Héritage multiple (rappel des principes)

Une classe peut dériver d'un nombre quelconque de classes de bases

- mais pas plusieurs fois (directement) de la même classe

```
class A { … };
class B { … };
class C : public A { … };
class D : public A, B, protected C { … };
```

Héritage privé de B (défaut)

- en cas de plusieurs héritages (indirects) de la même classe, plusieurs copies de la classe de base dans la classe dérivée

Membres de D
Membres de C
Membres de A
Membres de B
Membres de A

Héritage multiple (ambiguïté des noms d'attributs)

Les membres des classes de base peuvent avoir les même noms

- Utiliser l'opérateur de portée :: pour lever l'ambiguïté :

```
class A { … protected: int x; };
class B { … protected: int x; };
class C : public A, public B {
 …
 void M (…) { A::x = B::x; }
};
```

Héritage multiple (ambiguïté des noms de méthode)

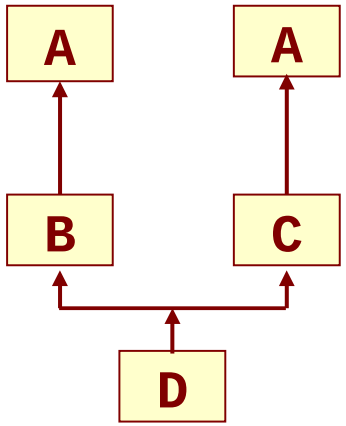
La résolution de surcharge n'intervient qu'après la résolution de nom

```
class A {
 public: void M (int, int); …
};
class B {
 public: void M (int); …
};
class C : public A, public B { … };
C c;
c.M(3);     // NON: A::M ou B::M ?
c.B::M(3); // OK
```

Héritage multiple
(ambiguïté des conversions)

En cas d'héritage répété, pas de conversion implicite:

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




```
D* p = new D;
A* pa = p; // KO, quel A ?
A* pc = static_cast<C*>(p); // OK
D* q = static_cast<D*>(pc); // NON,
                            // de quel A ?
```

Héritage multiple (méthodes virtuelles)

Problèmes des méthodes virtuelles des classes de base qui ont même signature :

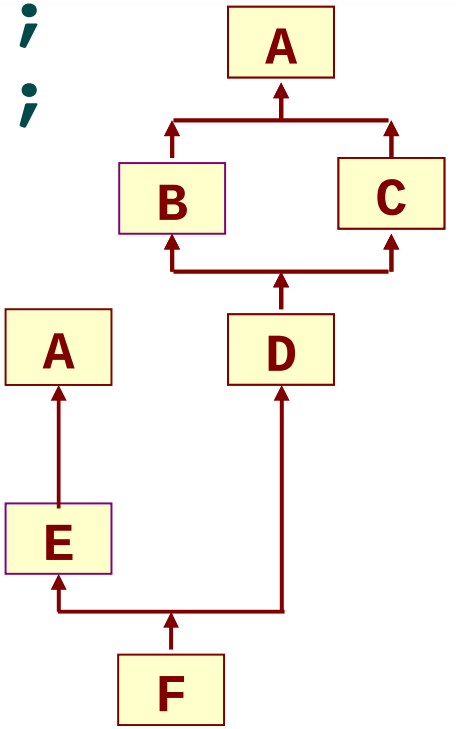
```
class A { public: virtual void M(){…} … };
class B : public A { public: void M(){…} …};
class C : public A { public: void M(){…} …};
class D : public B, public C { … };

A* p = new D(); p->M(); // B::M ou C::M ?
```

- donc redéfinir `M` dans D
- Si `C` n'a pas de méthode `M`, pour `p->M()`, `B::M` est appelée car `B::M` *domine* `A::M`

Héritage multiple
(dérivation virtuelle)

Si héritage virtuel, classe de base non dupliquée:

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



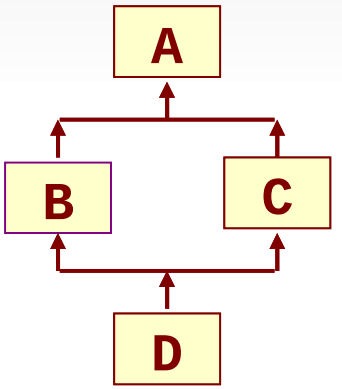
Mais attention

```
class E : public A {…};
class F : public D,
          public E {…};
```

Héritage multiple (construction & dérivation virtuelle)

La classe de base virtuelle est construite en premier, et une seule fois:

```
class A { public: A(…); … };
class B : public virtual A {
  public: B(…): A(…) {}
  …
};
class C : public virtual A {
  public: C(…): A(…) {}
…
};
class D : public B, public C {
  public: D(…): A(…), B(…), C(…) {…}
  …
};
```

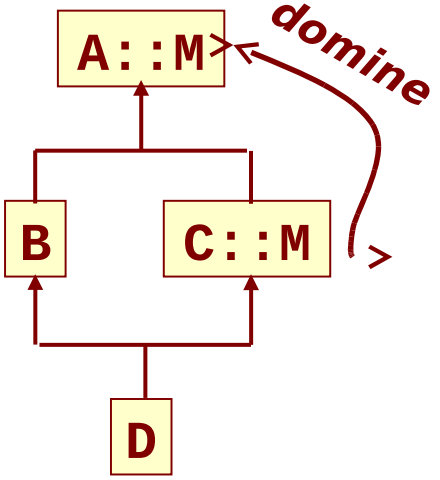


B et C ne construirons pas A de nouveau

Héritage multiple (dérivation virtuelle & liaison dynamique)

La méthode qui *domine* est appelée

```
class A {
    public: virtual void M() {}
    …
};
class B : public virtual A {
    … // ne redéfinit pas M
};
class C : public virtual A {
    public: void M(…) {}
    …
};
class D : public B, public C { … };

A* p = new D; … p->M(); // ⇒ C::M
```



Implémentation des classes et de l'héritage

Supposons les classes

```
class A {
int x;
};
```

```
class B : public A {
int a;
};
```

Leur traduction en C pourrait être

```
typedef struct A {
int x;
} A;
```

```
typedef struct B {
   A _parente;
   int a;
} B;
```

Implémentation des classes et de l'héritage virtuel

Supposons les classes

```
class A { … };
```

```
class B : virtual A { … };
```

```
class C : virtual A { … };
```

```
class D : public B, public C { … };
```

Leur traduction en C pourrait être

```
typedef struct A {…} A;
```

```
typedef struct BsansA {
 A* _parenteA; … // attr. De B
} B;
```

```
typedef struct B {
 A _parenteA;
 BsansA _ptrA;
} B;
```

Idem pour C

```
typedef struct D {
 A _parenteA;
 BsansA _parenteB; CsansA _parenteC; …
} C;
```

Implémentation des méthodes

Soit la classe

```
class A {
 int x; int y;
 A (int xx, int yy) {
  x = xx; y = yy;
 }
 void M() { x++; }
 void N() { y++; }
};
```

Sa traduction en C pourrait être

```
typedef struct A {
 int x; int y
} A;
void A_A (A* _this, int xx, int yy) {
 _this->x = xx; _this->y = yy;
}
```

```
void A_M (A* _this) {
 _this->x++;
}
void A_N (A* _this) {
 _this->y++;
}
```

L'allocation est faite avant l'appel du constructeur

Implémentation des méthodes

Et pour la classe

```
class B : public A {
 int a;
 B (int xx, int yy, int aa)
   : A (xx, yy), a (aa) { }
 void M() { x++; a++; }
};
```

Sa traduction en C pourrait être

```
typedef struct B {
 A _parenteA; int a;
} B;

void B_B (B* _this, int xx,
          int yy, int aa) {
 A_A (&_this->_parenteA, xx, yy);
 _this->a = aa;
}
```

```
void B_M (B* _this) {
 _this->_parenteA.x++;
 _this->a++;
}
```

donc **`B b; … b.N();`**
donne, puisque A::N attend un A*
`A_N (&b._parenteA);`

Implémentation des méthodes virtuelles (héritage simple)

Supposons maintenant

```
class A {
 ...
 virtual void M() { x++; }
 virtual void N() { y++; }
};
```

```
class B : public A {
 ...
 void M() { x++; a++; }
};
```

Leur traduction en C pourrait être

```
typedef struct A {
 _VTBL* _vtbl;
 int x; int yy;
} A;
```

```
typedef struct B {
 A _parenteA;
 int a;
} B;
```

on réutilise la vtbl de A

vtbl = tableau de pointeurs sur fonctions : `typedef void (*_VTBL) ();`

Implémentation des méthodes virtuelles (héritage simple)

Avec

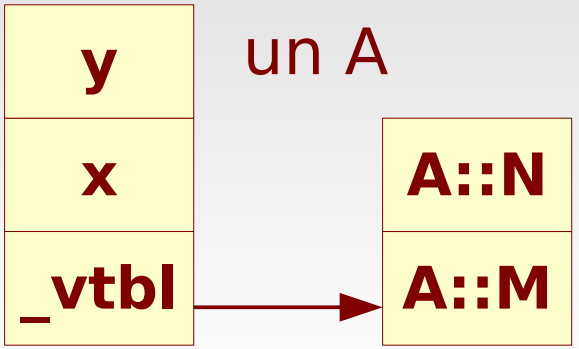
```
void A_A (A* _this, int xx, int yy) {
 _this->_vtbl = _A_VTBL;
 _this->x = xx; _this->y = yy;
}

_VTBL* _A_VTBL[] = { &_A_M, &_A_N };
```

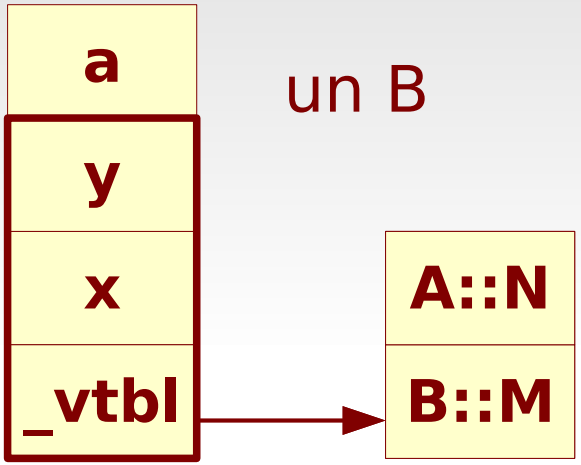
Et

```
void B_B (B* _this, int xx, int yy, int aa) {
 A_A (&_this->_parenteA, xx, yy);
 _this->_parenteA._vtbl = _B_VTBL;
 _this->a = aa;
}

_VTBL* _B_VTBL[] = { &_B_M, &_A_N };
```

ce qui explique pourquoi il n'y a pas de liaison dynamique dans la construction de la base A

Implémentation des méthodes virtuelles (héritage simple)





Donc

```
A* p; ...

a->M();
a->N();
```

se traduit par

```
(*p->_vtbl[0])(p);
(*p->_vtbl[1])(p);
```

Implémentation des classes (héritage multiple)

Supposons maintenant les classes

```
class A {
 int x;
 void M()
 void P();
};
```

```
class B {
 int a;
 void N()
};
```

```
class C : public A,
          public B {
 int k;
 void M() { ... }
};
```

Leur traduction en C pourrait être

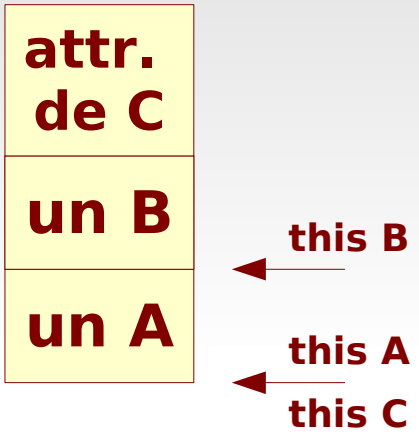
```
typedef struct A {
int x;
} A;
```

```
typedef struct C {
   A _parenteA;
   B _parenteB;
   int k;
} C;
```

Implémentation des classes (héritage multiple)

La structure d'un C est donc

L'appel d'une méthode peut changer la valeur de this



Les méthodes sont :

```
void A_M (A* this);
void A_P (A* this);
void B_N (B* this);
void C_M (C* this);
```

et

```
C c;
c.M();
c.N();
c.P();
```

se traduit par

```
C c; ...
C_M (&c);
B_N (&c._parenteB);
A_P (&c._parenteA);
```

Implémentation des méthodes virtuelles (héritage multiple)

Dans le cas d'appel de méthodes virtuelles, on ne sait pas quelle méthode est effectivement appelée, donc quel décalage de this appliquer

Supposons les méthodes M, N, P, Q virtuelles

```
class A {
 int x;
 void M();
 void P();
};
```

```
class B {
 int a;
 void N();
 void Q() {
N(); x++;
 }
};
```

```
class C : public A,
          public B {
 int k;
 void M() { Q(); }
 void N() { k++; }
};
```

Implémentation des méthodes virtuelles (héritage multiple)

Si M, N, P, Q sont virtuelles

```
class A {
 int x;
 void M();
 void P();
};
```

```
class B {
 int a;
 void N();
 void Q() {
N(); x++;
 }
};
```

```
class C : public A,
          public B {
 int k;
 void M() { Q(); }
 void N() { P(); k++; }
};
```

alors

```
C c;
A* a = &c;
a->M();
```

appelle successivement
C::M, B::Q, C::N, A::P

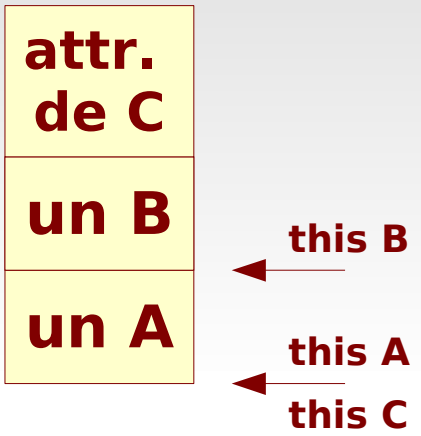
Implémentation des méthodes virtuelles (héritage multiple)

Les tables virtuelles contiendront respectivement des pointeurs

- pour A, sur : A::M et A::P
- pour B, sur : B::N et B::Q
- pour C :
  - comme ses parents A et B ont un pointeur sur une table, il n'a pas besoin d'un pointeur propre
  - la table de C (_parenteA._vtbl) : C::M, A::P, C::N, B::Q
    (elle sert aussi de table pour les C vus comme des A)
  - la table des C vus comme des B (_parenteB._vtbl) : C::N et B::Q

Implémentation des méthodes virtuelles (héritage multiple)

Puisqu'il faut ajuster this selon
La méthode appelée, les tables virtuelles contiendront des couples
(adresse-méthode, ajustement-this)



La règle est simple :

- ajustement = 0 dans la vtbl des C (et des C vus comme A) si méthode de A ou C, = Tb (taille de B) si méthode de B
- ajustement = 0 dans la vtbl des C vus comme B si méthode de B, = -Tb si méthode de A ou C

Donc :

- vtbl des C (et des C vus comme A) :
  (C::M,0), (A::P,0), (C::N,0), (B::Q,Tb)
- vtbl des C vus comme B : (C::N, -Tb), (B::Q, 0)

Implémentation des méthodes virtuelles (appel de méthode)

type générique de pointeur sur méthode

```
typedef void (*MPTR) (void*);
```

élément de la vtbl

```
typedef struct VTLB {
  MPTR meth;
  int  offset;
} VTBL;
```

vtbl de C

vtab quand vu comme un C ou un A

```
#define TB sizeof(A)
   VTAB C_vtab[] = { {(MPTR)C_M, 0}, {(MPTR)A_P, 0},
                     {(MPTR)C_N, 0}, {(MPTR)B_Q, TB} };
```

vtab quand vu comme un B

```
   VTAB C_B_vtab [] = { {(MPTR)C_N, -TB}, {(MPTR)B_Q, 0} };
```

macro pour appeler une méthode

```
#define CALL(p,k) (*p->_vtbl[k].meth)((char*)p + p->_vtbl[k].offset)
```

appel a->M() si M a le n° 0 => `CALL (a, 0);`