

# Héritage du C++

Philippe Prados

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



*Préservez l'environnement,  
n'imprimez pas ce document*

## TABLE DES MATIERES

1.	Héritage simple .....	3
2.	Héritage multiple .....	3
3.	Méthodes virtuelles.....	5
4.	Héritage multiple et méthodes virtuelles.....	7
4.1	Une classe avec méthodes virtuelles.....	7
4.2	Plusieurs classes avec méthodes virtuelles .....	9
5.	Héritage virtuel .....	11
6.	Héritages virtuels et méthodes virtuelles .....	14
7.	Résumé.....	16

## Avant propos

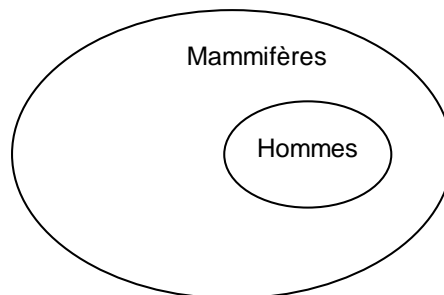
Ce document explique comment le compilateur traduit les différents héritages..

Pour bien maîtriser le C++, il me semble nécessaire de connaître les mécanismes utilisés par le compilateur pour générer le programme. Cet article décrit différentes traductions possibles de l'héritage et du polymorphisme qui sont les caractéristiques les plus connues du langage.

Écrire un programme en suivant une approche objet consiste à spécifier des composants simples participant à une architecture plus complexe. L'ensemble de ces composants est en relation. L'héritage est une relation particulière (fort utile) proposée par les langages objets. Voici une description des différentes techniques utilisées pour générer les héritages. Attention, les traductions proposées ne sont pas imposées. Chaque compilateur peut choisir une autre approche plus efficace.

## 1. HERITAGE SIMPLE

Dans le C++, l'héritage simple est une syntaxe permettant d'ajouter des comportements et des attributs à un composant simple. Dans une représentation ensembliste, cela correspond à un sous-ensemble ayant un comportement particulier. Par exemple, un Homme hérite des capacités de Mammifère.

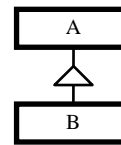


Toutes les propriétés de Mammifère sont héritées par Homme. Un Homme est une catégorie de Mammifère. Les attributs et les méthodes de Mammifère existent pour l'Homme.

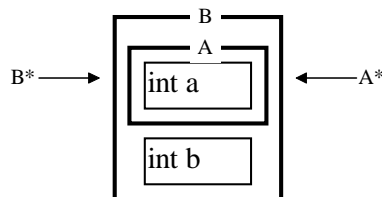
Prenons un exemple d'héritage simple, et regardons comment le compilateur stocke l'objet en mémoire.

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

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



L'objet est stocké en mémoire ainsi :



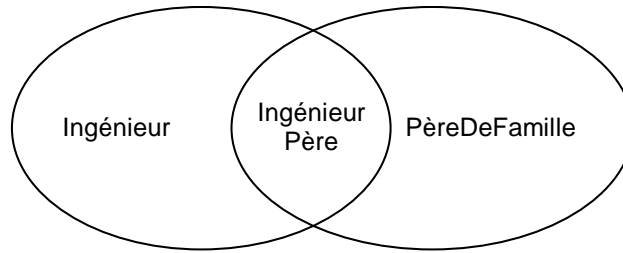
La structure B est accolée à la structure A. Un pointeur de type B est directement compatible avec un pointeur de type A. En langage C, cela se traduit comme cela :

```
struct A
{ int a;
};

struct B
{ struct A A; // Objet A hérité
  int b;
};
```

## 2. HERITAGE MULTIPLE

L'héritage multiple est une intersection entre plusieurs ensembles. Une classe hérite des capacités de plusieurs ensembles. Un homme peut être à la fois Ingénieur et PèreDeFamille.

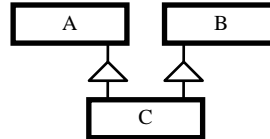


Un IngénieurPère hérite des capacités de Ingénieur et des capacités de PèreDeFamille.

L'héritage multiple fonctionne comme précédemment, soit :

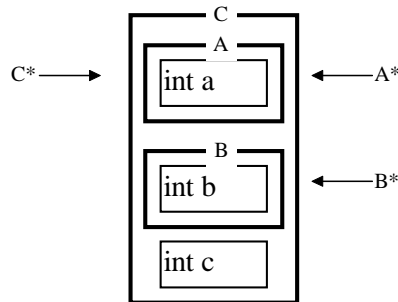
```
class A
{ public:
  int a;
};

class B
{ public:
  int b;
};
```



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

L'objet est stocké en mémoire ainsi :



On remarque que l'objet C est construit en accolant les deux objets A et B. Le pointeur de l'objet C est compatible avec le pointeur de la partie A de C, mais pas avec le pointeur de la partie B de C. Cela montre qu'une conversion d'un pointeur C vers un pointeur B modifie la valeur du pointeur.

```
C* pc=&c;
B* pb=&c; // Ajuste le pointeur, conversion en pointeur B*
assert((void*)pb!=(void*)pc);
```

Cela explique pourquoi il faut faire attention lors des conversions. Les conversions par défaut ne posent pas de problème, car elles adaptent correctement la valeur du pointeur. Par contre, les conversions avec des pointeurs de type incompatible ne modifient pas la valeur du pointeur. Cela entraîne des erreurs très difficiles à détecter. En langage C, cela se traduit comme cela :

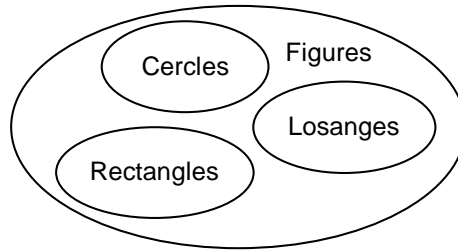
```
struct A
{ int a;
};

struct B
{ int b;
};

struct C
{ struct A A; // Objet A hérité
  struct B B; // Objet B hérité
  int c;
};
```

### 3. METHODES VIRTUELLES

Le polymorphisme est la capacité de modifier certains comportements hérités. Une *Figure* géométrique peut être héritée par des *Cercles*, des *Rectangles*, des *Losanges*.



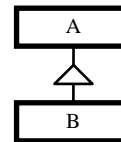
Une *Figure* offre la capacité de s'afficher. L'affichage d'un *Cercle* est différent de l'affichage d'un *Rectangle*. Chacun va vouloir modifier la méthode d'affichage de la *Figure* pour tenir compte des ses capacités propres d'affichage. Le polymorphisme permet à chaque classe de redéfinir un traitement hérité d'une classe de base.

Le polymorphisme est implémenté à l'aide des méthodes virtuelles. Le compilateur construit une table de pointeur de fonctions pour chaque classe. Chaque instance de la classe possède un pointeur caché dans l'objet, pointant sur cette table. Ce pointeur est initialisé par le constructeur de l'objet.

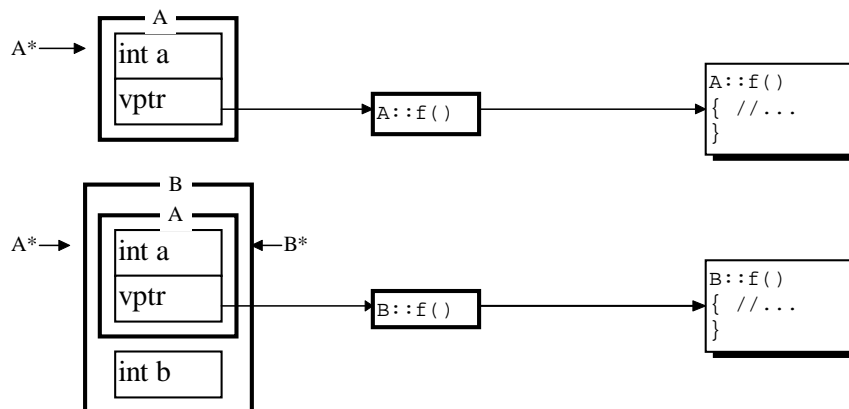
```

class A
{ public :
  int a;
  virtual void f();
};

class B : public A
{ public:
  int b;
  virtual void f();
};
  
```



Les deux objets sont stockés en mémoire comme cela :



Le pointeur `vptr` pointe sur la table de saut. Lorsque la méthode `f()` est appelée pour un pointeur de type `A`, une indirection est effectuée pour appeler la méthode indiquée dans la table de saut. L'appel ressemble à cela :

```

A a;
B b;
A* pa=&a;
pa->vptr[0](); // Appel de A::f()
pa=&b;
pa->vptr[0](); // Appel de B::f()
  
```

On constate que le même appel peut aboutir à la version `A::f()` ou `B::f()`. Le polymorphisme est réglé par ce mécanisme. Chaque nouvelle méthode virtuelle est ajoutée à la suite de la table de saut. Un index lui est attribué. Le compilateur modifie chaque appel de méthode virtuelle par l'appel précédent, avec un index par méthode.

Le polymorphisme ne peut apparaître qu'avec des pointeurs ou des références. L'appel d'une méthode par l'opérateur "point", si l'objet n'est pas une référence, n'est pas ambigu. Dans ce cas, le compilateur peut appeler directement la méthode et utiliser éventuellement la version inline de celle-ci.

```

A a;
A* pa=&a;
a.f(); // Appel inline
pa->f(); // Appel non inline
  
```

Cela permet de générer en `inline` la méthode si nécessaire. Il n'est pas absurde de déclarer une méthode virtuelle en `inline`. Le compilateur choisit la technique d'accès la plus appropriée à l'utilisation.

Le pointeur `vp_ptr` est initialisé par le constructeur. Mais attention, ce pointeur est initialisé lors de l'accolade ouvrante de ce dernier. Les constructeurs de A et de B sont traduits comme cela :

```
A::A()
{
  vp_ptr=A::vtable;
}

B::B()
{
  A::A();
  vp_ptr=B::vtable;
}
```

On constate que le pointeur `vp_ptr` pointe sur la table `B::vtable` après l'appel du constructeur de A. Cela entraîne qu'un appel à une méthode virtuelle lors de l'exécution du constructeur de A appellera la méthode virtuelle de A mais pas la version virtuelle de B. Il ne faut pas appeler de méthode virtuelle dans les constructeurs.

Les destructeurs initialisent le pointeur `vp_ptr` avant de s'exécuter. Il n'est donc pas possible d'appeler une méthode virtuelle dans un destructeur. Les destructeurs de A et de B sont traduits comme cela :

```
A::~~A()
{
  vp_ptr=A::vtable;
  // Code du destructeur
}

B::~~B()
{
  vp_ptr=B::vtable;
  // Code du destructeur
  A::~~A();
}
```

En langage C, cela se traduit ainsi :

```
typedef void (*fnvtable)();

fnvtable A::vtable[] = // Table de saut de A
{(fnvtable)A::f};

struct A
{
  int a;
  fnvtable* vp_ptr; // Pt de tbl de saut
};
// ctr
A::A(A* const this)
{
  this->vp_ptr=A::vtable; // Init vp_ptr
}

// dtr
A::~~A(A* const this)
{
  this->vp_ptr=A::vtable; // Reset vp_ptr
}

////////////////////////////////////
fnvtable B::vtable[] = // Table de saut de B
{(fnvtable)B::f};

struct B
{
  struct A A; // Objet A hérité
  int b;
};

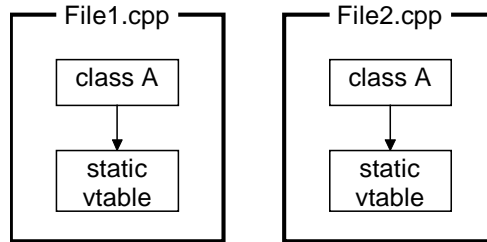
// ctr
B::B(B* const this)
{
  A::A(&this->A);
  this->A.vp_ptr=B::vtable; // Modifie vp_ptr de A
}

// dtr
B::~~B(B* const this)
{
  this->A.vp_ptr=B::vtable; // Reset vp_ptr de A
  A::~~A(&this->A);
}
```

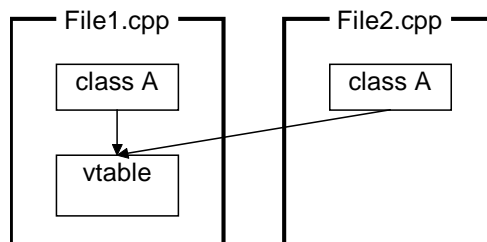
Cette traduction en langage C n'est qu'un exemple de traduction possible. Les compilateurs sont libres de traduire le code C++ comme ils le veulent. En général, le pointeur `vp_ptr` est ajouté à la fin de la structure C pour permettre une traduction aisée vers une structure C classique.

L'ajout d'une méthode virtuelle à une structure ne modifie pas le début de la structure équivalente C. Une fonction C peut recevoir une structure C++ ayant une méthode virtuelle. Cela n'est pas obligatoire et dépend de chaque compilateur.

D'autre part, certains compilateurs créent la table de sauts virtuels en statique dans chaque module. Il y a autant de tables de sauts, qu'il y a de modules utilisant une méthode virtuelle de la classe.



D'autres génèrent un enregistrement particulier pour le `link` afin de partager les tables de sauts lors du `link`. Dans ce cas, il n'existe qu'une table de sauts par classe dans l'ensemble de l'application. Une autre approche consiste à générer la table de saut dans le fichier déclarant la première méthode virtuelle de la classe. Celle-ci ne devant être présente qu'une seule fois dans l'application, la table de saut l'est également.



Cela entraîne qu'il n'est pas possible de comparer les pointeurs de table de sauts virtuels entre deux instances d'une classe. Les pointeurs `vptr` ne sont pas forcément les mêmes, mais les fonctionnalités, elles, sont identiques. Il n'est pas possible de comparer deux instances ayant un `vptr` à l'aide de `memcmp()`.

```
class A
{ int a;
  public:
  virtual void print();
  int operator ==(const A& x)
  { return !memcmp(this,&x,sizeof(x)); // Erreur
  }
};
```

## 4. HERITAGE MULTIPLE ET METHODES VIRTUELLES

Lors de l'utilisation de méthode virtuelle et d'héritage multiple, le compilateur doit effectuer des adaptations dans le code généré.

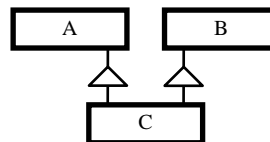
### 4.1 Une classe avec méthodes virtuelles

Si une classe hérite de plusieurs autres, dont une possède des méthodes virtuelles, il y a un problème d'adaptation de pointeur.

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

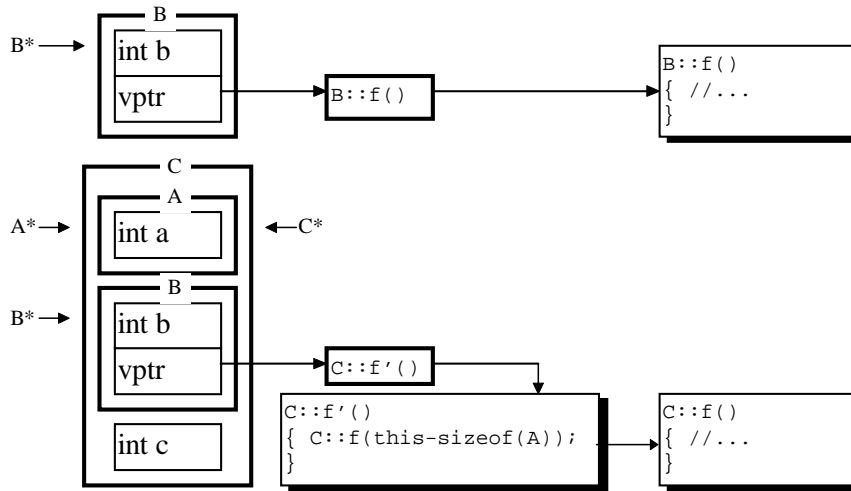
class B
{ public:
  int b;
  virtual void f();
};

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



La méthode `C::f()` doit recevoir un pointeur `this` sur un objet de type C, alors que la méthode `B::f()` attend un pointeur de type B. Comme nous l'avons vu plus haut, en cas d'héritage multiple, un pointeur de type C n'est pas directement compatible avec un pointeur de

type B. Il faut ajuster le pointeur. Le pointeur étant ajusté sur la partie B de C, il faut l'ajuster de nouveau pour retourner à l'adresse de l'objet C. Le compilateur règle cela, en créant une méthode supplémentaire permettant d'ajuster la valeur de `this`. La mémoire est gérée comme décrit ci-après :



En langage C, cela se traduit ainsi :

```
typedef void (*fnvtable)();

struct A
{ int a;
};

// ctr
A::A(A* const this)
{
}

// dtr
A::~~A(A* const this)
{
}

////////////////////////////////////
fnvtable B::vtable[] = // Table de saut de B
{(fnvtable)B::f};

struct B
{ int b;
  fnvtable* vptr; // Pt de tbl de saut
};

B::B(B* const this)
{ this->vptr=B::vtable; // Init vptr
}

B::~~B(B* const this)
{ this->vptr=B::vtable; // Reset vptr
}

////////////////////////////////////
struct C
{ struct A A; // Objet A hérité
  struct B B; // Objet B hérité
  int c;
};

fnvtable C::vtable[] = // Table de saut de C
{(fnvtable)C::f' };

void C::f'(B* const this)
{ C::f((C*)((char*)this)-offsetof(C,B)); // Convertie en (C*)this
}

// ctr
C::C(C* const this)
{ A::A(&this->A);
```

```

B::B(&this->B);
this->B.vptr=C::vtable;    // Modifie vptr de B
}

// dtr
C::~~C(C* const this)
{ this->B.vptr=C::vtable;    // Modifie vptr de B
  B::B(&this->B);
  A::A(&this->A);
}

```

#### 4.2 Plusieurs classes avec méthodes virtuelles

Lors d'un héritage multiple de classes ayant des méthodes virtuelles, il existe plusieurs pointeurs vptr.

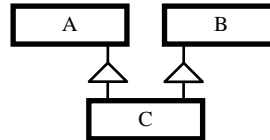
```

class A
{ public:
  int a;
  virtual void fa();
  virtual void f();
};

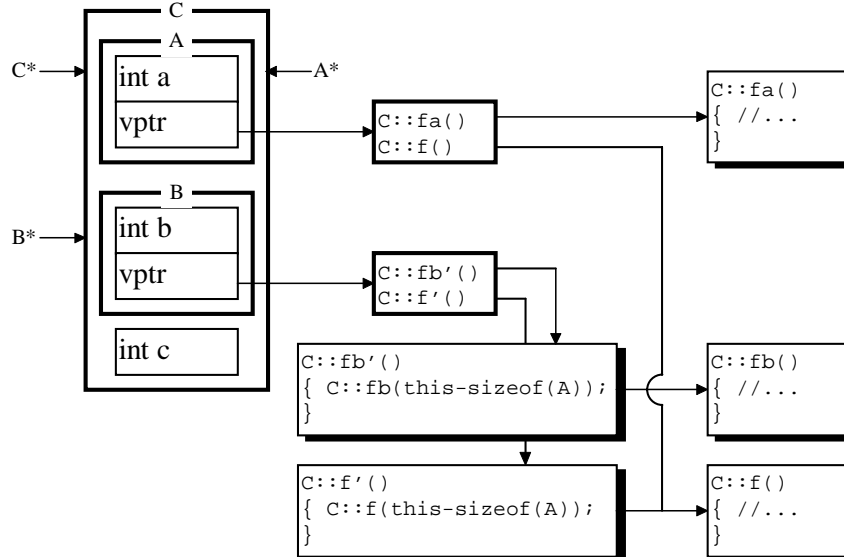
class B
{ public:
  int b;
  virtual void fb();
  virtual void f();
};

class C : public A, public B
{ public:
  int c;
  virtual void fa();
  virtual void fb();
  virtual void f();
};

```



La mémoire est gérée comme cela :



Les méthodes supplémentaires ne sont ajoutées que pour les méthodes virtuelles des classes héritées en deuxième position et suivantes.

Si une méthode possède la même signature dans deux classes différentes, héritées par le même objet, la méthode virtuelle est indiquée dans les vtable des deux classes de base. C'est le cas de la méthode f().

En langage C, cela se traduit comme cela :

```

typedef void (*fnvtable)();

fnvtable A::vtable[]= // Table de saut de A
{ (fnvtable)A::fa,
  (fnvtable)A::f
};

struct A

```

```

{ int a;
  fnvtable* vptr;    // Pt de tbl de saut
};

// ctr
A::A(A* const this)
{ this->vptr=A::vtable;    // Init vptr
}
// dtr
A::~A(A* const this)
{ this->vptr=A::vtable;    // Reset vptr
}

////////////////////////////////////
fnvtable B::vtable[]= // Table de saut de B
{(fnvtable)B::fb,
 (fnvtable)B::f
};

struct B
{ int b;
  fnvtable* vptr;    // Pt de tbl de saut
};

// ctr
B::B(B* const this)
{ this->vptr=B::vtable;    // Init vptr
}

// dtr
B::~B(B* const this)
{ this->vptr=B::vtable;    // Reset vptr
}

////////////////////////////////////
struct C
{ struct A A; // Objet A hérité
  struct B B; // Objet B hérité
  int c;
};

void C::fb'(B* const this)
{ C::fb((C*)((char*)this)-offsetof(C,B)); // Convertie en (C*)this
}

void C::f'(B* const this)
{ C::f((C*)((char*)this)-offsetof(C,B)); // Convertie en (C*)this
}

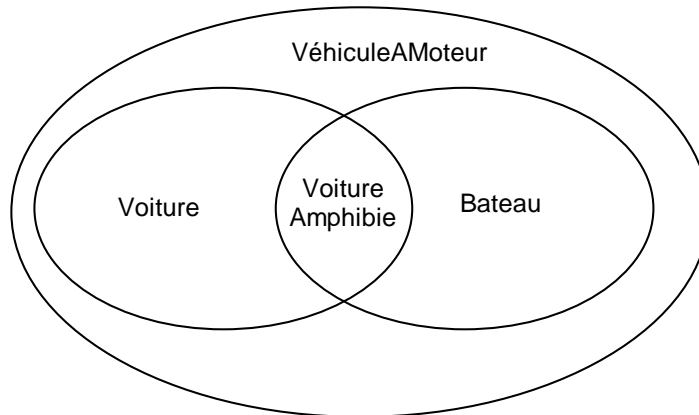
fnvtable C::vtable1[]=// Table de saut de C pour A
{(fnvtable)C::fa,
 (fnvtable)C::f
};
fnvtable C::vtable2[]=// Table de saut de C pour B
{(fnvtable)C::fb',
 (fnvtable)C::f'
};

// ctr
C::C(C* const this)
{ A::A(&this->A);
  this->A.vptr=C::vtable1;    // Modifie vptr de A
  B::B(&this->B);
  this->B.vptr=C::vtable2;    // Modifie vptr de B
}
// dtr
C::~C(C* const this)
{ this->B.vptr=C::vtable2;    // Modifie vptr de B
  this->A.vptr=C::vtable1;    // Modifie vptr de A
  B::~B(&this->B);
  A::~A(&this->A);
}

```

## 5. HERITAGE VIRTUEL

L'héritage virtuel est l'enrichissement le plus complexe. Un Bateau hérite de VéhiculeAMoteur. Une Voiture également. Une VoitureAmphibie hérite de Voiture et de Bateau. Si ce véhicule utilise le même moteur pour la propulsion lors de l'utilisation en Voiture et en Bateau, le moteur doit être partagé par les deux classes héritées. Dans une représentation ensembliste, cela donne :



La voiture amphibie n'hérite qu'une seule fois des capacités de VéhiculeAMoteur.

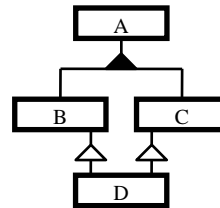
Voici un exemple simple.

```
class A
{ public:
  int a;
  virtual void g() { cout << "A::g()" << endl; }
  virtual void f() { cout << "A::f()" << endl; }
};

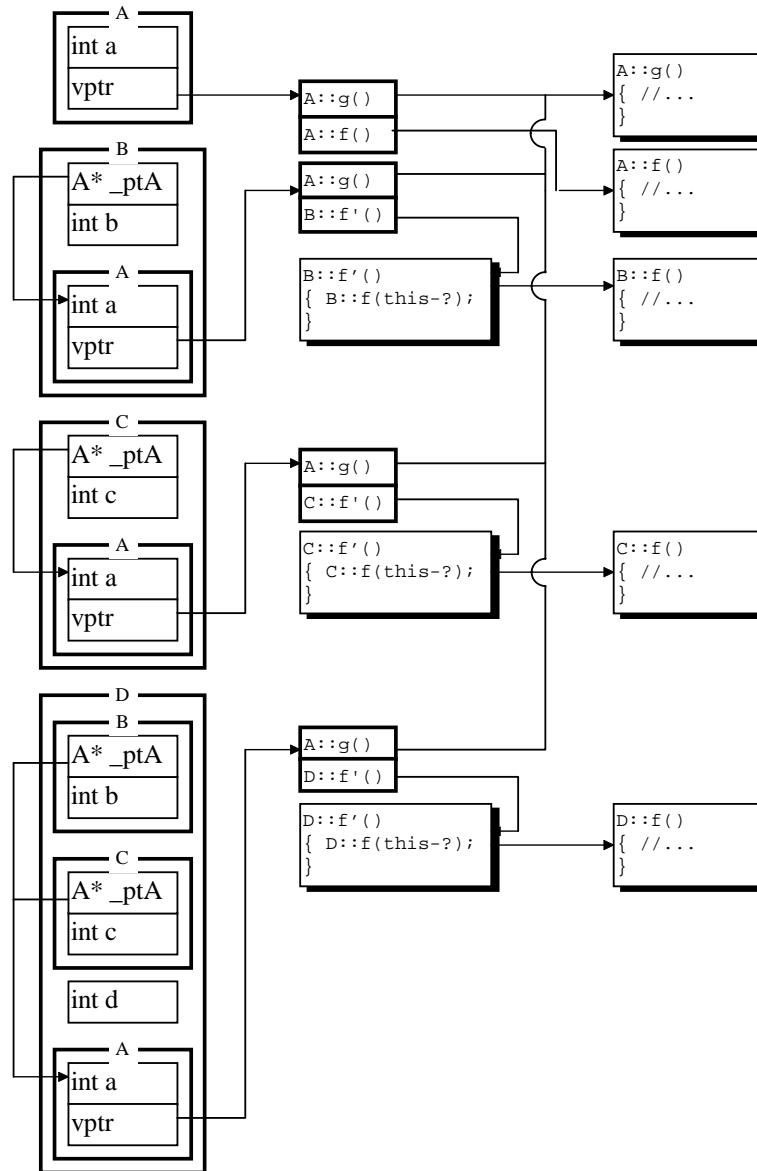
class B : virtual public A
{ public:
  int b;
  virtual void f() { cout << "B::f()" << endl; }
};

class C : virtual public A
{ public:
  int c;
  virtual void f() { cout << "C::f()" << endl; }
};

class D : public B, public C
{ public:
  int d;
  virtual void f() { cout << "D::f()" << endl; }
};
```



Le compilateur implante ces classes comme décrit ci-dessous :



La classe A possède un pointeur sur une table virtuelle. Cette table contient des pointeurs sur des méthodes. Pour un héritage simple, une autre table est simplement construite en ajustant les pointeurs des méthodes. Le problème se complique pour un héritage virtuel. En effet, dans l'exemple précédent, un pointeur de type B n'est pas directement compatible avec un pointeur de type A. Dans ce cas, il faut que le programme, lise le pointeur `_ptA` pour connaître l'adresse de l'objet A. Inversement, l'appel de la méthode `f()` à partir d'un pointeur de type A, doit ajuster le pointeur `this` pour qu'il pointe sur un objet du type de la nouvelle version de la méthode virtuelle. Dans l'exemple suivant,

```
void main()
{ D d;
  A* ptA=&d;
  ptA->f();

  D* ptD=&d;
  ptD->f();
}
```

l'appel de `f()` est effectué avec un pointeur `this` pointant sur la partie A de D, et non sur un objet de type D. Il faut ajuster ce pointeur pour le faire pointer sur un objet de type D. C'est le rôle des fonctions "prime".

Celles-ci soustraient à `this` l'offset nécessaire puis appellent la version des méthodes correspondantes. Ces méthodes "prime" ne sont exécutées que lors d'un appel d'une méthode virtuelle d'une classe héritée.

En langage C, cela se traduit comme cela :

```
typedef void (*fnvtable());
```

```

fnvtable A::vtable[]= // Table de saut de A
{(fnvtable)A::g,
 (fnvtable)A::f
};

struct A
{ int a;
  fnvtable* vptr; // Pt de tbl de saut
};

// ctr
A::A(A* const this)
{ this->vptr=A::vtable; // Init vtable
}

// dtr
A::~A(A* const this)
{ this->vptr=A::vtable; // Reset vtable
}

/////////////////////////////////////////////////////////////////
struct _B // Objet B sans A
{ A* _ptA; // Pointeur sur A
  int b;
};

struct B
{ struct _B _B; // Objet B sans A
  struct A A; // Objet A
};

void B::f'(A* const this)
{ B::f((B*)((char*)this)-offsetof(B,A)); // Convertie en (B*)this
}

fnvtable B::vtable[]= // Table de saut de B
{(fnvtable)A::g,
 (fnvtable)B::f'
};

// ctr
_B::_B(_B* const this)
{ this->_ptA->vptr=B::vtable; // Modifie vptr
}
B::B(B* const this)
{ A::A(&this->A); // ctr de A
  this->_B._ptA=&this->A; // Init _ptA de _B
  _B::_B(&this->_B); // ctr de Obj B sans A
}

// dtr
_B::~_B(_B* const this)
{ this->_ptA->vptr=B::vtable; // Modifie vptr
}
B::~B(B* const this)
{ this->_B._ptA=&this->A; // Reset _ptA de _B
  _B::~_B(&this->_B); // dtr de Obj B sans A
  A::~A(&this->A); // dtr de A
}

/////////////////////////////////////////////////////////////////
struct _C // Objet C sans A
{ A* _ptA; // Pointeur sur A
  int c;
};

struct C
{ struct _C _C; // Objet C sans A
  struct A A; // Objet A
};

void C::f'(A* const this)
{ C::f((C*)((char*)this)-offsetof(C,A)); // Convertie en (C*)this
}

fnvtable C::vtable[]= // Table de saut de C
{(fnvtable)A::g,
 (fnvtable)C::f'
};

```

```

// ctr
_C::_C(_C* const this)
{ this->_ptA->vptr=C::vtable; // Modifie vptr
}
C::C(C* const this)
{ A::A(&this->A); // ctr de A
  this->_C._ptA=&this->A; // Init _ptA de _C
  _C::_C(&this->_C); // ctr de Obj C sans A
}

// dtr
_C::~~C(_C* const this)
{ this->_ptA->vptr=C::vtable; // Modifie vptr
}
C::~~C(C* const this)
{ this->_C._ptA=&this->A; // Reset _ptA de _C
  _C::~~C(&this->_C); // dtr de Obj C sans A
  A::~~A(&this->A); // dtr de A
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
struct D
{ struct _B _B; // Obj B sans A
  struct _C _C; // Obj C sans A
  struct A A; // Obj A
  int d;
};

void D::f(A* const this)
{ D::f((D*)((char*)this)-offsetof(D,A)); // Convertie (D*)this
}

fnvtable D::vtable[]= // Table de saut de D
{(fnvtable)A::g,
 (fnvtable)D::f'
};

// ctr
D::D(D* const this)
{ A::A(&this->A); // ctr de A
  this->_B._ptA=&this->A; // Init _ptA de _B
  _B::_B(&this->_B); // ctr Obj B sans A
  this->_C._ptA=&this->A; // Init _ptA de _C
  _C::_C(&this->_C); // ctr Obj C sans A
  this->A.vptr=D::vtable; // Modifie vptr
}

// dtr
D::~~D(D* const this)
{ this->A.vptr=D::vtable; // Reset vptr
  _C::~~C(&this->_C); // dtr Obj C sans A
  _B::~~B(&this->_B); // dtr Obj B sans A
  A::~~A(&this->A); // dtr de A
}

```

Ceci est la théorie. En réalité, pour éviter au maximum les méthodes “prime”, les méthodes virtuelles sont générées avec un pointeur `this` pointant réellement sur un objet du type où la déclaration initiale de la méthode est effectuée. Dans le cas présent, `this` pointe sur la partie A de D, alors qu’il est vu par le programmeur comme étant de type D. Toute la compilation soustrait les offsets nécessaires pour accéder aux objets de type D. Cela veut dire qu’une même méthode `f()` rédigée normalement ou virtuellement est générée complètement différemment.

Malheureusement, cette technique n’est pas toujours possible. En effet, s’il existe une classe E héritant de la classe D, les éléments de E sont ajoutés à la fin de la structure D. Un pointeur sur E retrouve le même schéma que pour les méthodes “prime”. Dans ce cas, le compilateur ajoute les méthodes “prime”.

## 6. HERITAGES VIRTUELS ET METHODES VIRTUELLES

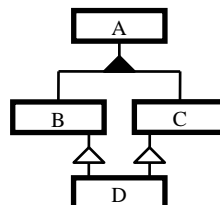
Une classe dérivant virtuellement d’une classe de base et ajoutant de nouvelles méthodes virtuelles, ajoute un pointeur `vptr`.

```

class A
{ public:
  int a;
  virtual void f() { cout << "A::f()" << endl; }
};

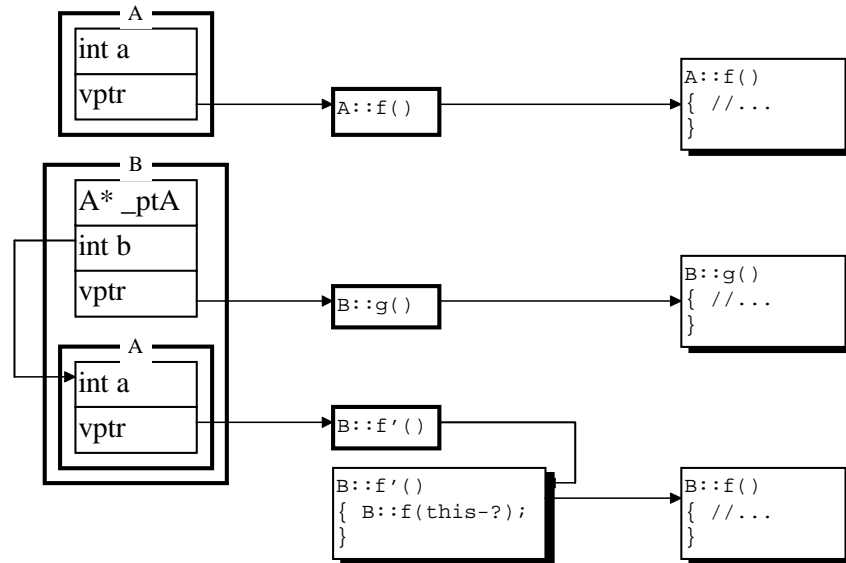
class B : virtual public A
{ public:

```

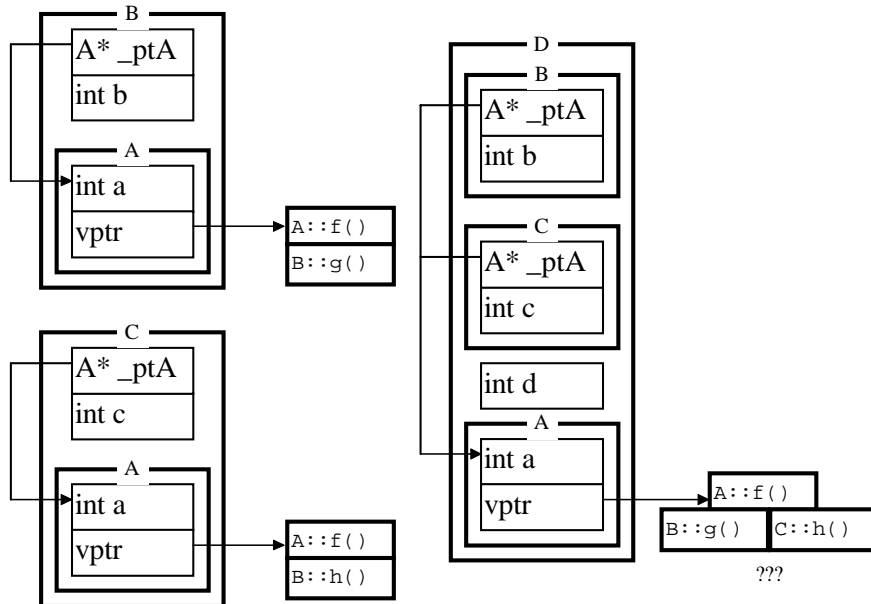


```
int b;
virtual void f() { cout << "B::f()" << endl; }
virtual void g() { cout << "B::g()" << endl; }
};
```

Le compilateur implante ces classes comme cela :



L'appel de `g()` est effectué *via* le pointeur `vptr` de B. Le `vptr` de A ne peut pas être utilisé, car si une autre classe hérite virtuellement de A et déclare une autre méthode virtuelle `h()`, elle serait également en deuxième position dans la `vtable` de A. Deux méthodes différentes se retrouveraient à la même position.



Le compilateur ne pourrait pas construire la `vtable` de D. C'est pour cela qu'il faut obligatoirement ajouter un pointeur `vptr` pour toute nouvelle méthode virtuelle d'une classe dérivée virtuellement.

En langage C, cela se traduit comme cela :

```
typedef void (*fnvtable)();

fnvtable A::vtable[] = // Table de saut de A
{ (fnvtable)A::f
};

struct A
{ int a;
  fnvtable* vptr; // Pt de tbl de saut
};
```

```

// ctr
A::A(A* const this)
{ this->vptr=A::vtable;      // Init vptr
}

// dtr
A::~A(A* const this)
{ this->vptr=A::vtable;      // Reset vptr
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
struct _B      // Objet B sans A
{ A* _ptA;     // Pointeur sur A
  int b;
  fnvtable* vptr; // Pt de tbl de saut
};

struct B
{ struct _B _B;      // Objet B sans A
  struct A A; // Objet A
};

void B::f'(A* const this)
{ B::f((B*)((char*)this)-offsetof(B,A)); // Convertie en (B*)this
}

fnvtable B::vtable1[]// Table de saut de B pour A
{ (fnvtable)B::f'
};

fnvtable B::vtable2[]// Table de saut de B
{ (fnvtable)B::g
};

// ctr
_B::_B(_B* const this)
{ this->_ptA->vptr=B::vtable1; // Modifie vptr de A
  this->vptr=B::vtable2;
}
B::B(B* const this)
{ A::A(&this->A); // ctr de A
  this->_B._ptA=&this->A; // Init _ptA de _B
  _B::_B(&this->_B); // ctr de Obj B sans A
}

// dtr
_B::~_B(_B* const this)
{ this->vptr=B::vtable2;
  this->_ptA->vptr=B::vtable1; // Modifie vptr de A
}
B::~B(B* const this)
{ _B::~_B(&this->_B); // dtr de Obj B sans A
  A::~A(&this->A); // dtr de A
}

```

## 7. RESUME

Pour conclure, voici un résumé des informations à connaître pour comprendre les principes de base de la traduction interne des héritages et du polymorphisme.

- L'héritage est traduit par l'accumulation des attributs des classes héritées.
- Le polymorphisme est géré à l'aide d'un pointeur caché, référençant une table de saut indiquant chaque méthode virtuelle. Ce pointeur sur la table de saut est initialisé dans le constructeur de la classe.
- Ce pointeur n'est jamais copié lors d'une affectation ou d'un constructeur de copie.
- La conversion d'un pointeur d'une classe dérivée vers un pointeur d'une classe de base peut modifier la valeur du pointeur.
- L'héritage virtuel utilise une indirection supplémentaire pour accéder à la classe de base.
- Il n'est pas possible de convertir un pointeur d'une classe de base héritée virtuellement en un pointeur d'une classe dérivée.