PICC
Programação Orientada por Objectos em C++
Parte 2

Fernando Miguel Carvalho

Secção de Programação

ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

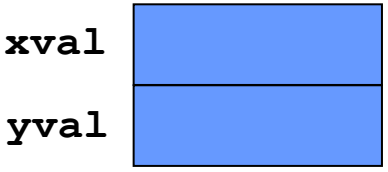
Tópicos

- Herança múltipla
  - Disposição dos objectos
- Problemas da herança múltipla
  - Problema do diamante
  - Derivação virtual
- Informação de *runtime* (RTTI)
  - Conversão de tipos

Disposição dos objectos em memória

- Recordemos a disposição de um objecto em memória

```
class Point2D
{
    int xval;
    int yval;
public:
    Point2D(int x, int y);
    int getXCoord();
    int getYcoord();
};
```

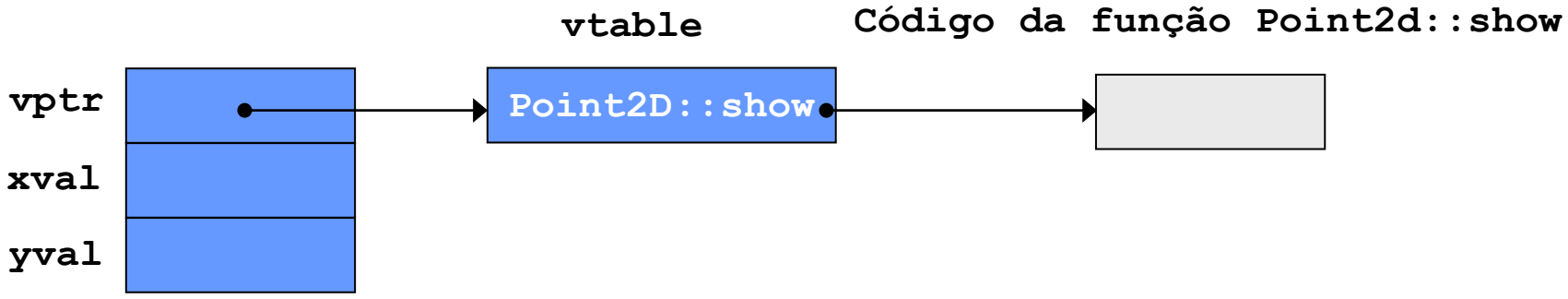


Um objecto de um tipo sem métodos virtuais, não tem tabela de métodos virtuais associada (vtable), nem informação de tipo em *runtime* (RTTI)

Disposição dos objectos em memória

- Recordemos a disposição de um objecto em memória

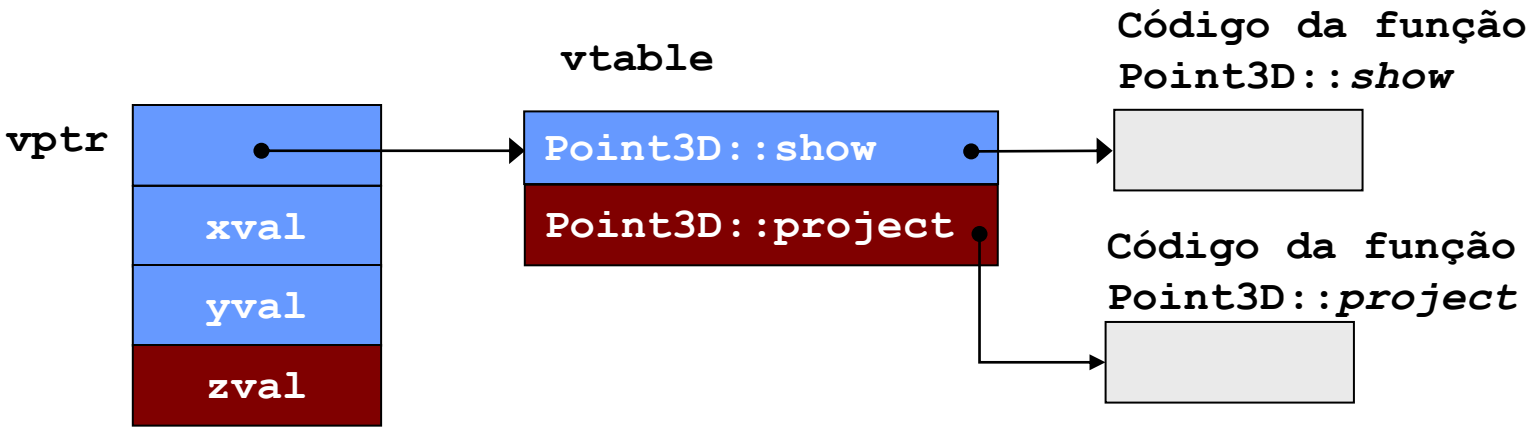
```
class Point2D
{
    int xval;
    int yval;
public:
    Point2D(int x, int y);
    virtual void show(ostream&);
    int getXCoord();
    int getYcoord();
};
```



Disposição dos objectos em memória II

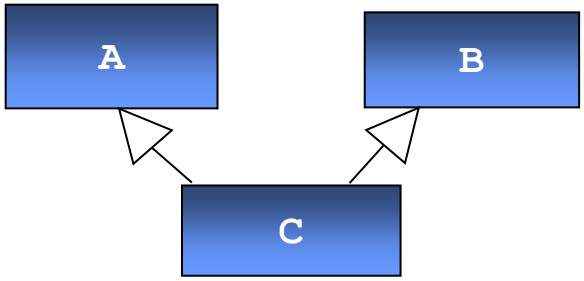
```
class Point3D: public Point2D
{
    int xval;
    int yval;
    int zval;

public:
    Point3D(int x, int y, int z);
    void show(ostream&);
    virtual Point2d project(int eixo);
};
```



Herança múltipla

- Existem casos onde é necessário que uma classe herde funcionalidade e campos de duas hierarquias distintas



Que problemas podem surgir com a derivação múltipla?

Que alterações são necessárias na disposição dos objectos em memória?

Herança múltipla – Colisão de nomes

- Dada a definição das classes A e B

```
class A
{
   int field_a;
public:
    virtual void fa();
    virtual void fc();
};
```

```
class B
{
    int field_b;
public:
    virtual void fb();
    virtual void fc();
};
```

Problemas?

```
class C: public A, public B
{
...
};

C* c_ptr=...;
c_ptr->fc(); //erro. f() de A ou de B?
```

Existe uma colisão de nomes nas classes base. Como resolver essa colisão?

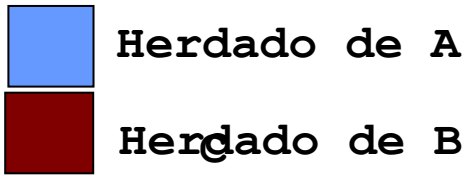
Herança múltipla – Colisão de nomes II

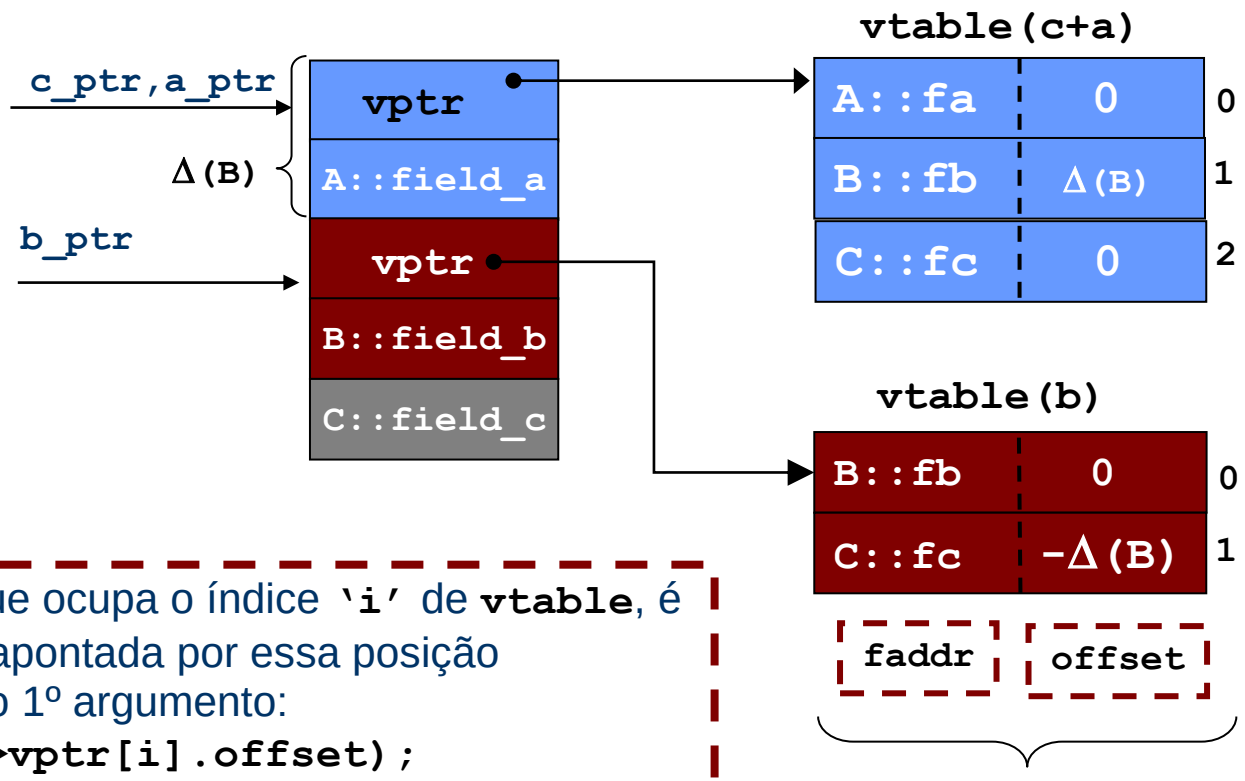
- Uma possível solução passa por revolver essa ambiguidade, implementando o método em C

```
class C: public A, public B
{
  int field_c;
  public:
    void f(){A::fc();}
};
```

Herança múltipla – disposição em memória

```
C* c_ptr = new C();
A* a_ptr = (A) c_ptr;
B* b_ptr = (B) c_ptr;
```





Chamar um método: **p->f()** que ocupa o índice **'i'** de **vtable**, é equivalente a chamar a função apontada por essa posição passando o **this** correcto como 1º argumento:

```
p->vptr[i].faddr(p + p->vptr[i].offset);
```

this

Cada elemento da **vtable** tem o endereço da função e o **offset** de acerto do **this**.

- Ex: **c_ptr->**fb() ⇔ c_ptr->fb(c_ptr + Δ(B))

```
c_ptr->vptr[1].faddr(c_ptr + c_ptr->vptr[1].offset);
```

- Ex: **b_ptr->**fc() ⇔ b_ptr->fc(b_ptr - Δ(B))

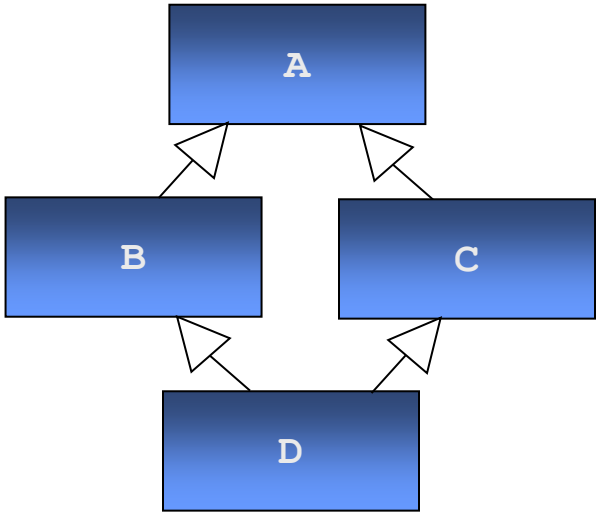
```
b_ptr->vptr[1].faddr(b_ptr + b_ptr->vptr[1].offset);
```

Herança múltipla – disposição em memória

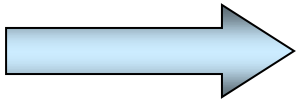
```
class Derivada: Base 0, Base 1, ...,Base n-1 {...}
```

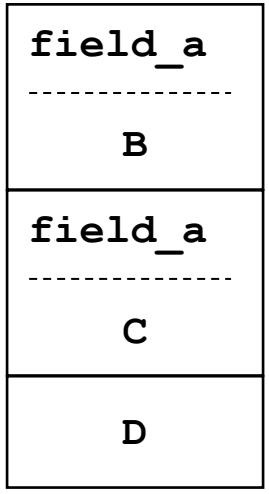
- Existem **n** *vtables*, sendo **n** o número de classes base
  (Ex: herança simples requere 1 *vtable*)
  - A **tabela 0** tem métodos virtuais da 1ª **classe base** + das **restantes classes base** + da **classe derivada**, tal como na herança simples.
    - nesta tabela, os métodos das classes base **1** a **n-1**, têm um offset diferencial (Δ) para a respectiva tabela.
  - cada **tabela auxiliar** (de 1 a n-1) tem apenas os métodos declarados pela respectiva classe base.
    - nesta tabela, os métodos redefinidos têm um offset diferencial (Δ) para o início do objecto.
- Um ponteiro do tipo base 0 ou do tipo derivado, aponta para o slot que tem a tabela 0.
- Um ponteiro do tipo base 1… n-1, aponta para o slot que tem a respectiva tabela.

Herança múltipla – duplicação de membros




```
class A { public: int field_a;};
class B: public A {};
class C: public A {};
class D: public B, public C {};
```

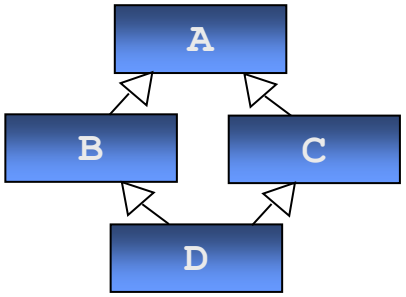




- Existe ambiguidade na definição de D, uma vez que existe uma duplicação do campo `field_A` (um herdado através de B e outro através de C)
- Esta situação é conhecida pelo problema do diamante

Herança múltipla – duplicação de membros II

- Além disso poderá existir uma ambiguidade nas conversões entre apontadores

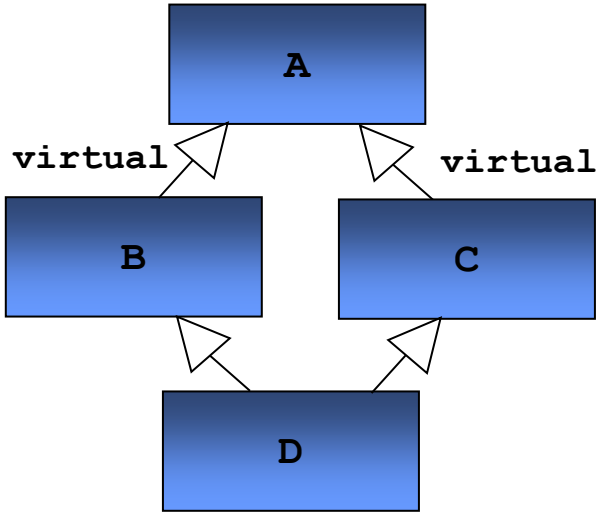



```
D* d_ptr=new D();
A* a_ptr= d_ptr; //ambíguo
a_ptr= (A*)d_ptr; //ambíguo
a_ptr= (A*)(C*)d_ptr; //OK, mas pouco elegante
```

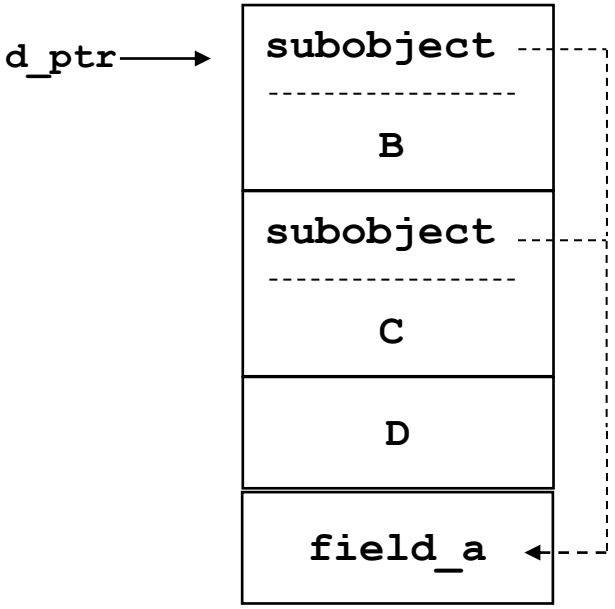
- É possível alterar esta situação se alterarmos a derivação de B e C, passando a ser virtual, ou seja:

```
class A { public: int field_a;};
class B: virtual public A {};
class C: virtual public A {};
class D: public B, public C {};
```

Herança múltipla – derivação virtual




```
D* d_ptr=new D();
```



- Cada objecto ***B*** e ***C*** terá o seu sub objecto ***A***
- No entanto, ***D*** apenas terá uma cópia de ***A***
- Como a posição de ***A*** não é a mesma em todos os objectos, terá de se utilizar um apontador para ***A*** em cada objecto onde ***A*** é o objecto da classe base virtual

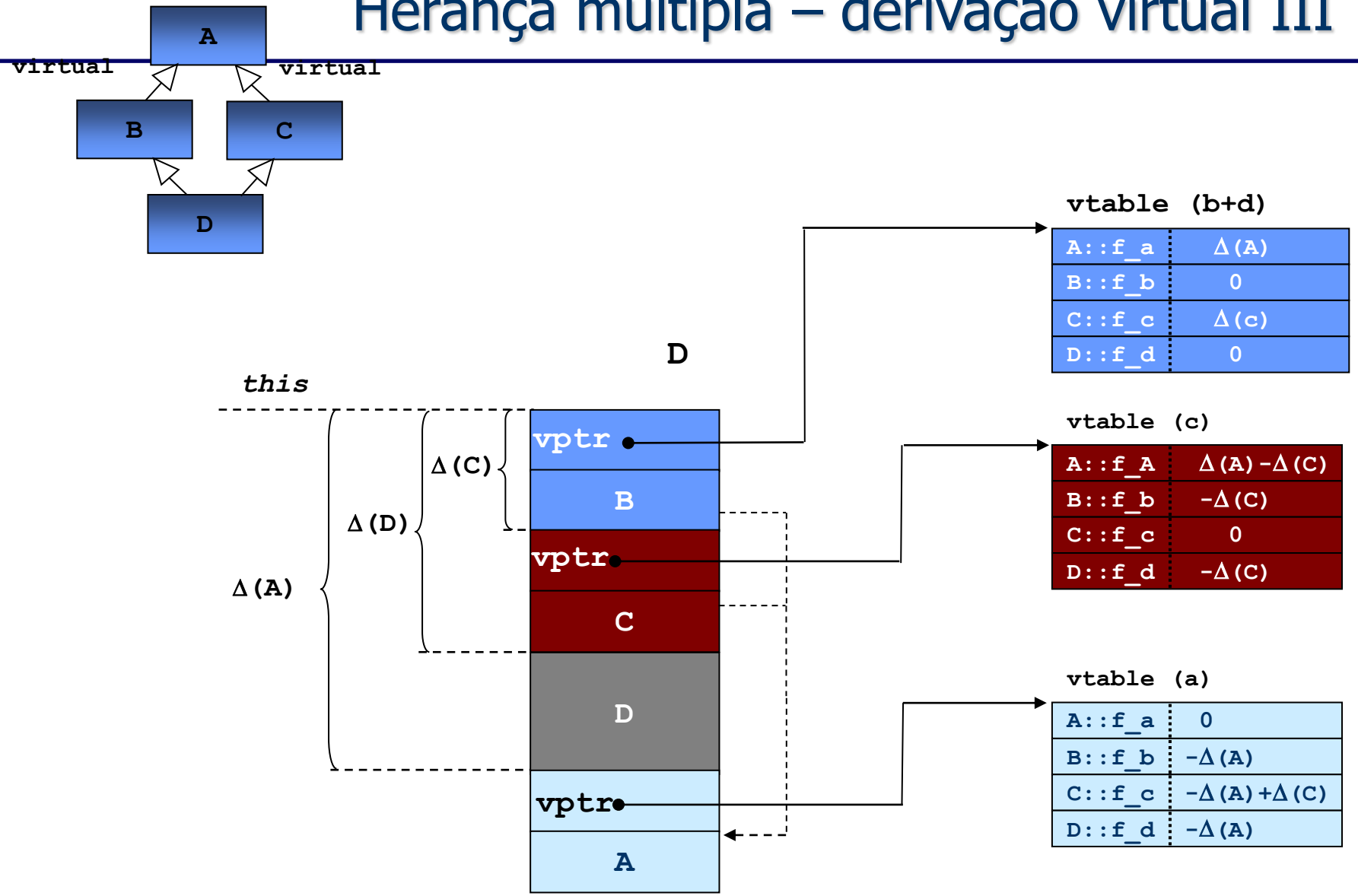
Herança múltipla – derivação virtual II

- Consideremos então a seguinte definição de cada uma das classes

```
class A {
  public: int i;
          virtual void f_a();
          virtual void f_b();
          virtual void f_c();
          virtual void f_d();};
class B: virtual public A {void f_b();};
class C: virtual public A {void f_c();};
class D: public B, public C {void f_d();};
```

- Note-se que nunca existe uma redefinição do mesmo método virtual em B e C, sem também existir simultaneamente em D
- Dessa forma evita-se ambiguidades
- Deve-se garantir que redefinições de métodos de uma classe base virtual devem ocorrer num único caminho das folhas à raiz da hierarquia

Herança múltipla – derivação virtual III



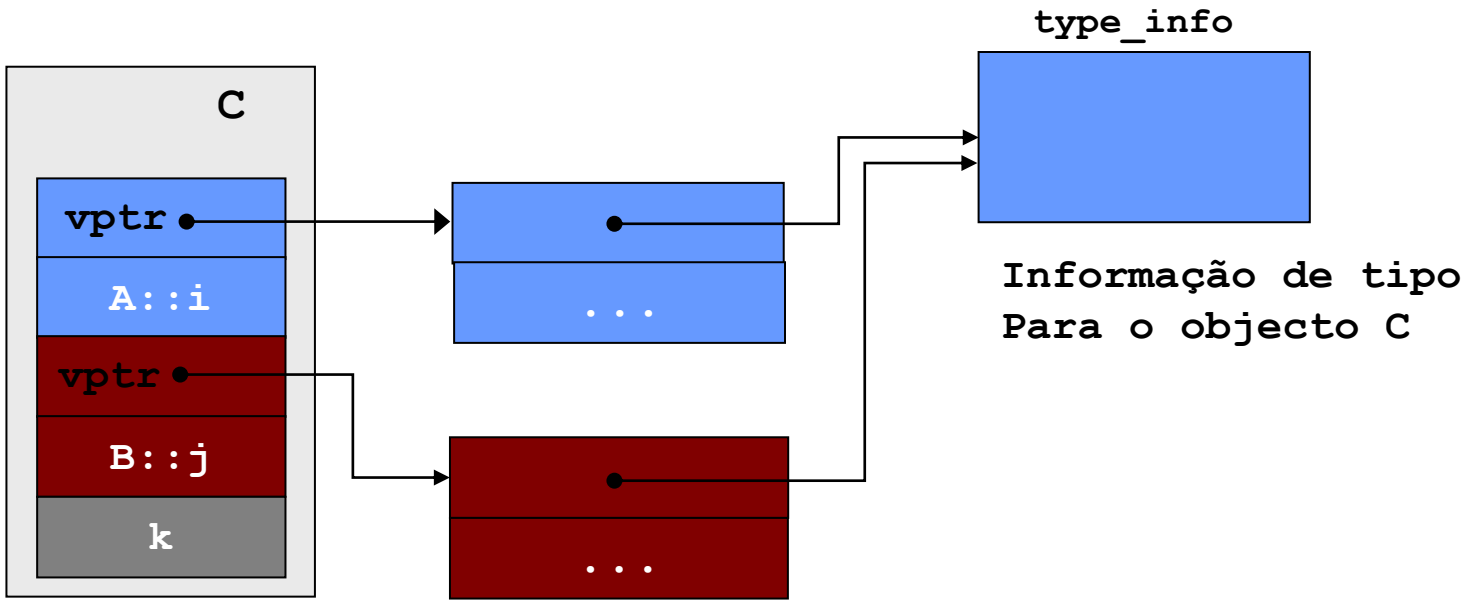
Informação de *runtime* (RTTI)
Conversão de tipos

ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Runtime Type Information - RTTI

- Para que seja possível efectuar conversões das classes bases para as classes derivadas com segurança, é necessário que seja possível determinar em cada momento qual o tipo real que um apontador "aponta"
- O suporte a essa necessidade veio trazer um "overhead" adicional em termos de espaço e tempo de execução
  - É necessário guardar em cada objecto um conjunto de informação sobre o seu tipo
  - É necessário em tempo de execução aceder a essa informação para determinar o tipo realmente apontado
- A sua utilização deve ser utilizada apenas quando é necessário
- Muitas das vezes o seu uso resulta de uma má estruturação do código, pouco OO

Runtime Type Information – RTTI III



Runtime Type Information – RTTI II

- Note-se que este mecanismo apenas permite *downcast* seguro para tipos com polimorfismo
- Como as classes onde existem métodos virtuais já contém uma tabela de métodos virtuais, pode-se utilizá-las para guardar mais um apontador para ser usado pelo mecanismo de *RTTI*
- Normalmente, o primeiro apontador de uma tabela de métodos virtuais é utilizado para isso.

typeid

```
typeid( type-id ) ou typeid( expression )
```

```
Ex:
Point * ptr = new Point();
const type_info& typePoint = typeid(Point);
const type_info& typeOfPtr = typeid(*p); // equals to typePoint
```

- O operador **`typeid`** permite determinar o tipo de um objecto em tempo de execução.
- O resultado do **`typeid`** é um **`const type_info&`**.
- O resultado é uma referência para um objecto **`type_info`** que representa ou o **`type-id`** ou a **`expression`** passada como parâmetro.

typeid... para cast seguro

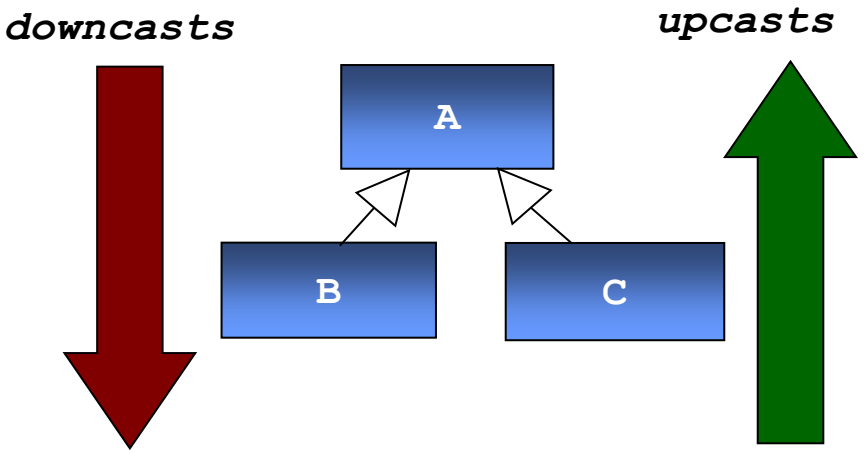
```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
Square * CastToSquare(Shape * s){
  Square * sq = NULL;
  if(typeid(*s) == typeid(Square))
    sq = static_cast<Square*>(s);
  return sq;
}
int main(){
  Shape * s1 = new Circle();
  Shape * s2 = new Square();
  printf("s1: %p (Circle)\n", s1);
  printf("s2: %p (Square)\n", s2);
  Square * sq;
  printf("sq: %p (lixo)\n", sq);
  sq = CastToSquare(s1);
  printf("sq: %p (NULL)\n", sq);
  sq = CastToSquare(s2);
  printf("sq: %p (Square)\n", sq);
}
```

```
s1: 00323148 (Circle)
s2: 00323158 (Square)
sq: 0012FF80 (lixo)
sq: 00000000 (NULL)
sq: 00323158 (Square)
```

Conversão de tipos

- Numa hierarquia de classes, é necessário ter cuidado com as conversões de tipo, uma vez que nem todas são possíveis

Cuidado.
Conversões de tipos neste sentido podem ser inválidas.
e.g. Nem todos os A's são B's



Conversões de tipos neste são sempre válidas.
e.g. Os B's são sempre A's

Com derivação pública

Conversão de tipos II

- Além da típica conversão ((t)x), utilizada de forma estática, herdada do C, existem agora operadores para conversão de tipos, dos quais se destacam:
  - static_cast<*tipo*> (expressão)
  - reinterpret_cast<*tipo*> (expressão)
  - dynamic_cast<*tipo*> (expressão)
    - Permite efectuar *downcasts* com segurança!!!

Conversão de tipos – static e reinterpret_cast

- static_cast<T> (v)
  - Funciona de forma semelhante a ***(T)v***
  - Ou seja, para que o resultado seja correcto ***v*** tem de ser um subtipo de ***T*** ou existir uma conversão implícita do tipo de ***v*** para ***T***
- reinterpret_cast<T> (v)
  - Faz conversões entre tipos não relacionados, e.g. *int* e *pointer*
  - Não existe garantia da validade dos valores depois da conversão
  - Perigoso…
- dynamic_cast<T> (v)
  - Suportado pela existência da RTTI

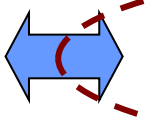
dynamic_cast<T> (v)

```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};

Square * CastToSquare(Shape * s){
  Square * sq = NULL;
  if(typeid(*s) == typeid(Square))
    sq = static_cast<Square*>(s);
  return sq;
}
```

```
Shape * s1 = new Circle();
Shape * s2 = new Square();
Square * sq;
```

```
sq = CastToSquare(s1);
```



```
sq = dynamic_cast<Square*>(s2);
```