Programmation C++

2ème année Informatique

J. Allali
julien.allali@labri.fr

ENSEIRB

Prog. C++

Plan

Plan

Plan

Héritage: définition

L'héritage est un mécanisme permettant de construire un type `T` à par d'un autre type `Base`. Le type `T` se retrouve doté de comportements (méthodes) et propriétés (attributs) du type de Base. Une relation de typage relie ces deux types: `T` est un sous-type de `Base` (ou classe dérivée de `Base`) tandis que `Base` est un super-type de `T`.

Héritage: définition

L'héritage est un mécanisme permettant de construire un type `T` à par d'un autre type `Base`. Le type `T` se retrouve doté de comportements (méthodes) et propriétés (attributs) du type de Base. Une relation de typage relie ces deux types: `T` est un sous-type de `Base` (ou classe dérivée de `Base`) tandis que `Base` est un super-type de `T`.

En C++, les membres hérités et la relation de typage qu'il existe entre `T` et `Base` ne sont pas necessairement visible de l'exterieur.

Héritage: définition

L'héritage est un mécanisme permettant de construire un type `T` à par d'un autre type `Base`. Le type `T` se retrouve doté de comportements (méthodes) et propriétés (attributs) du type de Base. Une relation de typage relie ces deux types: `T` est un sous-type de `Base` (ou classe dérivée de `Base`) tandis que `Base` est un super-type de `T`.

En C++, les membres hérités et la relation de typage qu'il existe entre `T` et `Base` ne sont pas necessairement visible de l'exterieur.

La relation est indiquée lors de la déclaration de `T`:

```
class T: public Base{ ...
```

Nous reviendrons sur le sens du **public** plus tard.

Héritage: exemple

Voici un exemple classique d'héritage:

```
class Tuyau{
  double _diametre;
public:
  double debit() const;
  double diametre() const;
};
```

```
class TuyauPer : public Tuyau{
  double _densite;
public:
  double densite() const;
};
```

Héritage: exemple

Voici un exemple classique d'héritage:

```
class Tuyau{
  double _diametre;
public:
  double debit() const;
  double diametre() const;
};
```

```
class TuyauPer : public Tuyau{
  double _densite;
public:
  double densite() const;
};
```

```
TuyauPer tper;
Tuyau &tuyau=tper; // relation de typage
tper.densite();    // ok
tuyau.densite();   // erreur!
tper.debit();      //ok
```

Héritage: pointeurs et référence

L'utilisation de la relation de typage ne peut se faire qu'avec les pointeurs ou les références.

Héritage: pointeurs et référence

L'utilisation de la relation de typage ne peut se faire qu'avec les pointeurs ou les références.

En effet, pour faire jouer la relation de sous-typage on souhaite manipuler une même instance à travers différentes variables de types différents.

Héritage: pointeurs et référence

L'utilisation de la relation de typage ne peut se faire qu'avec les pointeurs ou les références.

En effet, pour faire jouer la relation de sous-typage on souhaite manipuler une même instance à travers différentes variables de types différents.

```
TuyauPer tper;
Tuyau t=tper; // Erreur !
Tuyau *p=&tper; // ok
Tuyau &r=tper; // ok
TuyauPer &r2=r; // Erreur !
```

Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un permier paramètre caché (**this** dans la méthode).

Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un permier paramètre caché (**this** dans la méthode).
Lors d'un appel, la méthode choisie est celle du type de la variable. Si la méthode n'est pas présente directement dans le type, c'est la méthode compatible de son parent le plus proche qui est choisie.

Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un permier paramètre caché (**this** dans la méthode).
Lors d'un appel, la méthode choisie est celle du type de la variable. Si la méthode n'est pas présente directement dans le type, c'est la méthode compatible de son parent le plus proche qui est choisie.

```
class A{
public:
  void f(char ){ puts("A\n");}
};
class B:public A{
public:
  void f(int ){ puts("B\n");}
};
class C:public B{};
```

```
C c;
c.f(1) ; // =>B
c.f('a'); // =>B
c.A::f('A'); // => A
c.B::f('a'); // => B
B &b=c;
b.f('a'); // => B
```

On utilise l'opérateur de porter pour désigner la méthode à appeler.

Héritage: appel

```
class A{
public:
  void m(){}
};
```

```
class B : public A{
public:
  void m(){}
};
```

```
B b;
A a;
A &r=b;
b.m(); // b est de type B => B::m()
a.m(); // a est de type A => A::m()
r.m(); // r est de type A => A::m()
A *p=&b;
p->m(); // p est de type A* => A::m()
```

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c
$nm mang.o
         U __gxx_personality_v0
00000000 T _ZN1A1mEv
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c
$nm mang.o
         U __gxx_personality_v0
00000000 T _ZN1A1mEv
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`
Il faut savoir que cette étape d’encodage n’est pas normalisée et est propre à chaque compilateur. Ainsi, les objets compilés avec un compilateur ne seront sans doute pas exploitable avec un autre compilateur!

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c
$nm mang.o
         U __gxx_personality_v0
00000000 T _ZN1A1mEv
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`
Il faut savoir que cette étape d’encodage n’est pas normalisée et est propre à chaque compilateur. Ainsi, les objets compilés avec un compilateur ne seront sans doute pas exploitable avec un autre compilateur!
En C, il n’y a pas de “mangling”, une fonction génère un symbole de même nom que cette fonction.

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c
$nm mang.o
         U __gxx_personality_v0
00000000 T _ZN1A1mEv
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`
Il faut savoir que cette étape d’encodage n’est pas normalisée et est propre à chaque compilateur. Ainsi, les objets compilés avec un compilateur ne seront sans doute pas exploitable avec un autre compilateur!
En C, il n’y a pas de “mangling”, une fonction génère un symbole de même nom que cette fonction.
Le “mangling” permet la mise en oeuvre de la surcharge, des espaces de nom, des méthodes de classes...

Héritage: et constructeur

Lors de la contruction d'une instance de la classe `Fille`, sa classe de base, la classe `Mere` doit elle aussi être initialisée.
Pour initialiser la partie `Mere`, le compilateur ajoute un appel au constructeur par défault:

```
class Mere{
  public:
  Mere(){puts("Mere");}
};
class Fille:public Mere{
  public:
  Fille(){puts("Fille");}
};
```

```
Fille f;
/*
Affiche:

Mere
Fille
*/
```

Héritage: et constructeur

Lors de la contruction d'une instance de la classe `Fille`, sa classe de base, la classe `Mere` doit elle aussi être initialisée.
Pour initialiser la partie `Mere`, le compilateur ajoute un appel au constructeur par défault:

```
class Mere{
  public:
  Mere(){puts("Mere");}
};
class Fille:public Mere{
  public:
  Fille(){puts("Fille");}
};
```

```
Fille f;
/*
Affiche:

Mere
Fille
*/
```

Comment faire si la classe `Mere` n'a pas de constructeur par défaut?

Héritage: et constructeur

On peut indique le constructeur à appeler comme ceci:

```
class Mere{
  public:
  Mere(int i){puts("Mere");}
};
class Fille:public Mere{
  public:
  Fille():Mere(1)
  {puts("Fille");}
};
```

```
Fille f;
/*
Affiche:

Mere
Fille
*/
```

L'appel au constructeur de la classe `Mere` doit se faire avant l'initialisation des attributs de la classe.
Dans le cas contraire le compilateur inverse les initialisations.

Héritage: et constructeur

```
class Mere{
  public:
  Mere(int i){
    printf("Mere: %d\n",i);
  }
};
class Fille:public Mere{
  int v;
  public:
  Fille():v(1),Mere(v)
  {puts("Fille");}
};
```

```
Fille f;
/*
Affiche:

Mere: -1479451508
Fille
*/
```

Si l'on compile avec `-Wall`, le compilateur nous signal l'inversion, sinon rien n'est dit.

Héritage: et destructeur

De même que pour le constructeur, les appels au destructeur se font en “cascade”. C’est à dire qu’à la fin du destructeur de la classe `Fille`, un appel au destructeur de la classe `Mere` est ajouté:

```
class Mere{
  public:
  Mere(){puts("Mere");}
  ~Mere(){puts("~Mere");}
};
class Fille:public Mere{
  public:
  Fille(){puts("Fille");}
  ~Fille(){puts("~Fille");}
};
```

```
{
Fille f;
}
/*
Affiche:

Mere
Fille
~Fille
~Mere
*/
```

Héritage: et opérateur d'affectation

Nous avons vu que le compilateur ajoute dans les classes un opérateur d'affectation si celui-ci n'est pas écrit.

Héritage: et opérateur d'affectation

Nous avons vu que le compilateur ajoute dans les classes un opérateur d'affectation si celui-ci n'est pas écrit.
L'opérateur ajouté par le compilateur fait appel à l'opérateur de la classe `Mere`:

```
class Mere{
  int _v;
  public:
  Mere(int v=0):_v(v){}
  const Mere &operator=(const Mere &){
    printf("Mere:%d\n",_v);
    return *this;
  }
};
class Fille:public Mere{
  Mere m;
  public:
  Fille():Mere(10){}
}
```

```
{
Fille f,g;
f=g;
}
/*
Affiche:

Mere:10
Mere:0
*/
```

Héritage: et opérateur d'affectation

Voici à quoi ressemble l'opérateur d'affectation ajouté par le compilateur:

```
class Fille: public Mere{
  ...
  public:
  const Fille &operator=(const Fille &f){
    Mere::operator=(f);
    this->attribut1=f.attribut1;
    this->attribut2=f.attribut2;
    ...
  }
};
```

Dans le cas où nous voulons écrire notre propre opérateur, c'est à nous d'appeler explicitement l'opérateur d'affectation de la classe `Mere`.

Héritage: et opérateur d'affectation

```
class A{
  public:
  const A&operator=(const A&){
    puts("A=A"); return *this;}
};
class M{
  A a;
};
class F: public M{
  public:
  const F&operator=(const F&f){
    return *this;
  }
};
```

```
F f,g;

f=g;
// Pas d'affichage!
```

Héritage: et opérateur d'affectation

```
class A{
  public:
  const A&operator=(const A&){
    puts("A=A"); return *this;}
};
class M{
  A a;
};
class F: public M{
  public:
  const F&operator=(const F&f){
    M::operator=(f);
    return *this;
  }
};
```

```
F f,g;

f=g;
// Affiche A=A
```

Héritage: et opérateur d'affectation

```
class A{
  public:
  const A&operator=(const A&){
    puts("A=A"); return *this;}
};
class M{
  A a;
};
class F: public M{
  public:
  const F&operator=(const F&f){
    M::operator=(f);
    return *this;
  }
};
```

```
F f,g;

f=g;
// Affiche A=A
```

On peut faire appel à l'opérateur ajouté par le compilateur!

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class** `T:` **public** `Base`, protected: **class** `T:` **protected** `Base` et private: **class** `T:` **private** `Base`.
L'héritage est privé par défaut dans les classes et publique pour les structures.

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class** `T:` **public** `Base`, protected: **class** `T:` **protected** `Base` et private: **class** `T:` **private** `Base`. L'héritage est privé par défaut dans les classes et publique pour les structures.

Dans le cas de l'héritage **public**, la relation entre `T` et `Base` est visible de tous. Les méthodes **public** de `Base` sont **public** dans `T` et les méthodes **protected** sont **protected** dans `T`.

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class** `T:` **public** `Base`, protected: **class** `T:` **protected** `Base` et private: **class** `T:` **private** `Base`. L'héritage est privé par défaut dans les classes et publique pour les structures.

Dans le cas de l'héritage **public**, la relation entre `T` et `Base` est visible de tous. Les méthodes **public** de `Base` sont **public** dans `T` et les méthodes **protected** sont **protected** dans `T`.

Dans le cas de l'héritage **protected**, les méthodes **public** et **protected** de `Base` sont **protected** dans `T`.

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class** `T:` **public** `Base`, protected: **class** `T:` **protected** `Base` et private: **class** `T:` **private** `Base`. L'héritage est privé par défaut dans les classes et publique pour les structures.

Dans le cas de l'héritage **public**, la relation entre `T` et `Base` est visible de tous. Les méthodes **public** de `Base` sont **public** dans `T` et les méthodes **protected** sont **protected** dans `T`.

Dans le cas de l'héritage **protected**, les méthodes **public** et **protected** de `Base` sont **protected** dans `T`.

Dans le cas de l'héritage **protected**, les méthodes **public** et **protected** de `Base` sont **private** dans `T`.

Héritage: public, protected et private

On ne peut faire utiliser un relation type/sous-type que si l'on peut avoir accès à un membre publique de la classe `Base` à travers `T`.

Héritage: public, protected et private

On ne peut faire utiliser un relation type/sous-type que si l'on peut avoir accès à un membre publique de la classe `Base` à travers `T`.

Si l'héritage est publique, tout le monde voit les méthodes publiques de `Base` à travers `T`. Ainsi on pourra toujours voir un `T` comme un `Base`.

Héritage: public, protected et private

On ne peut faire utiliser un relation type/sous-type que si l'on peut avoir accès à un membre publique de la classe `Base` à travers `T`.

Si l'héritage est publique, tout le monde voit les méthodes publiques de `Base` à travers `T`. Ainsi on pourra toujours voir un `T` comme un `Base`.

Si l'héritage est protégé alors seules les fonctions/classes amies, sous-classes et amies des sous-classes pourront voir un `T` comme un `Base`.

Héritage: public, protected et private

On ne peut faire utiliser un relation type/sous-type que si l'on peut avoir accès à un membre publique de la classe `Base` à travers `T`.

Si l'héritage est publique, tout le monde voit les méthodes publiques de `Base` à travers `T`. Ainsi on pourra toujours voir un `T` comme un `Base`.

Si l'héritage est protégé alors seules les fonctions/classes amies, sous-classes et amies des sous-classes pourront voir un `T` comme un `Base`.

Si l'héritage est privé alors seules les fonctions/classes amies pourront voir un `T` comme un `Base`.

Héritage: public, protected et private

```
class A{
...
};
class B: public A{
...
};
class C: protected A{
...
};
class D: public C{
...
};
class E: private A{
...
};
class F: public E{
...
};
```

```
B b;
A&a=b; // ok

C c;
A *p=&c; // erreur!

void D::f(){
  A &r=*this; // ok
}

void F::f(){
  A &r=*this; // erreur
}

void E::f(){
  A &r=*this; // ok
}
```

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récuperer une implémentation.

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récuperer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récuperer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Un exemle: on veut implémenter une classe `Pile`. Pour cela on souhaite utiliser la classe `Vector`.
Or, une pile n'est pas un vecteur particulier, on ne souhaite donc pas que la classe `Pile` présente toutes les méthodes de la classe `Vector`.

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récuperer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Un exemle: on veut implémenter une classe `Pile`. Pour cela on souhaite utiliser la classe `Vector`.
Or, une pile n'est pas un vecteur particulier, on ne souhaite donc pas que la classe `Pile` présente toutes les méthodes de la classe `Vector`.

La solution sans héritage privé consiste à utiliser la délégation: on dispose d'un attribut de type `Vector` que l'on utilise pour implanter la pile.

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récuperer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Un exemle: on veut implémenter une classe `Pile`. Pour cela on souhaite utiliser la classe `Vector`.
Or, une pile n'est pas un vecteur particulier, on ne souhaite donc pas que la classe `Pile` présente toutes les méthodes de la classe `Vector`.

La solution sans héritage privé consiste à utiliser la délégation: on dispose d'un attribut de type `Vector` que l'on utilise pour implanter la pile.

Si l'on souhaite que cet "délégation" soit accessible aux sous classes alors on utilise l'héritage protégé sinon on utilise l'héritage privé.

Héritage: et polymorphisme

Jusqu'à présent, l'héritage permet le factorisation de code mais ne permet pas de mettre en oeuvre le polymorphisme.
En effet, pour qu'il y ai polymorphisme il faut que la méthode appelée sur une instance soit celle correspondant au type réel de l'instance et non au type de la variable manipulant cette instance.

Héritage: et polymorphisme

Jusqu'à présent, l'héritage permet le factorisation de code mais ne permet pas de mettre en oeuvre le polymorphisme.
En effet, pour qu'il y ai polymorphisme il faut que la méthode appelée sur une instance soit celle correspondant au type réel de l'instance et non au type de la variable manipulant cette instance.

Pour mettre en oeuvre le polymorphisme il faut donc utiliser l'instance elle-même pour connaître son type.

Héritage: et polymorphisme

Jusqu'à présent, l'héritage permet le factorisation de code mais ne permet pas de mettre en oeuvre le polymorphisme.
En effet, pour qu'il y ai polymorphisme il faut que la méthode appelée sur une instance soit celle correspondant au type réel de l'instance et non au type de la variable manipulant cette instance.

Pour mettre en oeuvre le polymorphisme il faut donc utiliser l'instance elle-même pour connaître son type.

Ceci est implanté grace à un attribut caché appelé `vtable`. C'est un pointeur vers une table référençant toutes les méthodes pour lesquelles le polymorphisme doit être mis en oeuvre.

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes "virtuelles", c'est à dire pour lesquelles on veut du polymorphisme.

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes "virtuelles", c'est à dire pour lesquelles on veut du polymorphisme.

```
class Base {

public:
  virtual std::string toString() const {
    return "Base"
  }
};
```

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes "virtuelles", c'est à dire pour lesquelles on veut du polymorphisme.

```
class Base {

public:
  virtual std::string toString() const {
    return "Base"
  }
};
```

On notera que la présence de la `vtable` fait "grossir" les instances de la classe: on passe ici de 1 octet à 4 octets (sur un machine 32 bits).

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes "virtuelles", c'est à dire pour lesquelles on veut du polymorphisme.

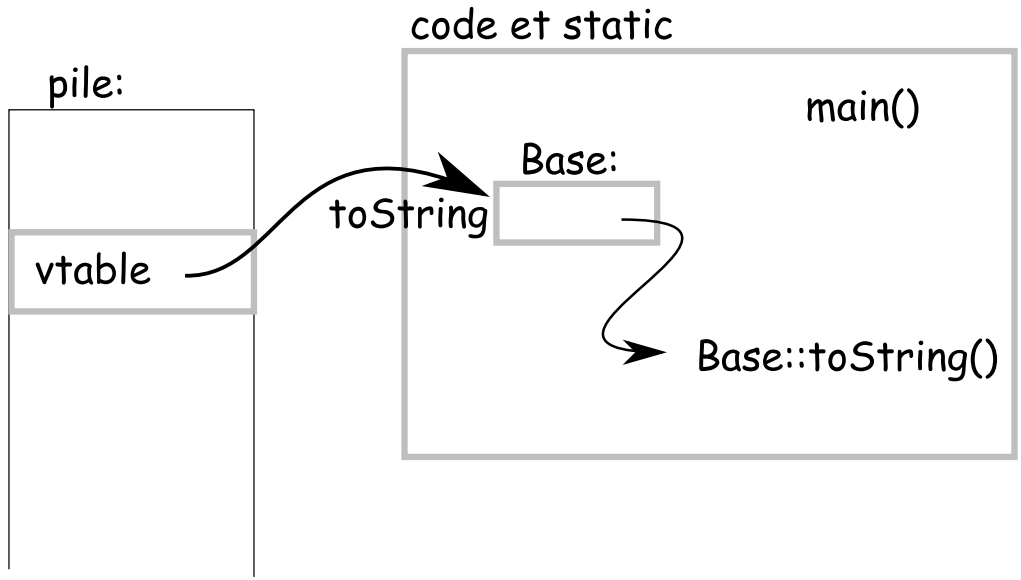
```
class Base {

public:
   virtual std::string toString() const {
     return "Base"
   }
};
```

On notera que la présence de la `vtable` fait "grossir" les instances de la classe: on passe ici de 1 octet à 4 octets (sur un machine 32 bits).

Ceci crée une table associée à la classe Base. Cette table est une table de pointeurs de fonctions: les méthodes virtuelles

Héritage: vtable

```
Base b;
```



Héritage: vtable

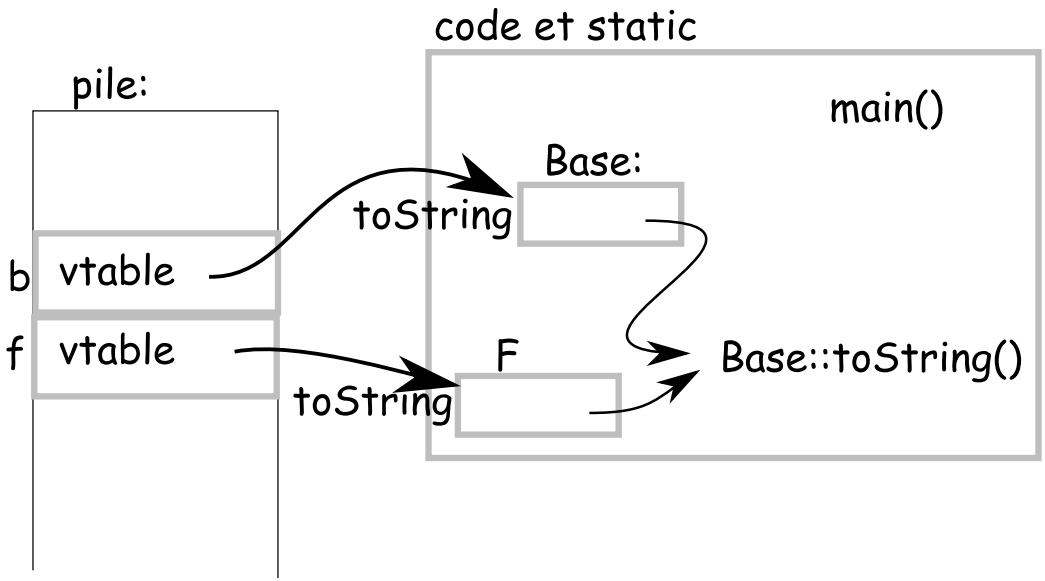
Ajoutons une classe dérivée de `Base`:

```
class F : public Base{
  public:
};
```

Si l'on ne redéfini pas la méthode `toString` alors on obtient le schéma suivant

Héritage: vtable

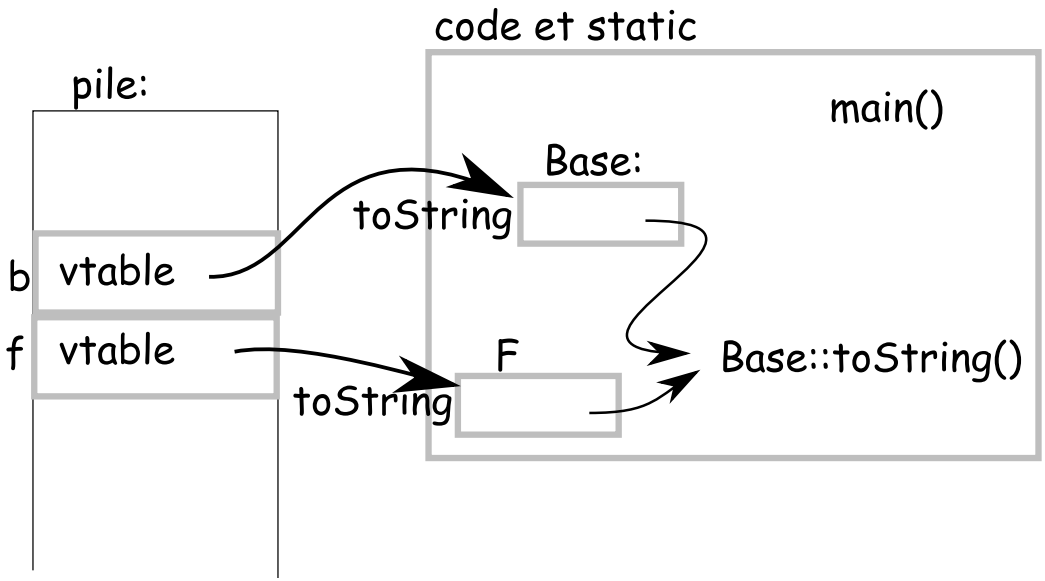
```
Base b;
F f;
```



Une table des fonctions virtuelles est créée pour la classe `F`. Celle-ci est initialement construite par “recopie” de la table de la classe de base.

Héritage: vtable

```
Base b;
F f;
```



Une table des fonctions virtuelles est créée pour la classe `F`. Celle-ci est initialement construite par “recopie” de la table de la classe de base.

Dans les constructeurs, le compilateur ajoute une instruction permettant d’initialiser l’attribut caché `vtable` pour qu’il pointe vers la table correspondant au type de l’instance.

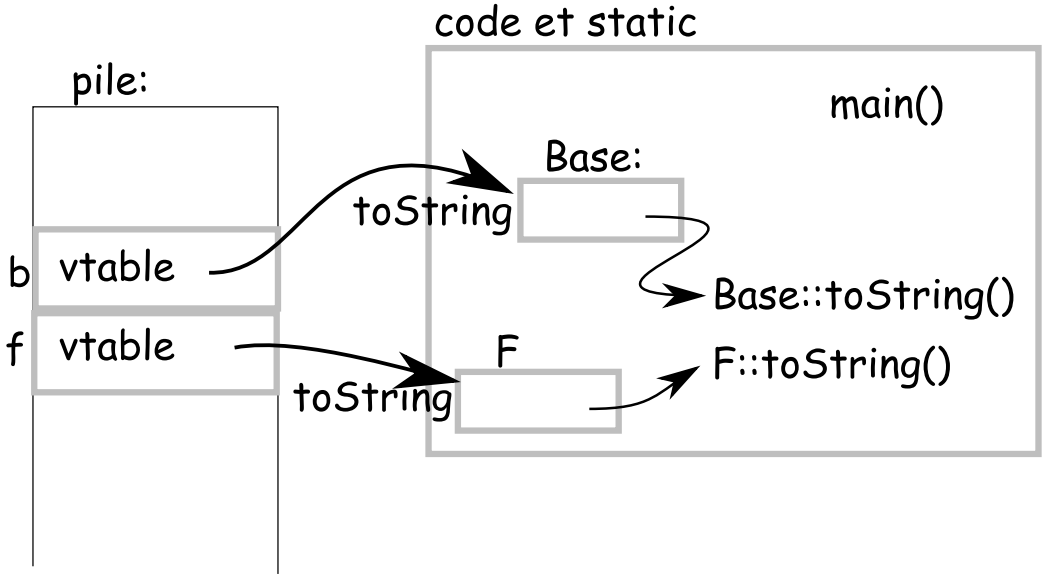
Héritage: vtable

Cas où l'on redéfini la méthode dans la classe `F`:

```
class F : public Base{
  public:
  virtual std::string toString() const{
    return "Fille";
  }
};
```

Héritage: vtable

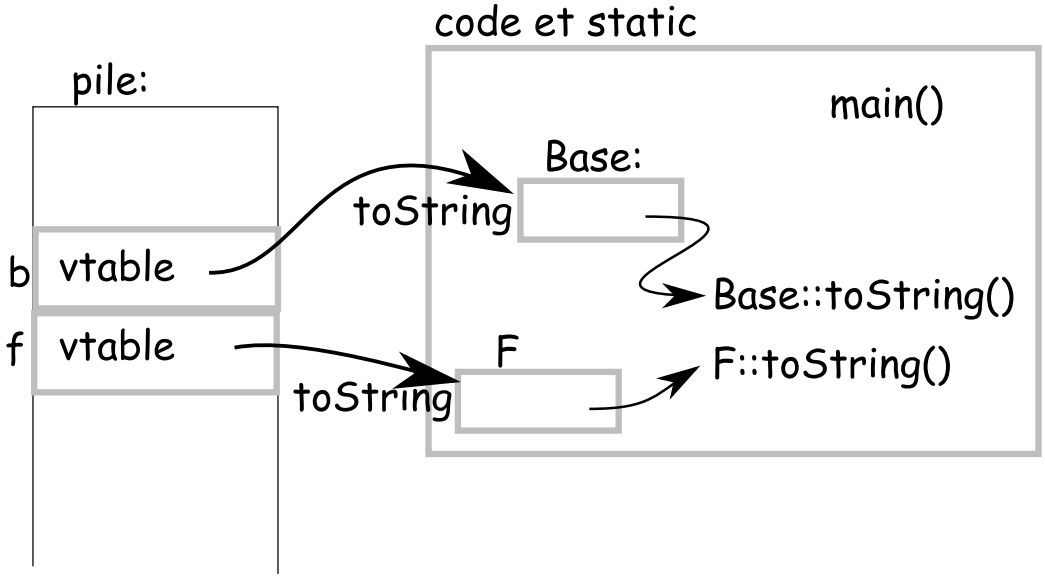
```
Base b;
F f;
```



Si une méthode est redéfinie: même nom, même arguments (même type) alors l'entrée dans la pointeur dans la table correspondant à cette méthode est modifiée.

Héritage: vtable

```
Base b;
F f;
```



Si une méthode est redéfinie: même nom, même arguments (même type) alors l'entrée dans la pointeur dans la table correspondant à cette méthode est modifiée.
On voit que ce mécanisme ne modifie en rien le code de la méthode, c'est la façon dont la méthode va être appelée qui va être changé.

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;
Base &r=f;

r.toString();
```

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;
Base &r=f;

r.toString();
```

Une fois cette méthode trouvée, il y a deux possibilités: soit la méthode est virtuelle, soit elle ne l'ai pas.

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;
Base &r=f;

r.toString();
```

Une fois cette méthode trouvée, il y a deux possibilités: soit la méthode est virtuelle, soit elle ne l'ai pas.

Si elle n'est pas virtuelle, alors c'est la méthode du type de la variable qui est appelée, dans l'exemple `Base::toString`.

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

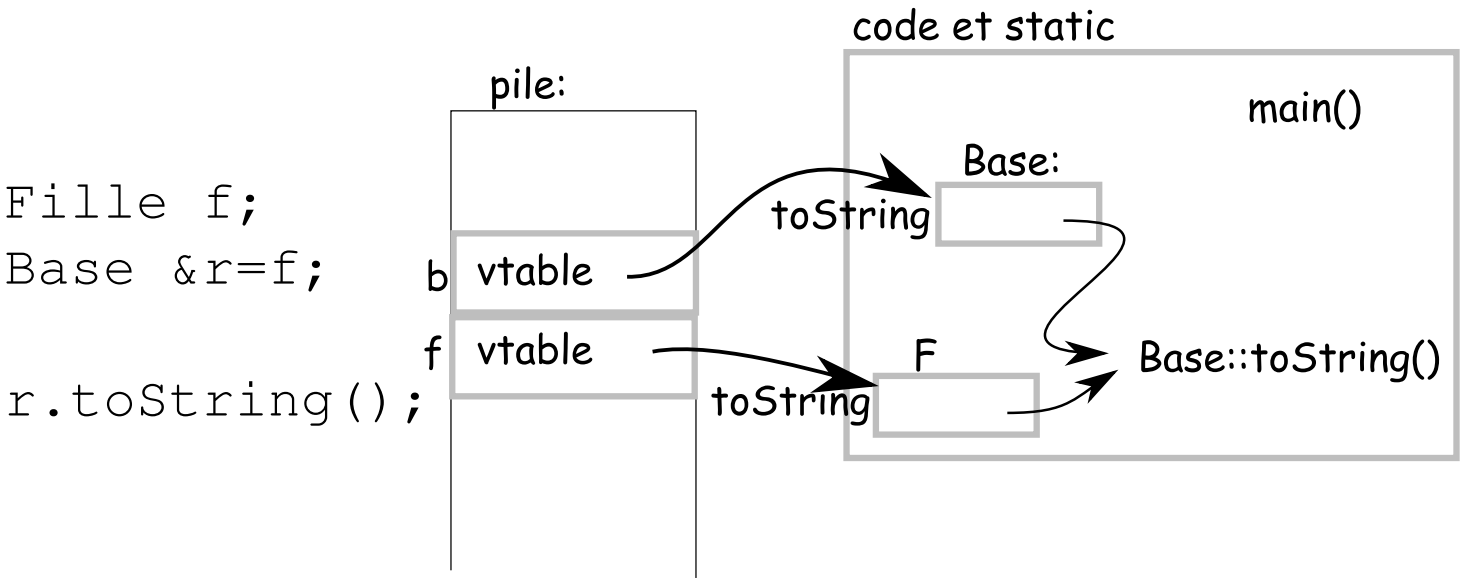
```
Fille f;
Base &r=f;

r.toString();
```

Une fois cette méthode trouvée, il y a deux possibilités: soit la méthode est virtuelle, soit elle ne l'ai pas.

Si elle n'est pas virtuelle, alors c'est la méthode du type de la variable qui est appelée, dans l'exemple `Base::toString`.

Si elle est virtuelle, alors c'est la méthode dont l'adresse est dans la `vtable` qui est appelée, dans l'exemple cela dépend si elle a été redéfinie.

Héritage: vtable

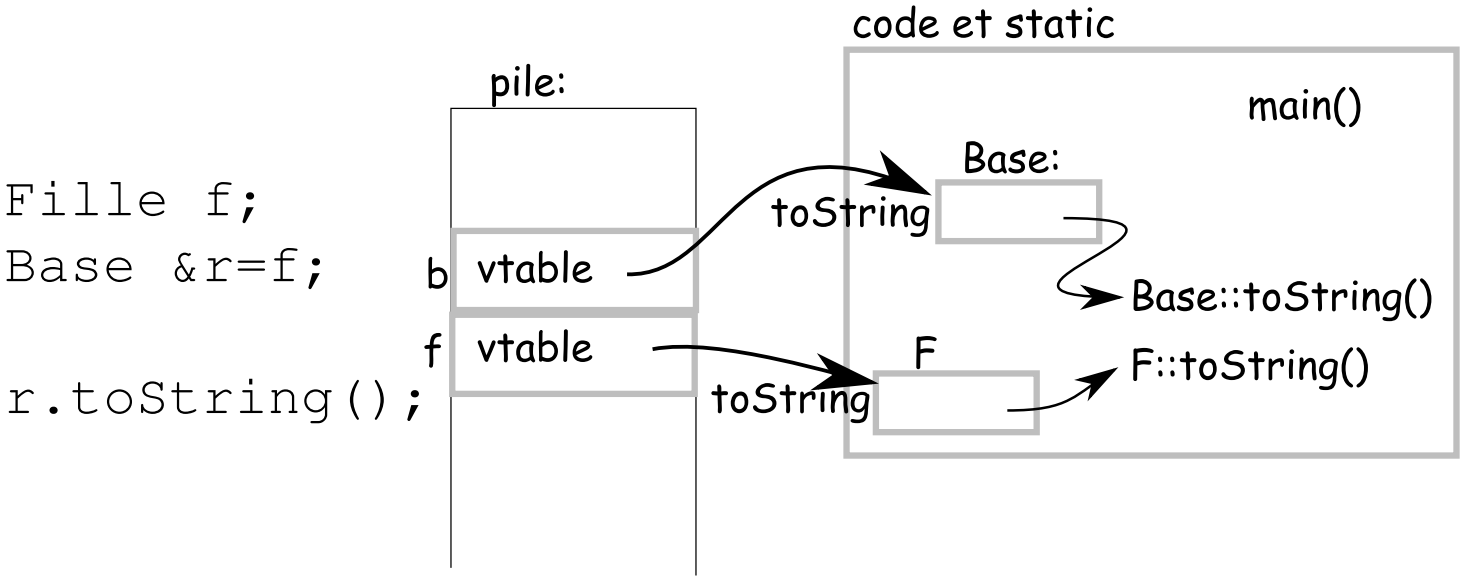


L'appel à la méthode peut être vu comme

`t.vtable["toString"]();`

Dans cet exemple, cela correspond à l'appel de `Base::toString()` car la méthode n'a pas été redéfinie.

Héritage: vtable



L'appel à la méthode peut être vu comme

`t.vtable["toString"]();`

Dans cet exemple, cela correspond à l'appel de `Fille::toString()` car la méthode a été redéfinie.

Héritage: virtuelle pure

Une méthode est dite “virtuelle pure” si elle n’est pas implémentée dans la classe où elle est déclarée.

```
classe Base{
   public:
   virtual std::string toString() const =0;
};
```

Le `=0` est bien compréhensible: cela correspond à mettre le pointeur dans la table des fonctions virtuelles à 0!

Héritage: virtuelle pure

Une méthode est dite “virtuelle pure” si elle n’est pas implémentée dans la classe où elle est déclarée.

```
classe Base{
   public:
   virtual std::string toString() const =0;
};
```

Le `=0` est bien compréhensible: cela correspond à mettre le pointeur dans la table des fonctions virtuelles à 0!

Une classe n’est instanciable que si sa table des fonctions virtuelles ne comporte pas de 0.

Héritage: virtuelle pure

Une méthode est dite “virtuelle pure” si elle n’est pas implémentée dans la classe où elle est déclarée.

```
classe Base{
  public:
  virtual std::string toString() const =0;
};
```

Le `=0` est bien compréhensible: cela correspond à mettre le pointeur dans la table des fonctions virtuelles à 0!

Une classe n’est instanciable que si sa table des fonctions virtuelles ne comporte pas de 0.

Dans cette exemple, `Base` n’est donc pas instanciable et toute sous classe de `Base` ne sera instanciable que si elle redéfinie la méthode `toString`.

Héritage: virtuelle pure

Soit le code suivant:

```
class Base{
  public:
  virtual void m(const Base &)=0;
};

class Fille{
  public:
  virtual void m(const Fille &){}
}
```

Question:

- est-ce que `Base` est instanciable?
- est-ce que `Fille` est instanciable?

Héritage: virtuelle pure

Soit le code suivant:

```
class Base{
  public:
  virtual void m(const Base &)=0;
};

class Fille{
  public:
  virtual void m(const Fille &){}
}
```

Question:

- est-ce que `Base` est instanciable?
- est-ce que `Fille` est instanciable?

Les deux classes ne sont pas instanciables car la méthode `Fille::m` est une surcharge et non une redéfinition.

Héritage: et redéfinition

La redéfinition d'une méthode consiste à écrire dans une classe dérivée une méthode qui

- porte le même nom
- a le même nombre et type d'arguments
- a comme type de retour un sous-type du type de retour de la méthode redéfinie ou le même type.

Héritage: et valeur par défaut

En C++, les arguments des fonctions et méthodes peuvent prendre des valeurs par défaut pour les arguments:

```
void f(int =0){}

f(); // Appel f(0);
```

Les valeurs par défaut doivent être précisées dans la déclaration, en effet elle indique au compilateur comment compléter l'appel avec ces valeurs si necessaire.

Héritage: et valeur par défaut

En C++, les arguments des fonctions et méthodes peuvent prendre des valeurs par défaut pour les arguments:

```
void f(int =0){}

f(); // Appel f(0);
```

Les valeurs par défaut doivent être précisées dans la déclaration, en effet elle indique au compilateur comment compléter l'appel avec ces valeurs si necessaire.

Les valeurs par défaut doivent être données du dernier argument vers le premier argument afin d'éviter toute ambiguité.

Héritage: et valeur par défaut

```
class M{
public:
   virtual ~M(){}
   virtual void m(int =0)=0;
};
class F: public M{
   public:
   virtual void m(int i=1){}
};
```

```
int main()
{
  F f;
  M &m=f;

  m.m(); // appel de F::m(0);
  f.m(); // appel de F::m(1);
}
```

Bien que l'appel soit virtuel, c'est le type de la variable qui est utilisé pour connaître les valeurs par défaut des arguments!

Héritage: et valeur par défaut

```
class M{
public:
   virtual ~M(){}
   virtual void m(int =0)=0;
};
class F: public M{
   public:
   virtual void m(int i){}
};
```

```
int main()
{
  F f;
  M &m=f;

  m.m(); // appel de F::m(0);
  f.m(); // Erreur !
}
```

D'une manière générale on evitera de mélanger redéfinition, surcharge et valeur par défaut: il faut toujours (c'est aussi valable pour les opérateurs, exceptions...) que le comportement attendu soit "limpide".

Héritage: et using

Lorsque l'on mélange redéfinition et surchage il peut arriver des choses "bizarre":

```
class M{
  public:
  void m(const M&);
};
class F: public M{
  public:
  void m(const F&);
};
```

```
F f,g;
M &m=f,&n=g;
f.m(g); // 1. ok
m.m(n); // 2. ok
m.m(f); // 3. ok
f.m(m); // 4. erreur!
```

Héritage: et using

Lorsque l'on mélange redéfinition et surchage il peut arriver des choses "bizarre":

```
class M{
  public:
  void m(const M&);
};
class F: public M{
  public:
  void m(const F&);
};
```

```
F f,g;
M &m=f,&n=g;
f.m(g); // 1. ok
m.m(n); // 2. ok
m.m(f); // 3. ok
f.m(m); // 4. erreur!
```

Quelles méthodes sont appelées en 1,2 et 3 ?

Héritage: et using

Lorsque l'on mélange redéfinition et surchage il peut arriver des choses "bizarre":

```
class M{
  public:
  void m(const M&);
};
class F: public M{
  public:
  void m(const F&);
};
```

```
F f,g;
M &m=f,&n=g;
f.m(g); // 1. ok
m.m(n); // 2. ok
m.m(f); // 3. ok
f.m(m); // 4. erreur!
```

Quelles méthodes sont appelées en 1,2 et 3 ?

Pour le cas 4, comme `F` hérite de `M` on pourrait s'attendre à ce que la méthode `M::m` soit appelée.

Le problème ici est lié à l'algorithme de résolution des appels et à la surcharge.

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` contenant une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` contenant une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?
- Oui (partie 1):
  - Si une méthode est compatible: elle est choisie
  - Si plusieurs méthodes sont compatibles: ambiguité
  - Si aucune méthode n'est compatible: erreur.

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` contenant une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?
- Oui (partie 1):
  - Si une méthode est compatible: elle est choisie
  - Si plusieurs méthodes sont compatibles: ambiguité
  - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
  - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` contenant une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?
- Oui (partie 1):
  - Si une méthode est compatible: elle est choisie
  - Si plusieurs méthodes sont compatibles: ambiguité
  - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
  - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` contenant une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?
- Oui (partie 1):
  - Si une méthode est compatible: elle est choisie
  - Si plusieurs méthodes sont compatibles: ambiguité
  - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
  - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).
- Si elle est virtuelle:
  - On effectue l'appel en utilisant la `vtable`
  - Si la méthode a été redéfinie dans `I` alors c'est `m::I` qui est appelée.
  - Sinon c'est la méthode du plus proche parent.

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` contenant une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?
- Oui (partie 1):
  - Si une méthode est compatible: elle est choisie
  - Si plusieurs méthodes sont compatibles: ambiguité
  - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
  - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).
- Si elle est virtuelle:
  - On effectue l'appel en utilisant la `vtable`
  - Si la méthode a été redéfinie dans `I` alors c'est `m::I` qui est appelée.
  - Sinon c'est la méthode du plus proche parent.
- Sinon on appel `m::V`

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` contenant une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?⇒ using
- Oui (partie 1):
  - Si une méthode est compatible: elle est choisie
  - Si plusieurs méthodes sont compatibles: ambiguité
  - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
  - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).
- Si elle est virtuelle:
  - On effectue l'appel en utilisant la `vtable`
  - Si la méthode a été redéfinie dans `I` alors c'est `m::I` qui est appelée.
  - Sinon c'est la méthode du plus proche parent.
- Sinon on appel `m::V`

Héritage: et using

On peut utiliser le mot clé **using** pour inclure des méthodes parent dans les méthodes à inspecter:

```
class M{
  public:
  void m(const M&);
};
class F: public M{
  public:
  using M::m;
  void m(const F&);
};
```

```
F f,g;
f.m(g); // 1. ok
M &m=f,&n=g;
m.m(n); // 2. ok
m.m(f); // 3. ok
f.m(m); // 4. ok
```

Le **using** `M::m;` dans la classe `F` indique au compilateur d'inclure la ou les méthodes de nom `m` dans la classe `M` dans l'algorithme de résolution.

Héritage: classe abstraite

On appel une classe abstraite une classe ayant au moins une méthode virtuelle pure.

Héritage: classe abstraite

On appel une classe abstraite une classe ayant au moins une méthode virtuelle pure.

En C++, il n'existe pas de définition d'interface (dans la norme), cependant on pourra appeler interface une classe ne disposant que de méthodes virtuelles pures (et un destructeur virtuel).

Héritage: classe abstraite

On appel une classe abstraite une classe ayant au moins une méthode virtuelle pure.

En C++, il n'existe pas de définition d'interface (dans la norme), cependant on pourra appeler interface une classe ne disposant que de méthodes virtuelles pures (et un destructeur virtuel).

Les classes abstraites et les interfaces ne sont pas instanciables.

Une classe abstraite ou interface ne peuvent pas être utilisées comme type de retour par recopie ou argument par valeur (recopie).

On ne peut utiliser que des pointeurs et références sur ces types.

Héritage: et interface

Exemple d'une interface:

```
class Usager{
  public:
  virtual std::string nom() const=0;
  virtual void monterDans(Transport &)=0;
};
```

Héritage: et classe abstraite

Exemple d'une classe abstraite:

```
class PassagerAbstrait{
  protected:
  std::string _nom;
  int _etat;
  int _destination;
  ...
  virtual choixPlaceMontee(Bus &b)=0;
  ...
  public:
  void estDebout() const {
    return _etat==DEBOUT;
  }
};
```

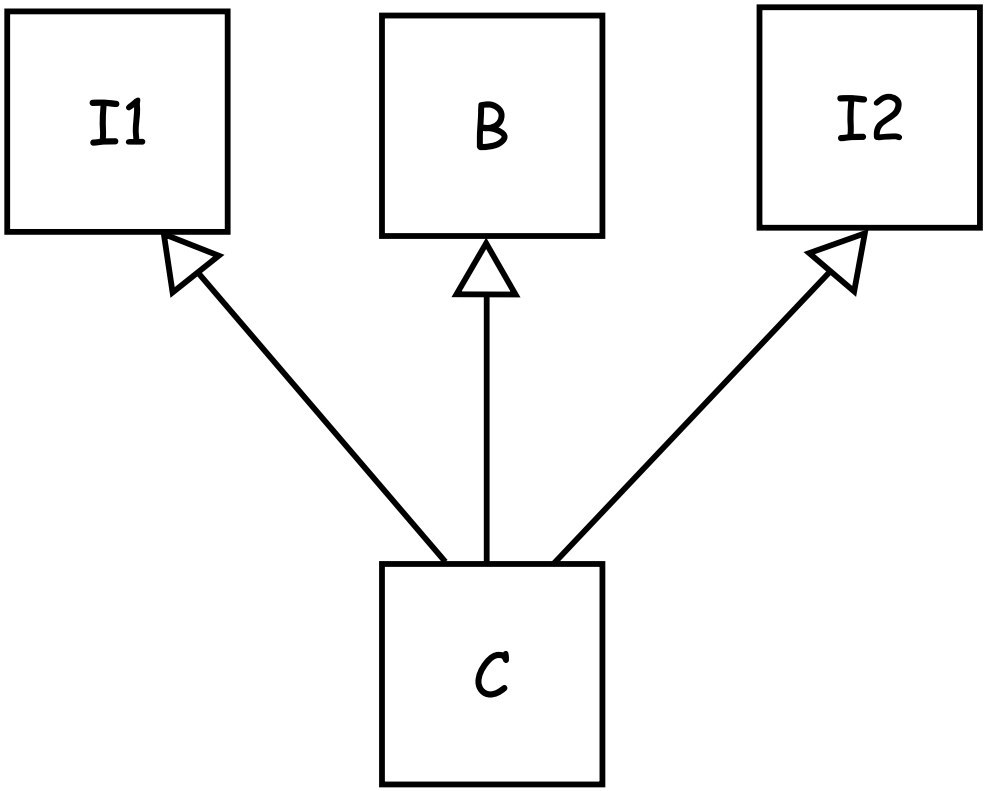
Héritage multiple

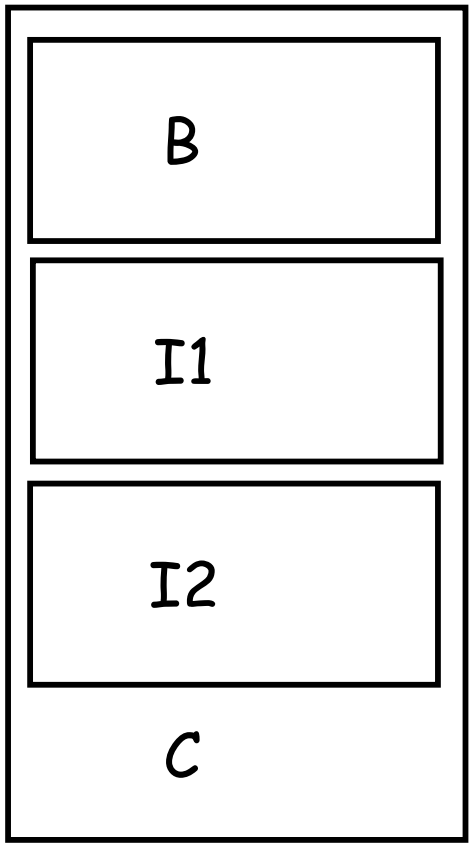
En C++, une classe peut avoir plusieurs classes de base, que celle-ci soient abstraites ou non:

```
class I1{ ... };
class I2{ ... };
class B{ ...};
class C: public B, public I1, public I2 {
  // implémentation de toutes le méthodes
  // virtuelles pures.
};
```

La classe `C` hérite ici de trois classes. Ainsi, on pourra se représenter `C` en mémoire de la façon suivante:

Hérite multiple: exemple





Héritage multiple

L'opérateur de porté et le **using** sont très utils pour lever les ambiguités:

```
class I1{
  public:
  void m();
};
class I2{
  public:
  void m();
};

class F: public I1, public I2{
  ....
  m(); // Erreur!
  I1::m(); // ok
  ...
};
```

Héritage multiple

L'opérateur de porté et le **using** sont très utils pour lever les ambiguités:

```
class I1{
  public:
  void m();
};
class I2{
  public:
  void m();
};

class F: public I1, public I2{
  ....
  using I2::m;
  m(); // ok
  I1::m(); // ok
  ...
};
```

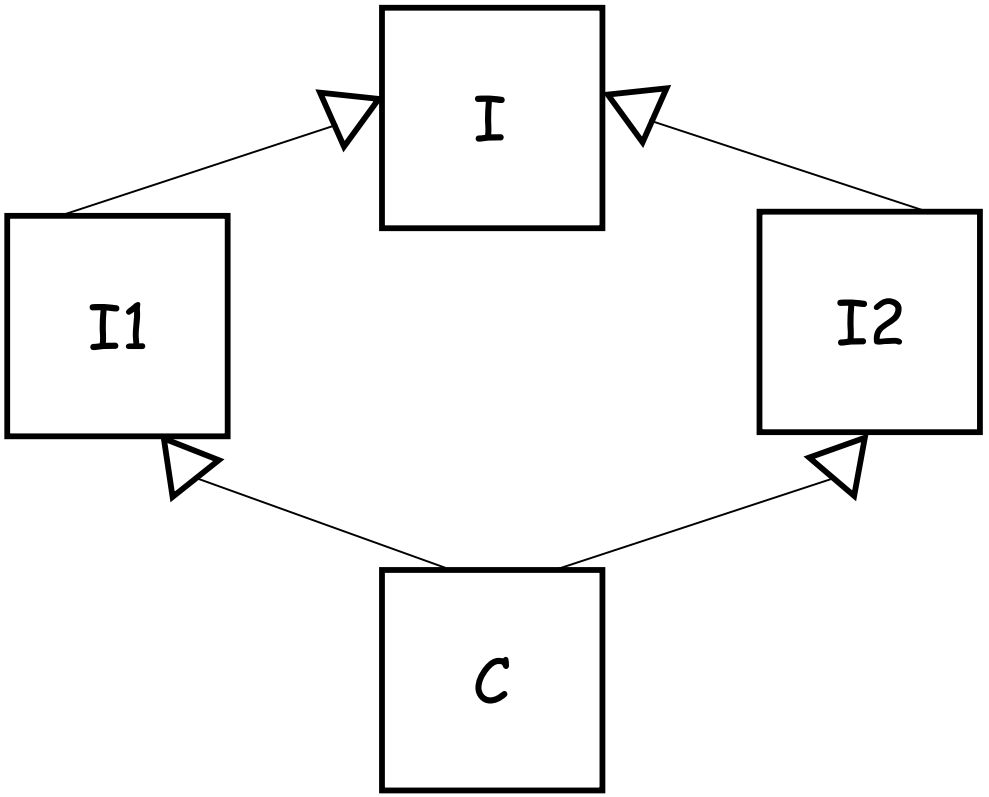
Héritage multiple virtuel

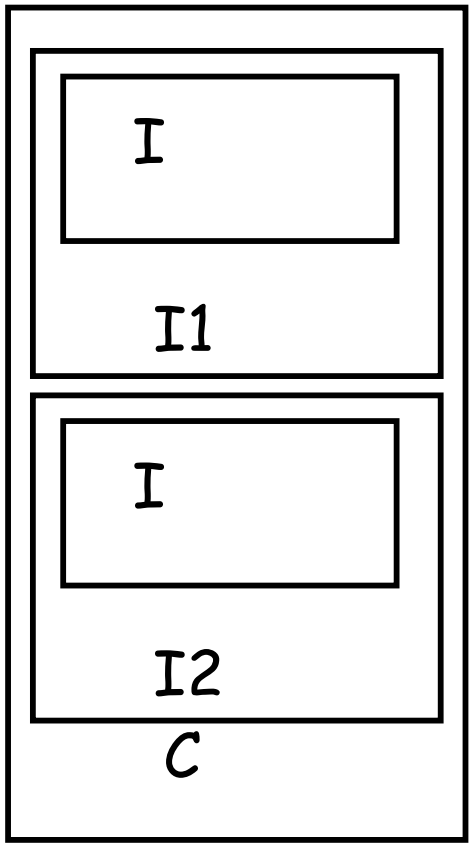
Il peut arriver qu'une classe possède plusieurs fois une même classe comme classe de base:

```
class I{ public: int _value; };

class I1: public I{};

class I2: public I{};

class C: public I1, public I2{};
```

Dans ce cas, C dispose de deux instances de la classe I:

Hérite multiple: exemple





Héritage multiple: ambiguïté

Ainsi, une instance de la classe `C` à deux attributs `_value`:

```
C c;
c._value ; // Erreur!
```

L'opérateur de porté permet de lever cette ambiguïté:

```
C c;
c.I1::_value ; // Ok
c.I1::I::_value ; // Ok
c.I2::_value ; // Ok
```

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Pour cela, on utilise l'héritage virtuel.

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Pour cela, on utilise l'héritage virtuel.

L'héritage virtuel permet d'indiquer qu'une classe est prête à partager une de ces classes de base avec une autre classe qui aurait fait de même:

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Pour cela, on utilise l'héritage virtuel.

L'héritage virtuel permet d'indiquer qu'une classe est prête à partager une de ces classes de base avec une autre classe qui aurait fait de même:

```
class I{};
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};
```

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

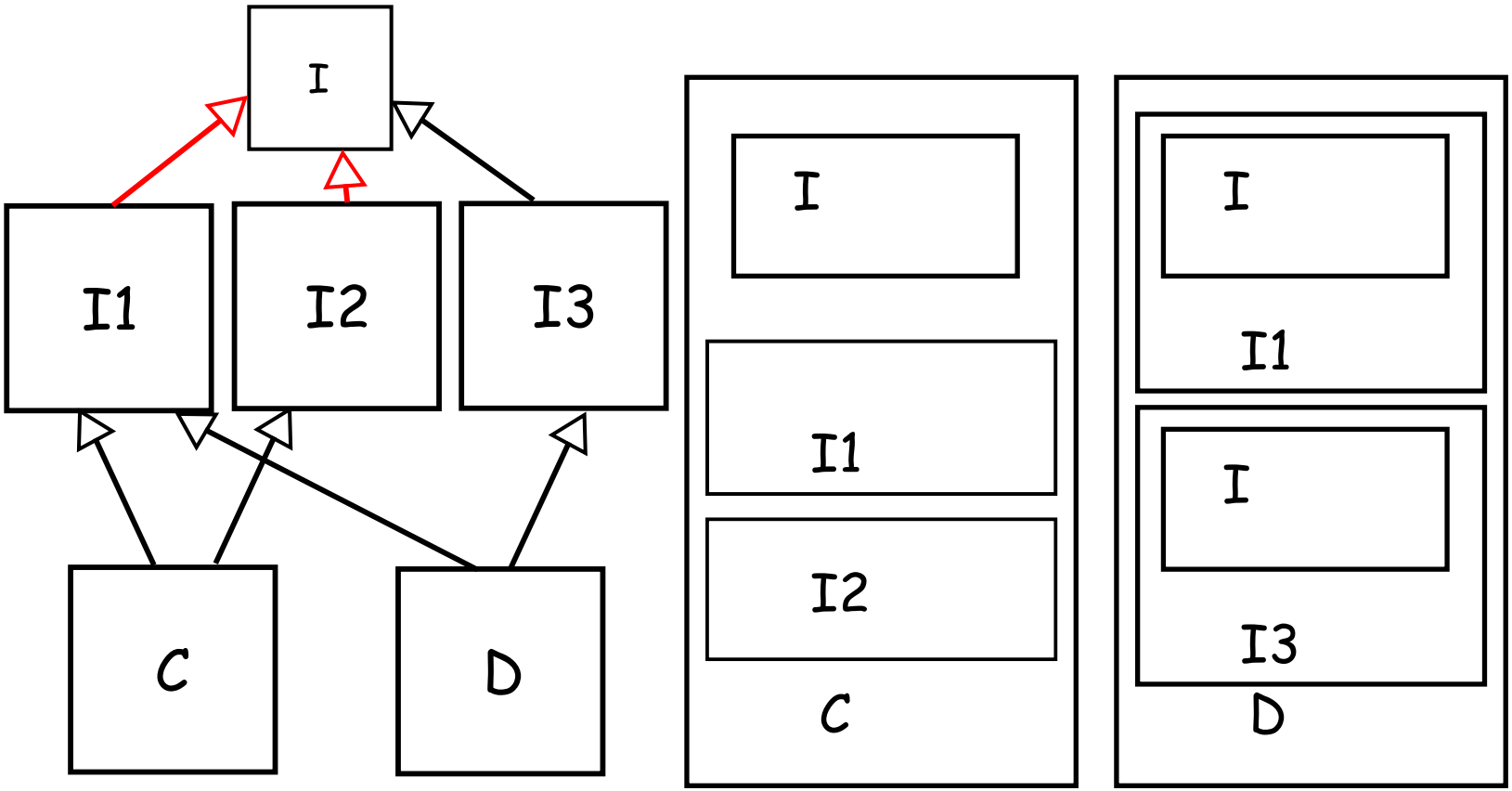
Pour cela, on utilise l'héritage virtuel.

L'héritage virtuel permet d'indiquer qu'une classe est prête à partager une de ces classes de base avec une autre classe qui aurait fait de même:

```
class I{};
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};
```

Une instance de la classe `C` ne contient qu'un seul `I`
Une instance de la classe `D` en possède deux

Hérite multiple: virtuel



Hérite multiple: virtuel

L'héritage virtuel permet de lever des ambiguïtés:

```
class I{ public: virtual void m()=0; };
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};

C c;
c.m(); // ok
D d;

d.m(); // Erreur!
```

On utilisera systématiquement l'héritage virtuel lors de l'héritage d'une interface.

Hérite multiple: virtuel

L'héritage virtuel permet de lever des ambiguïtés:

```
class I{ public: virtual void m()=0; };
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};

C c;
c.m(); // ok
D d;

d.m(); // Erreur!
```

On utilisera systématiquement l'héritage virtuel lors de l'héritage d'une interface.

Dans le cas ci-dessus, `I` doit bien entendu posséder un destructeur virtuel.

Hérite multiple: virtuel

D'une façon générale, on n'utilisera pas l'héritage virtuel si la classe de base possède des propriétés (attributs) pouvant changer.

On pourra s'autoriser l'héritage virtuel si la classe de base est un interface, ou bien une classe ne possédant que des méthodes virtuelles pures, un constructeur par défaut et des attributs dont la valeur est fixée dans le constructeur.

Plan

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

La programmation générique consiste à écrire du code (fonction ou classe) indépendamment de certain type. Ceux-ci seront fixés plus tard lors de l'utilisation de ce code.

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

La programmation générique consiste à écrire du code (fonction ou classe) indépendamment de certain type. Ceux-ci seront fixés plus tard lors de l'utilisation de ce code.

En C++ c'est le mot clé **template** qui permet la définition de la généricité

Template: présentation

En plus du support de la programmation objet, le C++ ajoute au C la programmation générique.

La programmation générique consiste à écrire du code (fonction ou classe) indépendamment de certain type. Ceux-ci seront fixés plus tard lors de l'utilisation de ce code.

En C++ c'est le mot clé **template** qui permet la définition de la généricité

On préfixera le code générique d'une intruction **template**<**class** T> où T devient un type variable, paramètre du code.

Template: présentation

On pourra écrire deux types de code générique: des classes génériques et des fonctions génériques.
Voici un exemple de fonction générique:

```
template<class T>
void swap(T &a, T&b){
  T t=a;
  a=b;
  b=t;
}
```

Cette fonction effectue l'echange de valeur entre `a` et `b` quel que soit leur type.
On voit cependant que `a` et `b` doivent être de même type

L'utilisation de cette fonction `swap` se fait comme suit:

```
template<class T>
void swap(T &a, T&b){...}

int a=0;
int b=1;
char c='a';
```

```
swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

L'utilisation de cette fonction `swap` se fait comme suit:

```
template<class T>
void swap(T &a, T&b){...}

int a=0;
int b=1;
char c='a';
```

```
swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

Dans le cas 1, nous indiquons explicitement le type pour `T`, le compilateur remplace alors `T` par **int** et trouve la fonction `swap<int>(int &,int &)` qui convient à l'appel. Par la même occasion, il compile cette fonction.

L'utilisation de cette fonction `swap` se fait comme suit:

```
template<class T>
void swap(T &a, T&b){...}

int a=0;
int b=1;
char c='a';
```

```
swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

Dans le cas 1, nous indiquons explicitement le type pour `T`, le compilateur remplace alors `T` par **int** et trouve la fonction `swap<int>(int &,int &)` qui convient à l'appel. Par la même occasion, il compile cette fonction.

Dans le cas 2, le compilateur cherche la fonction `swap` et essaye de déduire un type convenable pour `T` à partir des arguments (ceci n'est pas toujours possible). Il construit `swap<int>` et compile cette fonction.

L'utilisation de cette fonction `swap` se fait comme suit:

```
template<class T>
void swap(T &a, T&b){...}

int a=0;
int b=1;
char c='a';
```

```
swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

Dans le cas 1, nous indiquons explicitement le type pour `T`, le compilateur remplace alors `T` par **int** et trouve la fonction `swap<int>(int &,int &)` qui convient à l'appel. Par la même occasion, il compile cette fonction.

Dans le cas 2, le compilateur cherche la fonction `swap` et essaye de déduire un type convenable pour `T` à partir des arguments (ceci n'est pas toujours possible). Il construit `swap<int>` et compile cette fonction.

Dans le cas 3, il n'arrive pas à trouver une fonction `swap` avec le bon prototype et génére une erreur.

Template: et compilation

Un problème se pose pour la compilation d'un code template.

En effet, celui-ci étant à trous, il ne peut être compiler qu'une fois les types de paramètres connus.

Pour cela, on écrira ne peut plus seulement mettre le prototype dans le `.hpp`, nous metterons aussi le code de la fonction:

en une fois:

```
template<class T>
void swap(T &a, T& b){
  T t=a;
  a=b;
  b=t;
}
```

en deux fois:

```
template<class T>
void swap(T &, T& );

...

template<class T>
void swap(T &a, T& b){
  T t=a;a=b;b=t;
}
```

Nous verrons une alternative en TD.

Template: classe

Voici un exemple de classe paramétrée (ou générique):

```
template<class T>
class Entier{
  T _value;
  public:
  Entier(const T &t):_value(t){}
};
```

Template: classe

Voici un exemple de classe paramétrée (ou générique):

```
template<class T>
class Entier{
  T _value;
  public:
  Entier(const T &t):_value(t){}
};
```

Dans le cas d'une classe, on écrira donc le code directement dans le fichier `.hpp`

Template: classe

Voici un exemple de classe paramétrée (ou générique):

```
template<class T>
class Entier{
  T _value;
  public:
  Entier(const T &t):_value(t){}
};
```

Dans le cas d'une classe, on écrira donc le code directement dans le fichier `.hpp`

On peut tout à fait séparer les déclarations de l'implémentation:

```
template<class T>
class Entier{
  T _value;
  public:
  Entier(const T &t);
};
```

```
template<class T>
Entier<T>::Entier(const T &t):_value(t)
{}
```

Template: classe

Lors de l'utilisation d'une classe générique, nous devons indiquer explicitement le type à utiliser pour les paramètres:

```
Entier a; // Erreur!
Entier<int> b; // ok
Entier<long> c; // ok
```

On parlera de type ±complet+ pour désigner une classe générique dont l'ensemble des paramètres sont spécifiés.

Template: classe

Lors de l'utilisation d'une classe générique, nous devons indiquer explicitement le type à utiliser pour les paramètres:

```
Entier a; // Erreur!
Entier<int> b; // ok
Entier<long> c; // ok
```

On parlera de type ±complet+ pour désigner une classe générique dont l'ensemble des paramètres sont spécifiés.

Il est important de retenir qu'il n'existe aucune relation de typage entre deux types complets correspondant à une même classe générique.

Template: valeur par défaut

On peut spécifier des valeurs par défaut pour les paramètres d'une classe générique (pas des fonctions). Celles-ci doivent être fournit de la droite vers la gauche:

```
template<class T=int>
class Entier{...

template<class D, class H=int>
class HashTable{...

template<class T=int,class C>  // Erreur!
int compare(T *, T *)

Entier e; // ok :)
```

Template: et type primitif

Un code peut aussi être paramétré par des valeurs (constante) de types primitifs:

```
template<int taille>
class Matrix{
  int _data[taille][taille];
  public:
};
```

Template: et type primitif

Un code peut aussi être paramétré par des valeurs (constante) de types primitifs:

```
template<int taille>
class Matrix{
  int _data[taille][taille];
  public:
};
```

Dans ce cas, `Matrix<4>` et `Matrix<3>` sont deux types différents. Cet exemple pourrait être utilisé en infographie.

Template: spécialisation

Il est possible de modifier l'implémentation pour certaines valeurs de paramètre:

```
template<class T>
void affiche(const T &t){
  std::cout<<t<<std::endl;
}
```

Dans le cas des entiers, on utilise printf:

```
template<>
void affiche<int>(const int &t){
  printf("%d\n",t);
}
```

Template: spécialisation

Dans le cas où les paramètres sont des types primifs, on peut spécialiser selon les valeurs:

```
template<int n>
void affiche(){
  printf("%d\n",n);
}

template<>
void affiche<0>(){
  printf("zero\n");
}

...
  affiche(); // erreur !
  affiche<1>(); // affiche: 1
  affiche<0>(); // affiche: zero
```

Template: spécialisation

Dans le cas des classes, la spécialisation peut aussi se faire sur les pointeurs:

```
template<class T>
class Tableau{
  T *_data;
  ...
};

template<class T>
class Tableau<T *>{
  T **_data; // Attention ici !
  public:
  Tableau(int i){
    ...
    _data[i]=NULL;
    ...
  }
};
```

Template: spécialisation partielle

Il est possible de ne spécialiser que certains de paramètre d'un code générique:

```
template<class T,class H>
class HashTable{
};

template<class T,class H>
class HashTable<T *,H>{
};

template<class T>
class HashTable<T *,int>{
};
```

Important: il n'y a aucune contrainte entre ces différentes implémentations car elle ne correspondent pas au même type

Template: contraintes

Lorsque l'on utilise un type dans un code générique, il faut faire très attention aux propriétés que l'on impose sur ce type:

Template: contraintes

Lorsque l'on utilise un type dans un code générique, il faut faire très attention aux propriétés que l'on impose sur ce type:

```
template<class T>
void swap(T &a,T&b){
  T t=a;
  a=b;
  b=t;
}
```

```
template<class T>
void swap(T &a,T&b){
  T t;
  t=a;
  a=b;
  b=t;
}
```

Template: contraintes

Lorsque l'on utilise un type dans un code générique, il faut faire très attention aux propriétés que l'on impose sur ce type:

```
template<class T>
void swap(T &a,T&b){
  T t=a;
  a=b;
  b=t;
}
```

```
template<class T>
void swap(T &a,T&b){
  T t;
  t=a;
  a=b;
  b=t;
}
```

On veillera à toujours indiquer clairement les méthodes attendues pour une type paramètre.

Template: STL et BOOST

Il existe deux grandes bibliothèques utilisant les templates, la Standard Template Library et BOOST.
La STL contient:

- des conteneurs (tableaux, map, ...),
- des iterateurs (parcours),
- des algorithmes (comparaison, recopie, recherche, tris...),
- des allocateurs mémoire,
- des foncteurs

La bibliothèque BOOST contient beaucoup, beaucoup de choses (conteneurs, graphes, maths, entrées/sorties, fichiers...).

Template: STL

Pour comprendre et utiliser la STL, il faut savoir qu'il est possible de déclarer un alias de type dans une classe:

```
template<class T>
class Entier{
  public:
  typedef T type;
};

Entier<int>::type i;
```

Ceci fait que la variable `i` est de type **int**

Template: STL

Parmis les conteneurs les plus connus, il y a les vecteurs, ceux-ci répondent au concept de “Conteneur à Accès Quelconque” et “Sequence avec insertion en fin”. Ces concepts définissent un certain nombre des propriétés: compléxité, nom de méthodes, type interne...

```
#include <vector>

std::vector<int> v;
v.push_back(0);
v.push_back(1);
v[0]=2;
for(std::vector<int>::iterator i=v.begin();
          i!=v.end();
          i++)
   std::cout<<"␣"<<*i;
```

Cet exemple utilise le type interne `iterator` qui correspond au type implémentant le concept d’itérateur pour la classe vecteur.

Template: STL

Les itérateurs ont été conçus pour fonctionner comme les pointeurs: on utilise ++ et −− pour se déplacer, ⋆ et −> pour utiliser la valeur représenté par l'itérateur.

Template: STL

Les itérateurs ont été conçus pour fonctionner comme les pointeurs: on utilise `++` et `--` pour se déplacer, `*` et `->` pour utiliser la valeur représenté par l'itérateur.
Voici un exemple permettant de tier un vecteur:

```
#include <vector>

vector<int> v;
v.push_back(0); v.push_back(5);  v.push_back(2);
std::sort(v.begin(),v.end());
```

L'algorithme `std::sort` effectue un trie des données contenues par la séquence des deux itérateurs.
Ainsi, cet algorithme pourra être utiliser avec de nombreux autre conteneur de la STL qui disposent des iterateurs et même des tableaux:

```
int t[3] = {0, 5, 2};
std::sort(t,t+3);
```

Template: et metaprogrammation

La metaprogrammation consiste à faire s'executer un programme par le compilateur au moment de la compilation et donc de façon statique.

Template: et metaprogrammation

La metaprogrammation consiste à faire s'executer un programme par le compilateur au moment de la compilation et donc de façon statique.

Un exemple classique de metaprogrammation est un calcul mathématique simple:

```
template<int a, int b>
struct Addition{
  static const int resultat=a+b;
};

std::cout<< Addition<2,3>::resultat <<std::endl;
```

Template: et metaprogrammation

et un peu plus compliqué:

```
template<int a, int b>
struct Puissance{
  static const int resultat=a*Puissance<a,b-1>::resultat;
};

template<int a>
struct Puissance<a,0>{
  static const int resultat=1;
};

std::cout<< Puissance<2,3>::resultat <<std::endl;
```

Template: et metaprogrammation

La metaprogrammation permet de faire des choses très puissance tel que la composition de types. Cependant, elle reste souvent réservée à l'implémentation de bibliothèques complexe et est difficile à maitriser.
Elle repose sur les concepts suivant:

- les templates,
- la spécialisation,
- les types internes,
- les attributs statiques et constants

Templates: et metaprogrammation

On peut ainsi écrire un “language” sur les types, avec des branchements conditionnels:

```
template<bool C, class T, class E>
struct IfThenElse{
  typedef T result;
};

template<class T,class E>
struct IfThenElse<false,T,E>{
  typedef E result;
};
```

Template: et metaprogrammation

Comparaison de type:

```
template<class T,class U>
struct Equal{
   static bool result=false;
};
template<class T>
struct Equal<T,T>{
   static bool result=true;
}
```