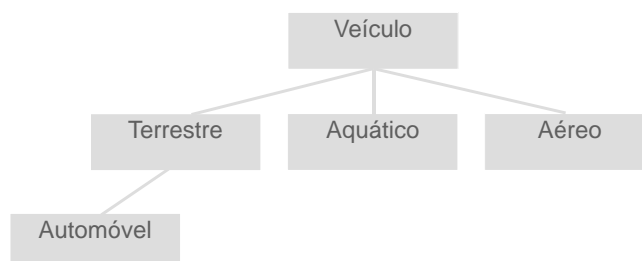
POO 2013/14 DEIS-ISEC

Herança

1

ESCOLHER ENTRE COMPOSIÇÃO OU DERIVAÇÃO ...

- Consideremos um sistema simples de classificação de veículos, representado esquematicamente na figura seguinte.



ESCOLHER ENTRE COMPOSIÇÃO OU DERIVAÇÃO ...

- A classe **Veiculo**, a primeira nesta hierarquia, tem a capacidade de atribuir e consultar o **peso** de um veículo.
- O nível seguinte corresponde às classes **Terrestre**, **Aquatico** e **Aereo**.
- Num percurso vertical, pode ver-se que **Automovel** é um caso particular de veículo **Terrestre**, que, por sua vez é um caso particular de **Veiculo**.
- A classe **Veiculo** representa tudo o que há de comum neste sistema de classificação.

ESCOLHER ENTRE COMPOSIÇÃO OU DERIVAÇÃO ...

- A classe **Veiculo** representa tudo o que há de comum neste sistema de classificação.

Como exemplo, esta classe pode definir-se (com a capacidade de atribuir e consultar o **peso** de um veículo) da seguinte maneira:

```
class Veiculo {
    int peso;
public:
    Veiculo( int p = 0 ) : peso(p) {}
    int getPeso()const{ return peso; }
    void setPeso( int p){ peso = p;}
};
```

ESCOLHER ENTRE COMPOSIÇÃO OU DERIVAÇÃO ...

- ◉ Para representar veículos que se deslocam em terra, pode definir-se uma nova classe, **Terrestre**, com a funcionalidade de **Veiculo**, mas para além disso, com a sua informação e funcionalidade específica.
- ◉ Relativamente aos veículos terrestres, vamos considerar o peso e a velocidade.

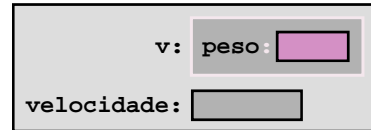
ESCOLHER ENTRE COMPOSIÇÃO OU DERIVAÇÃO ...

- ◉ As classes **Veiculo** e **Terrestre** estão relacionadas.
- ◉ Se for utilizado o mecanismo de composição, a classe **Terrestre** terá um membro do tipo **Veiculo**, uma vez que interessa a informação desta classe.

COMPOSIÇÃO ?

A relação de composição significaria:
um veículo **Terrestre** contém um
Veiculo (?).

objecto da classe Terrestre:



```
class Terrestre {
    Veiculo v;
    int velocidade;
public:
    Terrestre( int p=0, int vel=0):v(p), velocidade(vel){}
    int getPeso()const{ return v.getPeso(); }
    void setPeso( int p){ v.setPeso( p);}
    int getVelocidade()const{ return velocidade; }
    void setVelocidade( int vel){ velocidade = vel;}
};
```

POO 2013/14 DEIS-ISEC

Herança

7

NESTE CASO: DERIVAÇÃO

- ◉ Neste caso, a composição não é adequada, pois um veículo **Terrestre** não contém um **Veiculo**, mas é um caso particular de um veículo.
- ◉ Um veículo **Terrestre** é um caso especial de um **Veiculo**, ou seja tem todas as características de um veículo e ainda as suas características adicionais.
- ◉ A relação mais adequada a este caso é de derivação: a classe **Terrestre** é derivada de **Veiculo** (**Veiculo** é a classe base).

POO 2013/14 DEIS-ISEC

Herança

8

DERIVAÇÃO

Classe derivada

Classe base

```
class Terrestre : public Veiculo {
    int velocidade;
public:
    // ...
};
```

A classe **Terrestre**, sendo derivada da classe **Veiculo**, contém:

- todos os membros variáveis da classe **Veiculo**
- todas as funções membros da classe **Veiculo**.

sub-objecto adquirido por herança

objecto da classe Terrestre:

peso:	
}	
velocidade:	
}	

membros herdados

membros adicionais

Herança

POO 2013/14 DEIS-ISEC
9

DERIVAÇÃO

```
class Terrestre : public Veiculo {
    int velocidade;
public:
    Terrestre( int p = 0, int vel = 0 ):Veiculo(p) {
        velocidade = vel;
    }
    int getVelocidade()const{ return velocidade; }
    void setVelocidade( int vel){ velocidade = vel;}
    void imprime( ostream & saida )const{
        saida << "\nPeso: " << getPeso()
        << "\nVeloc: " << velocidade << endl;
    }
};
```

POO 2013/14 DEIS-ISEC
Herança
10

DERIVAÇÃO

- Os membros privados de **Veiculo**, são privados, mesmo relativamente às funções membros de **Terrestre**, ocupam espaço nos objectos da classe **Terrestre**, mas não são directamente acessíveis nas suas funções membros.
- Na função **imprime()** (membro da classe derivada **Terrestre**), é chamada a função membro **getPeso()**
 - a função **imprime()** não tem acesso directo a **peso** (membro privado da classe base **Veiculo**);
 - a função **imprime()** tem acesso directo a **getPeso()** (membro não privado da classe base **Veiculo**).

POO 2013/14 DEIS-ISEC

Herança

11

DERIVAÇÃO

- Neste exemplo, a classe **Terrestre** é derivada da classe **Veiculo** com o especificador de acesso **public**, pela declaração :

```
class Terrestre : public Veiculo {
    // ...
}
```

- Por esta razão um membro público da classe base é também um membro público da classe derivada. Sendo assim, na função **main()** é possível:

```
void main(){
    Terrestre t(800, 90);
    cout << "\nPeso: " << t.getPeso();
    // ...
}
```

membro público da classe **Veiculo** comporta-se como membro público da classe **Terrestre**

POO 2013/14 DEIS-ISEC

Herança

12

DERIVAÇÃO

- Na definição da classe **Terrestre**, o especificador de acesso **public** poderia ser substituído por **private** ou omitido, por ser a forma de derivação por omissão :

```
class Terrestre : private Veiculo {
    // ...
}
```

ou

```
class Terrestre : Veiculo {
    // ...
}
```

- Neste caso, um membro público da classe base seria um membro privado da classe derivada. Por exemplo, **getPeso()** seria um membro privado da classe **Terrestre**, não sendo acessível em **main()**.

PROTECTED

- Utiliza-se, na classe base, o especificador de acesso **protected**, para conseguir ter membros acessíveis a partir dos membros das classes derivadas, mas inacessíveis a partir do código exterior à classe ou classes derivadas.
- O especificador de acesso **protected** é equivalente ao **private**, com a seguinte exceção: os membros **protected** da classe base são acessíveis a partir dos membros das classes derivadas.
- Para o exterior da classe base ou das classes derivadas os membros **protected** não são acessíveis.
- O especificador de acesso **protected** pode colocar-se em qualquer ponto, na declaração da classe.

FORMAS DE DERIVAÇÃO

- Uma classe também pode ser derivada de outra forma, com o especificador **protected**. Neste caso, os membros **public** e **protected** da classe base são **protected** na classe derivada. Os membros **private** da classe base são sempre inacessíveis na classe derivada.
- Normalmente utiliza-se a forma de derivação **public**.

FORMAS DE DERIVAÇÃO

Forma de derivação	membros que na classe base são:	na classe derivada são:
private	private	inacessíveis
	protected	private
	public	private
protected	private	inacessíveis
	protected	protected
	public	protected
public	private	inacessíveis
	protected	protected
	public	public

EXEMPLO: MEMBROS PROTECTED

```
class Veiculo {
    int peso;
protected:
    int comprimento;
public:
    Veiculo( int p = 0, int c=0): peso (p), comprimento(c){}
    int getPeso()const{ return peso; }
    int getComprimento()const{ return comprimento; }
    void setPeso( int p){ peso = p;}
    void setComprimento( int c){ comprimento = c;}
};
```

POO 2013/14 DEIS-ISEC

Herança

17

```
class Terrestre : public Veiculo {
    int velocidade;
public:
    Terrestre(int p=0,int c=0,int vel=0)
        :Veiculo(p,c) {
        velocidade = vel;
    }

    int getVelocidade()const{ return velocidade; }

    void setVelocidade( int vel){ velocidade = vel;}

    void imprime(ostream & saida )const{
        saida << "Peso: " << getPeso() << endl;
        saida << "Compr: " << comprimento << endl;
        saida << "Veloc: " << velocidade << endl;
    }
};
```

O membro `comprimento`, definido como `protected` na classe base, é acessível na classe derivada.

POO 2013/14 DEIS-ISEC

Herança

18

```

void main(){
    Veiculo v(700);
    cout << "Peso: " << v.getPeso() << endl;
    // cout <<"Compr:" << v.comprimento << endl; ERRO

    Terrestre t(600, 4, 90);
    cout << "Peso: " << t.getPeso() << endl;
    // cout <<"Compr:" << t.comprimento << endl; ERRO
    cout<<"Veloc: "<< t.getVelocidade() << endl;
    t.imprime( cout);
}

```

O membro **comprimento**, definido como **protected** na classe **Veiculo**, é inacessível na função **main()**.

DERIVAÇÃO DE CLASSES E CONSTRUTORES

- ◉ Quando um objecto de uma classe derivada é criado, é chamado o construtor da classe base (que inicializa o sub-objecto adquirido por herança) antes do corpo do construtor da classe derivada ser executado.
 - Se o construtor da classe derivada não chamar explicitamente o construtor da classe base, o sub-objecto adquirido por herança é inicializado pelo construtor por omissão da classe base.
 - Pode explicitar-se a inicialização do sub-objecto adquirido por herança chamando o construtor da classe base com os argumentos correspondentes à inicialização pretendida. Esta chamada é colocada na **lista de inicialização** do construtor da classe derivada.

```

class Veiculo {
    int peso;
protected:
    int comprimento;
public:
    Veiculo( int p = 0, int c=0): peso (p), comprimento(c){}
    // ...
};
class Terrestre : public Veiculo {
    int velocidade;
public:
    Terrestre(int p=0,int c=0,int vel=0)
        :Veiculo(p,c) {
        velocidade = vel;
    }
    // ...
};

```

Chamada ao construtor da classe base na lista de inicialização do construtor da classe derivada.

Chamada ao construtor da classe base: nome da classe base seguido da lista de argumentos.

```

class Veiculo {
    int peso;
protected:
    int comprimento;
public:
    Veiculo( int p = 0, int c=0): peso (p), comprimento(c){}
    // ...
};
class Terrestre : public Veiculo {
    int velocidade;
public:
    Terrestre(int p=0,int c=0,int vel=0)
        :Veiculo(p,c), velocidade (vel) {}
    // ...
};

```

A lista de inicialização pode incluir a inicialização do sub-objecto adquirido por herança e também a inicialização de membros da classe.

DERIVAÇÃO DE CLASSES E DESTRUTORES

- Quando um objecto de uma classe derivada é destruído, são chamados os destrutores por ordem inversa da ordem de derivação: primeiro o destrutor da classe derivada, depois o destrutor da classe base (ao contrário do que se passa com a construção).
- Este comportamento de construtores e destrutores desencadearem a execução das versões correspondentes das classes base é especial destas funções. Quando uma função membro "normal" é chamada, apenas é executada uma versão (embora possa haver mais versões).

POO 2013/14 DEIS-ISEC

Herança

23

```

class Veiculo {
    int peso;
public:
    Veiculo( int p = 0 ) : peso(p) {}
    int getPeso()const{ return peso; }
    void setPeso( int p){ peso = p;}
    void imprime( ostream & saida) const{
        saida << "Peso: " << peso << endl;
    }
};

class Terrestre : public Veiculo {
    int velocidade;
public:
    Terrestre( int p = 0, int vel = 0):Veiculo(p),
    velocidade(vel){}
    int getVelocidade()const{ return velocidade; }
    void setVelocidade( int vel){ velocidade = vel;}
    void imprime( ostream & saida )const{
        Veiculo::imprime( saida );
        saida << "Veloc: " << velocidade << endl;
    }
};

```

Na definição da função `imprime()` da classe `Terrestre`, para chamar a versão da função `imprime()` da classe `Veiculo` tem que se explicitar que é a versão da classe `Veiculo` através do operador de contexto `Veiculo::imprime(saida);`

Se se chamasse simplesmente `imprime(saida);` seria uma chamada recursiva da versão local.

Redefinição da função `imprime()`

```
void main(){
    Veiculo v(700);
    v.imprime(cout);
    cout << endl;

    Terrestre t(800, 90);
    t.imprime( cout );
}
```

Saída:

```
Peso: 700
Peso: 800
Veloc: 90
```

- A função **imprime()** na classe derivada **Terrestre** redefine a função **imprime()** da classe base **Veiculo**.
- Quando se chama a função **getPeso()** para um objecto da classe **Terrestre**, funciona a versão existente na classe **Veiculo**.
- Quando se chama a função **imprime()** para um objecto da classe **Veiculo**, funciona a versão existente na classe **Veiculo**.
- Quando se chama a função **imprime()** para um objecto da classe **Terrestre**, funciona a versão redefinida na classe **Terrestre**.

POO 2013/14 DEIS-ISEC Herança 25

COMPOSIÇÃO E HERANÇA CONJUNTAMENTE

- É possível definir uma classe utilizando conjuntamente composição e herança. Quando é criado um objecto de uma classe derivada de outra e com membros que são objectos de classes:
 - é chamado primeiro o construtor da classe base,
 - em seguida são chamados os construtores dos membros que são objectos
 - e só depois é executado o corpo do construtor da classe a que pertence o objecto que está a ser criado.
- Os destrutores são chamados por ordem inversa dos construtores.
- No exemplo seguinte está definida uma classe **Derivada**, que deriva da classe **Base** e tem um membro inteiro e outro da classe **Membro**.

Composição e herança conjuntamente:
ordem das chamadas de construtores e destrutores

```

class Base {
    int i;
public:
    Base(int ii = 0) : i(ii) { cout << "Base(int ii)\n"; }
    ~Base(){ cout << "~Base()\n"; }
    void imprime( ostream & saida) const {
        saida << "Base: i=" << i << endl; }
};

class Membro {
    int k;
public:
    Membro(int kk = 0) : k(kk) { cout << "Membro(int kk)\n"; }
    ~Membro(){ cout << "~Membro()\n"; }
    void imprime( ostream & saida) const {
        saida << "Membro: k=" << k << endl; }
};

```

POO 2013/14 DEIS-ISEC

Herança

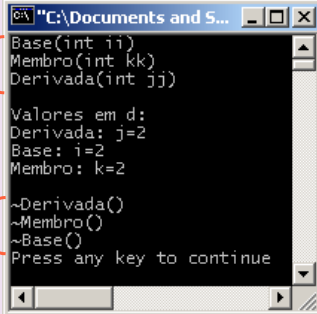
27

```

class Derivada : public Base {
    int j;
    Membro m;
public:
    Derivada(int jj): Base(jj), j(jj), m(jj) {
        cout << "Derivada(int jj)\n";
    }
    ~Derivada(){ cout << "~Derivada()\n"; }
    void imprime( ostream & saida) const {
        saida << "Derivada: j=" << j << endl;
        Base::imprime(saida);
        m.imprime(saida);
        saida << endl;
    }
};

void main() {
    Derivada d(2);
    cout << "\nValores em d:\n";
    d.imprime(cout);
    // chamada ao destrutor para d
}

```



Output from the command prompt:

```

Base(int ii)
Membro(int kk)
Derivada(int jj)

Valores em d:
Derivada: j=2
Base: i=2
Membro: k=2

~Derivada()
~Membro()
~Base()
Press any key to continue

```

POO 2013/14 DEIS-ISEC

Herança

28

```

class Base { /* a mesma versão do exemplo anterior */};
class Membro { /* a mesma versão do exemplo anterior */};

class Derivada : public Base {
    Membro m;
public:
    void imprime( ostream & saida) const {
        saida << "Derivada: \n";
        Base::imprime(saida);
        m.imprime(saida);
        saida << endl;
    }
};

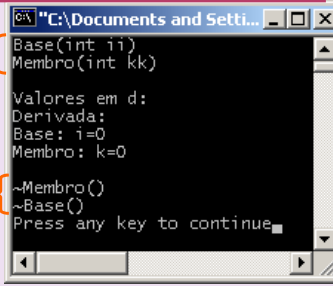
void main() {
    Derivada d;
    cout << "\nValores em d:\n";
    d.imprime(cout);
    // destruição de d
}

```

Como na classe Derivada não está definido explicitamente nenhum construtor, é gerado um construtor por omissão. Como também não está definido explicitamente um destrutor é gerado automaticamente um destrutor.

O construtor por omissão gerado automaticamente chama o construtor por omissão da classe base e o construtor por omissão da classe a que pertence o membro objecto.

O destrutor gerado automaticamente chama o destrutor da classe a que pertence o membro objecto e o destrutor da classe base.



```

C:\Documents and Settings\...>
Base(int ii)
Membro(int kk)

Valores em d:
Derivada:
Base: i=0
Membro: k=0

~Membro()
~Base()
Press any key to continue

```

POO 29

FUNÇÕES COM TRATAMENTO ESPECIAL RELATIVAMENTE À HERANÇA

- ◉ **Construtores:**
um construtor de uma classe derivada chama explicitamente ou implicitamente o construtor da classe base correspondente.
- ◉ **Construtor por cópia:**
no construtor por cópia de uma classe derivada é preciso chamar explicitamente o construtor por cópia da classe base.
- ◉ **Destrutores:**
um destrutor de uma classe derivada chama o destrutor da classe base correspondente.
- ◉ **Operador atribuição:**
a atribuição entre dois objectos de uma classe pode não ser suficiente para fazer a atribuição entre dois objectos de uma classe derivada. O operador atribuição não é herdado normalmente como pode ver-se no exemplo seguinte.

```

class Base {
    int i;
public:
    Base(int ii) : i(ii) {}
    Base(const Base & b) : i(b.i) {
        cout << "Base(const Base &)\n";
    }
    Base & operator=(const Base & b){
        i = b.i;
        cout << "Base::operator=()\n";
        return *this;
    }
    void imprime( ostream & saida)const{
        saida << "Base: i=" << i << endl;
    }
};

```

Composição e herança conjuntamente:
construtor por cópia e operador atribuição
gerados automaticamente na classe
derivada/composta

```

class Membro {
    int k;
public:
    Membro(int kk) : k(kk) {}
    Membro(const Membro& m) : k(m.k) {
        cout << "Membro(const Membro&)\n";
    }
    Membro & operator=(const Membro & m){
        k = m.k;
        cout << "Membro::operator=()\n";
        return *this;
    }
    void imprime( ostream & saida)const{
        saida << "Membro: k=" << k << endl;
    }
};

```



```

class Derivada : public Base {
    int j;
    Membro m;
public:
    Derivada(int jj) : Base(jj), j(jj), m(jj) {}

    void imprime( ostream & saida) const {
        saida << "Derivada: j=" << j << endl;
        Base::imprime(saida);
        m.imprime(saida);
    }
};

```

Na classe **Derivada** não estão definidos explicitamente construtor por cópia nem operador atribuição. Por esta razão estas funções são geradas automaticamente pelo compilador.

POO 2013/14 DEIS-ISEC

Herança

33

```

void main() {
    Derivada d(2);
    cout << "\nConstrução por copia : " << endl;
    Derivada d2 = d; // construção por copia
    cout << "\nValores em d2:\n";
    d2.imprime(cout);
    Derivada d3(3);
    cout << "\nAtribuição: " << endl;
    d3 = d; // atribuição

    cout << "\nValores em d3:\n";
    d3.imprime(cout);
}

```

O construtor por cópia gerado pelo compilador chama o construtor por cópia da classe base e o construtor por cópia do membro objecto.

O operador atribuição gerado pelo compilador chama o operador atribuição da classe base e o operador atribuição do membro objecto.

```

"C:\Documents and Settings\...\
Construcao por copia:
Base(const Base &)
Membro(const Membro&)

Valores em d2:
Derivada: j=2
Base: i=2
Membro: k=2

Atribuição:
Base::operator=()
Membro::operator=()

Valores em d3:
Derivada: j=2
Base: i=2
Membro: k=2
Press any key to continue.

```

POO 2013/14 DEIS-ISEC

Herança

34

HERANÇA, CONSTRUTOR POR CÓPIA E OPERADOR ATRIBUIÇÃO

- ◉ No exemplo anterior, na classe **Derivada** não está definido construtor por cópia nem operador atribuição. Por esta razão estas funções são geradas automaticamente pelo compilador.
- ◉ O construtor por cópia gerado pelo compilador chama o construtor por cópia da classe base e o construtor por cópia do membro objecto.
- ◉ O operador atribuição gerado pelo compilador chama o operador atribuição da classe base e o operador atribuição do membro objecto.

POO 2013/14 DEIS-ISEC

Herança

35

HERANÇA, COMPOSIÇÃO E CONSTRUTOR POR CÓPIA

- ◉ O exemplo seguinte é uma outra versão da classe **Derivada**, com construtor por cópia e operador atribuição explícitos.
 - ◉ O construtor por cópia da classe **Derivada** tem:
 - uma chamada explícita ao construtor por cópia da classe **Base** para inicializar o sub-objecto adquirido por herança e também
 - uma chamada explícita ao construtor por cópia da classe **Membro** para inicializar o membro objecto.
- ```
Derivada(const Derivada& d) : Base(d), m(d.m), j(d.j) {
```
- ◉ Se não se explicitassem estas chamadas na lista de inicialização do construtor por cópia, seriam chamados em seus lugares os correspondentes construtores por omissão. O sub-objecto adquirido por herança e os membros objectos seriam inicializados pelos construtores por omissão em vez de receberem cópias dos correspondentes sub-objectos do objecto da classe **Derivada** considerado origem da cópia.

POO 2013/14 DEIS-ISEC

Herança

36

## HERANÇA, COMPOSIÇÃO E OPERADOR ATRIBUIÇÃO

- ◉ O operador atribuição da classe **Derivada** inclui:
  - uma chamada explícita ao operador atribuição da classe **Base** para atribuir o sub-objecto adquirido por herança e também
  - atribuições dos membros objectos.

```
Base::operator =(d);
m = d.m;
```

- ◉ Se não se explicitassem estas chamadas não seria feita atribuição aos sub-objects do objecto da classe **Derivada**.

POO 2013/14 DEIS-ISEC

Herança

37

```
class Base { /* a mesma versão do exemplo anterior */};
class Membro { /* a mesma versão do exemplo anterior */};
class Derivada : public Base {
 int j;
 Membro m;
public:
 Derivada(int jj) : Base(jj), j(jj), m(jj) {}
 Derivada(const Derivada & d) : Base(d), m(d.m), j(d.j) {
 cout << "Derivada(const Derivada&)\n";
 }
 Derivada & operator=(const Derivada & d){
 Base::operator =(d);
 m = d.m;
 j = d.j;
 cout << "Derivada::operator=()\n";
 return *this;
 }
 void imprime(ostream & saida)const{
 saida << "Derivada: j=" << j << endl;
 Base::imprime(saida);
 m.imprime(saida);
 }
};
```

Chamada ao construtor por cópia da classe **Base**. A conversão de **Derivada&** em **Base&** (*upcasting*) é automática.

Chamada ao construtor por cópia do membro objecto.

Chamada ao operador atribuição da classe **Base**.

Atribuição do membro objecto.

POO 2013/14 DEIS-ISEC

Herança

38

```

void main() {
 Derivada d(2);
 cout << "\nChamada ao construtor por copia: " << endl;
 Derivada d2 = d; // chama o construtor por copia
 cout << "\nValores em d2:\n";
 d2.imprime(cout);
 Derivada d3(3);
 cout << "\nChamada ao operador atribuicao: " << endl;
 d3 = d; // atribuicao

 cout << "\nValores em d3:\n";
 d3.imprime(cout);
}

```

```

C:\Documents and Settings\mcorreia\...
Chamada ao construtor por copia:
Base(const Base &)
Membro(const Membro&)
Derivada(const Derivada&)

Valores em d2:
Derivada: j=2
Base: i=2
Membro: k=2

Chamada ao operador atribuicao:
Base::operator=(<)
Membro::operator=(<)
Derivada::operator=(<)

Valores em d3:
Derivada: j=2
Base: i=2
Membro: k=2
Press any key to continue

```

POO 2013/14 DEIS-ISEC

Herança

39

## HERANÇA E FUNÇÕES OVERLOADED

- Se uma função for redefinida numa classe derivada, todas as outras funções com o mesmo nome que possam existir na classe base, ficam indisponíveis na classe derivada.
- Se numa classe derivada for definida uma função com o mesmo nome de outra função da classe base, mas com protótipo diferente, na classe derivada ficam indisponíveis todas as funções da classe base com esse nome.
- O exemplo seguinte ilustra estas situações.

POO 2013/14 DEIS-ISEC

Herança

40

|                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> class Base { public:     int f() const { return 1; }     int f(string) const { return 1; }     void g() {} };  class Deriv1 : public Base { public:     void g() const {} };  class Deriv2 : public Base { public:     // Redefinicao:     int f() const { return 2; } };  class Deriv3 : public Base { public:     // Altera tipo de retorno:     void f() const { } }; </pre> | <div data-bbox="531 297 1246 331">Exemplo adaptado de <b>Thinking in C++</b>, 2nd Edition, Bruce Eckel 2000</div> <pre> class Deriv4 : public Base { public:     // Altera lista de argumentos:     int f(int) const { return 4; } };  void main() {     string s("Ola");     Deriv1 d1;     int x = d1.f();     d1.f(s);     Deriv2 d2;     x = d2.f();     //! d2.f(s); //versao indisponivel     Deriv3 d3;     d3.f();     //! x = d3.f(); //versao indisp.     Deriv4 d4;     //! x = d4.f(); //versao indisp.     x = d4.f(1); } ///:~ </pre> |
| POO 2013/14 DEIS-ISEC                                                                                                                                                                                                                                                                                                                                                                 | Herança 41                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |