

# LANGUAGE OBJ. AV.( C++ ) MASTER 1

Yan Jurski

U.F.R. d'Informatique  
Université de Paris Cité

VU À LA DERNIÈRE SÉANCE

# DÉFINITION CANONIQUE D'UNE CLASSE

pour les copies

pour les tableaux

pour les destructions

```
class ClasseCanon {  
public:  
    ClasseCanon();  
    ClasseCanon(const ClasseCanon &);  
    virtual ~ClasseCanon();  
    ClasseCanon &operator=(const ClasseCanon &);  
    friend ostream &  
        operator<<(ostream &, const ClasseCanon &);  
};
```

pour les affichages

pour les affectations

Illustration (rappels) :  
que penser des initialisations suivantes ?  
(s'il n'y avait pas d'optimisation)

```
class X{  
    public : X(int );  
};
```

```
int main() {  
    X x1{1};  
    X x2(3);  
    X x3{x1};  
    X x4(x1);  
    X x5=x1;  
    X x6={x1};  
    X x7= X(1);  
    X x8= 1;  
}
```

Illustration (rappels) :  
que penser des initialisations suivantes ?  
(s'il n'y avait pas d'optimisation)

```
class X{  
    public : X(int );  
};
```

```
int main() {  
    X x1{1};  
    X x2(3);  
    X x3{x1};  
    X x4(x1);  
    X x5=x1;  
    X x6={x1};  
    X x7= X(1);  
    X x8= 1;  
}
```

Rappelez vous bien que :

- il y a des nuances ...
- on peut compliquer les choses en définissant `X(const initializer_list<int> &)`
- les optimisations à la compilation peuvent perturber (et toutes ne sont pas optionnelles)

# Aujourd'hui : la relation d'héritage

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- visibilité de l'héritage
- liaison dynamique ou pas
- cas des destructeurs virtuels
- cas des copies/affectation
- petite chose sur final

## Plan :

- **généralités (spécialisation/généralisation)**
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- visibilité de l'héritage
- liaison dynamique ou pas
- cas des destructeurs virtuels
- cas des copies/affectation
- petite chose sur final

La **classe** :

- est une unité d'encapsulation
- elle définit un type de données
- elle est une description concrète, une implémentation, un modèle de construction etc

On peut penser à :

**l'étendre** : c. à d. lui ajouter des choses (données ou méthodes) ou simplement modifier son comportement.

La **factoriser** : c.à d. effectuer une réorganisation, une refonte du code, après avoir observé des points communs entre plusieurs classes



**L'héritage** (vu dans le sens d'une spécialisation/extension) consiste à personnaliser une classe existante de sorte qu'elle se conduise de façon particulière en :

- lui **ajoutant** des fonctionnalités
- **modifiant** le comportement des actions déjà existantes

L'héritage ne modifie pas la classe de base :

on construit une nouvelle **sous-classe**

## Plan :

- généralités (spécialisation/généralisation)
- **syntaxe, exemple : construction / destruction**
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- visibilité de l'héritage
- liaison dynamique ou pas
- cas des destructeurs virtuels
- cas des copies/affectation
- petite chose sur final

# Example :

```
class Individu {  
    public:  
        string getPrenom() const;  
};
```

```
#include "Individu.hpp"  
string Individu::getPrenom() const {  
    return "toto";  
}
```

# Déclaration / Définition d'une classe fille

```
#include "Individu.hpp"
class Sportif : public Individu {
    public:
        string getSportPratique() const;
};
```

```
#include "Sportif.hpp"
string Sportif::getSportPratique() const {
    return "Tennis";
}
```

# Déclaration / Définition d'une classe fille

Notez ici le  
qualificatif  
public

```
#include "Individu.hpp"
class Sportif : public Individu {
    public:
        string getSportPratique() const;
};
```

```
#include "Sportif.hpp"
string Sportif::getSportPratique() const {
    return "Tennis";
}
```

# Utilisation possible :

```
#include "Sportif.hpp"
int main() {
    Sportif s;
    cout << s.getPrenom() << " pratique le " <<
        s.getSportPratique();
    return EXIT_SUCCESS;
}
```

```
toto pratique le Tennis
```

Les méthodes de la classe de déclaration ainsi que celles de la classe mère sont disponibles

# On peut également modifier des comportements (redéfinir une fonction membre)

```
#include "Individu.hpp"
class Sportif : public Individu {

    public:
        string getSportPratique() const;
        string getPrenom() const;
};
```

```
#include "Sportif.hpp"
string Sportif::getSportPratique() const {
    return "Tennis";
}
string Sportif::getPrenom() const {
    return "** Raphael **";
}
```

redéfinition

# On peut également modifier des comportements (redéfinir une fonction membre)

```
#include "Individu.hpp"
class Sportif : public Individu {

    public:
        string getSportPratique() const;
        string getPrenom() const;
};
```

```
** Raphael ** pratique le tennis
```

```
#include "Sportif.hpp"
string Sportif::getSportPratique() const {
    return "Tennis";
}
string Sportif::getPrenom() const {
    return "** Raphael **";
}
```

redéfinition



# Redéfinir une fonction membre, en réutilisant la fonction mère :

```
#include "Individu.hpp"
class Sportif : public Individu {

    public:
        string getSportPratique() const;
        string getPrenom() const;
};
```

```
#include "Sportif.hpp"
string Sportif::getSportPratique() const {
    return "Tennis";
}
string Sportif::getPrenom() const {
    return "**"+Individu::getPrenom()+"**";
}
```

méthode mère

# Redéfinir une fonction membre, en réutilisant la fonction mère :

```
#include "Individu.hpp"
class Sportif : public Individu {
    public:
        string getSportPratique() const;
        string getPrenom() const;
};
```

```
** Toto ** pratique le tennis
```

```
#include "Sportif.hpp"
string Sportif::getSportPratique() const {
    return "Tennis";
}
string Sportif::getPrenom() const {
    return "**"+Individu::getPrenom()+"**";
}
```

méthode mère

# Etude de la construction

```
class Individu {  
    private:  
        const string prenom;  
    public:  
        Individu(const string );  
        string getPrenom() const;  
};
```

```
#include "Individu.hpp"  
Individu::Individu(const string p): prenom{p} {}  
string Individu::getPrenom() const {  
    return prenom;  
}
```

# Etude de la construction

```
class Sportif : public Individu {  
    private:  
        const string sport;  
    public:  
        Sportif(const string p, const string s);  
        string getSportPratique() const;  
};
```

Syntaxe *similaire* à  
l'initialisation des membres

```
#include "Sportif.hpp"  
Sportif::Sportif(const string p, const string s)  
    : Individu{p}, sport{s} {}  
string Sportif::getSportPratique() const {  
    return sport;  
}
```

# Etude de la construction

```
class Sportif : public Individu {  
    private:  
        const string sport;  
    public:  
        Sportif(const string p, const string s);  
        string getSportPratique() const;  
};
```

Syntaxe *similaire* à  
l'initialisation des membres

```
#include "Sportif.hpp"  
Sportif::Sportif(const string p, const string s)  
    : Individu{p}, sport{s} {}  
string Sportif::getSportPratique() const {  
    return sport;  
}
```

Après l'allocation mémoire, on initialise d'abord  
l'objet de base, avant ses attributs

Les appels aux destructeurs s'opèrent dans l'ordre inverse des constructions :

- d'abord celui de la sous-classe
- puis celui la super-classe,
- la désallocation intervient à la fin

# Destructions

```
class A {  
    public:  
        A() { cout << "A() "; }  
        ~A() { cout << "~A() "; }  
};  
class B : public A {  
    public:  
        B() { cout << "B() "; }  
        ~B() { cout << "~B() "; }  
};  
  
int main() {  
    B b;  
  
}
```

construction  
implicite  
B():A{}

```
A()  
B()  
~B()  
~A()
```

# Destructions

```
class A {
public:
    A() { cout << "A() "; }
    ~A() { cout << "~A() "; }
};
class B : public A {
public:
    B() { cout << "B() "; }
    ~B() { cout << "~B() "; }
};
class C : public B {
public:
    C() { cout << "C() "; }
    ~C() { cout << "~C() "; }
};
int main() {
    B b;
    C c;
}
```

construction  
implicite  
B():A{}

```
A ()
B ()
A ()
B ()
C ()
~C ()
~B ()
~A ()
~B ()
~A ()
```



## Plan :

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- **visibilité des membres/méthodes, amitié**
- polymorphisme, casting
- visibilité de l'héritage
- liaison dynamique ou pas
- cas des destructeurs virtuels
- cas des copies/affectation
- petite chose sur final

# Retour sur la protection (le contrôle d'accès)

# Retour sur la protection (le contrôle d'accès)

`private` : visible **uniquement** depuis les fonctions membres de **la classe**

`public` : visible depuis n'importe quelle partie du code

Être dans une fonction membre d'une sous-classe ne signifie pas être dans la classe de base !

```
class A {  
    private:  
        int a;  
        void f();  
};  
class B : public A {  
    private:  
        int b;  
        void g();  
};
```

```
void B::g() {  
    a = 10; // privé pour la sous-classe  
    f();    // privé pour la sous-classe  
    b = 10; // privé, mais de la classe (et de l'objet courant)  
    B aux;  
    aux.b=10; // privé, mais d'un autre objet de la même classe  
    aux.a = 10; // privé pour la sous-classe  
}
```

La protection telle qu'elle a été vue jusqu'ici peut-être considérée comme trop restrictive

On peut utiliser un troisième domaine :

un membre d'un objet qui aurait été déclaré `protected` dans une classe mère (d'héritage public) est :

- accessible lorsqu'il est considéré de l'intérieur (*i.e.* dans `this` ou d'un objet de sa classe)
- inaccessible lorsqu'il est considéré de l'extérieur

```
class A {
    protected :
        int a;
        void f();
};
class B : public A {
    private:
        int b;
        void g();
};
```

```
void B::g() {
    a = 10; // ok avec protected
    f();    // ok avec protected
    b = 10; // privé, mais de la classe (et de l'objet courant), ok
    B aux;
    aux.b=10; // privé, mais d'un autre objet de la même classe, ok
    aux.a = 10; // ok avec protected
}
```

# limite :

```
class A {  
    protected :  
        int a;  
        void f();  
};  
class B : public A {  
    private:  
        int b;  
        void g();  
};
```

```
void B::g() {  
    a = 10; // ok avec protected  
    f();    // ok avec protected  
    b = 10; // privé, mais de la classe (et de l'objet courant), ok  
    B aux;  
    aux.b=10; // privé, d'un autre objet de la même classe, ok  
    aux.a = 10; // ok avec protected  
  
    A tmp;  
    tmp.a; // error: 'int A::a' is protected within this context  
}
```

"L'intérieur" est donc toute la chaîne ascendante de l'objet courant, ou d'un objet du même niveau

# Exercise 1/2

```
class A {  
    private :  
        int a;  
        void f();  
};  
class B : public A {  
    protected :  
        int b;  
};
```

```
void A::f() {  
    B aux;  
    aux.a;    // ??  
    aux.f();  // ??  
    aux.b;    // ??  
}
```



# Exercice 1/2

```
class A {  
    private :  
        int a;  
        void f();  
};  
class B : public A {  
    protected :  
        int b;  
};
```

```
void A::f() {  
    B aux;  
    aux.a;    // oui on est dans A et dans la partie de la classe A de B  
    aux.f();  // oui on est dans A et dans la partie de la classe A de B  
    aux.b;    // ??  
}
```

# Exercice 1/2

```
class A {  
    private :  
        int a;  
        void f();  
};  
class B : public A {  
    protected :  
        int b;  
};
```

```
void A::f() {  
    B aux;  
    aux.a;    // oui on est dans A et dans la partie de la classe A de B  
    aux.f();  // oui on est dans A et dans la partie de la classe A de B  
    aux.b;    // non, le caractère protected est destiné aux sous classes  
              // ici, dans A, on est à l'extérieur de B  
}
```

# Exercice 2/2

```
class A {  
    protected :  
    int a;  
};  
  
class B : public A {  
    protected :  
    int b;  
    void f();  
};  
  
class C : public B {  
    protected :  
    int c;  
};
```

```
void B::f() {  
    A tmp;  
    tmp.a; // ??  
  
    B bis;  
    bis.a;  
    this->a;  
  
    C aux;  
    aux.a; // ??  
    aux.b; // ??  
    aux.c; // ??  
}
```

OK : on est dans la chaîne de this, ou on manipule un objet de la même classe

# Exercice 2/2

```
class A {  
    protected :  
        int a;  
};  
  
class B : public A {  
    protected :  
        int b;  
        void f();  
};  
  
class C : public B {  
    protected :  
        int c;  
};
```

```
void B::f() {  
    A tmp;  
    tmp.a; // ??  
  
    B bis;  
    bis.a;  
    this->a;  
  
    C aux;  
    aux.a; // OUI  
    aux.b; // OUI  
    aux.c; // NON  
}
```

C est aussi un B : on manipule  
un objet de la même classe

# Exercice 2/2

```
class A {  
    protected :  
    int a;  
};  
  
class B : public A {  
    protected :  
    int b;  
    void f();  
};  
  
class C : public B {  
    protected :  
    int c;  
};
```

```
void B::f() {  
    A tmp;  
    tmp.a; // NON  
  
    B bis;  
    bis.a;  
    this->a;  
  
    C aux;  
    aux.a; // OUI  
    aux.b; // OUI  
    aux.c; // NON  
}
```

cet accès n'est pas dans la chaîne de this, ni ne manipule un objet de la même classe

Les protections peuvent être considérées comme n'étant pas assez expressives, et on peut vouloir faire exceptions aux règles rigides `public/private/protected`.

C++ introduit le qualificatif `friend` : qui s'applique pour une classe entière, ou pour une méthode.

Il permet de donner à la classe ou à la méthode considérée des droits d'accès équivalents à ceux de la classe courante.

# Exemple d'amitiés :

```
class A {  
    private:  
        int aPrive;  
    protected:  
        int aProtege;  
    public:  
        int aPublic;  
};  
class B : public A {  
  
    private:  
        int bPrive;  
    protected:  
        int bProtege;  
    public:  
        int bPublic;  
    friend void f(); // une fonction externe à ces classes  
};
```

```
void f() {  
    B s;  
    s.aPrive;  
    s.aProtege;  
    s.aPublic;  
    s.bPrive;  
    s.bProtege;  
    s.bPublic;  
}
```

## Exemple 2 :

```
class A {
    private :
        int a;
        void f();
};
class B : public A {
    protected :
        int b;
        friend void A::f();    // on autorise f a tout voir de B
};
```

```
void A::f() {
    B aux;
    aux.a;    // oui on est dans A et dans la partie de la classe A de B
    aux.f();  // oui on est dans A et dans la partie de la classe A de B
    aux.b;    // ok à présent, grace à l'amitié pour f
}
```



## Exemple 2 :

```
class A {  
    private :  
        int a;  
        void f();  
};  
class B : public A {  
    protected :  
        int b;  
        friend class A;    // idem pour toute la classe A  
};
```

```
void A::f() {  
    B aux;  
    aux.a;    // oui on est dans A et dans la partie de la classe A de B  
    aux.f();  // oui on est dans A et dans la partie de la classe A de B  
    aux.b;    // ok à présent, grace à l'amitié de toute la classe  
}
```

## Plan :

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- **polymorphisme, casting**
- visibilité de l'héritage
- liaison dynamique ou pas
- cas des destructeurs virtuels
- cas des copies/affectation
- petite chose sur final

# LA COMPATIBILITÉ DES TYPES

D'une manière générale :

- "on peut toujours voir une instance d'une classe donnée comme un objet de sa super-classe"
- "on peut toujours convertir un pointeur vers une instance d'une classe donnée en un pointeur vers un objet de sa super-classe"

Voyons cela ensemble

*"on peut toujours voir une instance d'une classe donnée en un objet de sa super-classe"*

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

```
void B::f() {}  
  
int main() {  
    A a;  
    B b;  
    A & a2 {b};  
    ((B&)a2).f();  
}
```

pour illustrer cela on utilise une référence.

Sans elle on ferait une copie :

- à la définition de a2
- au moment du cast en B

*"on peut toujours convertir un pointeur vers une instance d'une classe donnée en un pointeur vers un objet de sa super-classe"*

```
class A {};  
class B : public A {};  
  
int main() {  
    A a;  
    B b;  
  
    A* pa{&a};  
    B* pb{&b};  
    A* pb2{&b};  
  
    B* pa2{&a}; // interdit heureusement  
    B* pa3{(B*)pb2}; // on veut assumer (et c'est correct)  
}
```

*"on peut toujours convertir un pointeur vers une instance d'une classe donnée en un pointeur vers un objet de sa super-classe"*

```
class A {};  
class B : public A {};  
  
int main() {  
    A a;  
    B b;  
  
    A* pa{&a};  
    B* pb{&b};  
    A* pb2{&b};  
  
    B* pa2{&a}; // interdit heureusement  
    B* pa3{(B*)pb2}; // on veut assumer (et c'est correct)  
}
```

Ici la conversion est faite de manière statique,  
le programmeur doit être sûr de son succès

*"on peut toujours convertir un pointeur vers une instance d'une classe donnée en un pointeur vers un objet de sa super-classe"*

```
class A {};  
class B : public A {};  
  
int main() {  
    A a;  
    B b;  
  
    A* pa{&a};  
    B* pb{&b};  
    A* pb2{&b};  
  
    B* pa2{dynamic_cast<B*>(pa)}; // donne nullptr  
    B* pa3{dynamic_cast<B*>(pb2)}; // donne pb2  
}
```

On peut vouloir faire la conversion de manière "dynamique". C'est utile si des calculs ont été fait sur pb2, et qu'on ne sait plus trop quelle est la nature de \*pb2.  
(Comparable à instanceof de java)

## Plan :

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- **visibilité de l'héritage**
- liaison dynamique ou pas
- cas des destructeurs virtuels
- cas des copies/affectation
- petite chose sur final



# héritages non public

dans nos exemples précédents :

```
class A {  
    public :  
        int pub;  
    protected :  
        int prot;  
    private :  
        int priv;  
};  
class B : public A {  
    void f();  
};
```

```
void B::f() {  
    pub;  
    prot;  
    priv; // non accessible  
}
```

# héritages non public

dans nos exemples précédents :

```
class A {  
    public :  
        int pub;  
    protected :  
        int prot;  
    private :  
        int priv;  
};  
class B : public A {  
    void f();  
};
```

```
void B::f() {  
    pub;  
    prot;  
    priv; // non accessible  
}
```

Le caractère public n'est pas systématique/obligatoire

# héritages non public

```
class A {  
    public :  
        int pub;  
    protected :  
        int prot;  
    private :  
        int priv;  
};  
class B : private A {  
    void f();  
};
```

```
void B::f() {  
    pub;    // ??  
    prot;   // ??  
    priv;   // ??  
}
```

protected ou private sont possibles à ce niveau là.

Ils **restreignent** une visibilité. Laquelle, comment ... ?

Personne d'autre (en dehors de B) ne peut savoir que B hérite de A

# héritages non public

```
class A {  
    public :  
        int pub;  
    protected :  
        int prot;  
    private :  
        int priv;  
};  
class B : private A {  
    void f();  
};
```

```
void B::f() {  
    pub;    // ok  
    prot;   // ok  
    priv;   // non mais normal  
}
```

Rien ne change quand on se situe simplement au niveau de la classe.  
C'est du point de vue de l'extérieur que qq chose va être bloqué ...

# héritages non public

```
class A {  
};  
class B : private A {  
};
```

```
int main() {  
    A* p;  
    p = new B{}; // refusé  
    return EXIT_SUCCESS;  
}
```

Le fait que B soit aussi un A est rendu invisible du point de vue de l'extérieur.

# héritages non public

```
class A {  
    public :  
        int pub;  
    protected :  
        int prot;  
    private :  
        int priv;  
};  
class B : private A {}  
class C : public B {  
    void f();  
};
```

```
void C::f() {  
    pub;    // non  
    prot;   // non  
    priv;   // non  
}
```

De même : les droits pour les champs et méthodes héritées ont été tronqués :  
au delà de B ils sont considérés privés.  
C ne sait pas que B est un A

# héritages non public

```
class A {};  
class B : protected A {};  
class C : public B {  
    void f();  
}
```

```
void C::f() {  
    A* p;  
    p = new B{}; // ok !  
}  
int main() {  
    A* p;  
    p = new B{}; // non  
    return EXIT_SUCCESS;  
}
```

avec protected : le fait d'être un A est une information transmise aux enfants, mais invisible de l'extérieur. C'est à dire que :

- les C savent que les B sont des A.
- du code "extérieur" ne peut pas lier un B à un A

# héritages non public

Illustration des accès internes avec protected

```
class A {  
    public :  
        int pub;  
    protected :  
        int prot;  
    private :  
        int priv;  
};  
class B : protected A {}  
class C : public B {  
    void f();  
};
```

```
void C::f() {  
    pub;    // oui  
    prot;   // oui  
    priv;   // non  
}
```

```
int main() {  
    B b;  
    b.pub;    // non  
    b.prot;   // non  
    b.priv;   // non  
}
```



# héritages non public

Par défaut, c'est private :

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

```
void C::f() {  
    A* p= new A{};  
    p = new B{}; // non  
}  
int main() {  
    A* p= new A{};  
    p = new B{}; // et non  
    return EXIT_SUCCESS;  
}
```

On vérifie : le fait que B soit aussi un A est "invisible" du point de vue de l'extérieur.

# héritages non public

Utilité ?

Exemple :

Ce sont B ou C qui dans leur code décideront quand invoquer les méthodes de A. Personne d'autre ne pourra les déclencher directement.

```
class A {  
    public : void dormir() {};  
};  
class B : protected A {  
    public : void compterMoutons() { dormir(); }  
};  
class C : protected B {  
    public : void suivreDeuxHeuresDeCours() { dormir(); }  
};
```

```
int main() {  
    new A{} -> dormir();  
    new B{} -> compterMoutons();  
    new C{} -> suivreDeuxHeuresDeCours();  
    ... // penser à les détruire aussi  
}
```

rien d'autre n'est  
faisable sur les objets  
créés

# héritages non public

Bizarrie ...

L'héritage privé doit rendre invisible que B est un A.  
En particulier sa conversion devrait être une erreur

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

```
int main() {  
    B b;  
    A* p;  
    // p = &b; // refusé  
    // A & a{b}; // refusé  
    // A & a{ dynamic_cast<A&> (b) };  
    // p= dynamic_cast<A*> (&b);  
    p=(A*) (&b); // autorisé ...  
    p->f(); // et ensuite ...  
    return EXIT_SUCCESS;  
}
```

pourtant le dernier cast passe ...

je n'ai pas de justification satisfaisante

## Plan :

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- visibilité de l'héritage
- **liaison dynamique ou pas**
- cas des destructeurs virtuels
- cas des copies/affectation
- petite chose sur final

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

```
void A::f() {cout << "dans A";}  
  
void B::f() {cout << "dans B";}  
  
int main() {  
    A a;  
    B b;  
    A & a2 {b};  
  
    a.f();  
    b.f();  
    a2.f();  
    ( (B&) a2 ).f();  
}
```

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

```
void A::f() {cout << "dans A";}  
  
void B::f() {cout << "dans B";}  
  
int main() {  
    A a;  
    B b;  
    A & a2 {b};  
  
    a.f();  
    b.f();  
    a2.f();  
    ((B&) a2).f();  
}
```

```
dans A  
dans B  
dans A  
dans B
```

on constate qu'ici c'est le type déclaré qui induit le choix de la méthode ... pas très "dynamique"

La méthode a été déterminée  
statiquement (à la compilation)  
on parle d' *early binding*

Une liaison tardive (*late binding*) est  
possible. Il faut une faire une distinction  
syntaxique pour choisir entre l'une ou  
l'autre.

On utilise le mot clé **virtual**

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

```
void A::f() {cout << "dans A";}  
  
void B::f() {cout << "dans B";}  
  
int main() {  
    A a;  
    B b;  
    A & a2 {b};  
  
    a.f();  
    b.f();  
    a2.f();  
    ((B&)a2).f();  
}
```

```
dans A  
dans B  
dans B  
dans B
```



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

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

```
void A::f() {cout << "dans A";}

void B::f() {cout << "dans B";}

int main() {
    A a;
    B b;
    A & a2 {b};

    a.f();
    b.f();
    a2.f();
    ((B&) a2).f();
}
```

```
dans A
dans B
dans B
dans B
```

Rq : on n'a pas répété le mot clé virtual devant la déclaration de f() dans B.  
On aura donc quelques interrogations.  
Un cas maladroit (?) est donné au transparent suivant

```

class A {
public :
    void f();
};

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

class C : public B {
public :
    void f();
};

```

```

void A::f() {cout << "dans A";}
void B::f() {cout << "dans B";}
void C::f() {cout << "dans C";}

int main() {
    A a;
    B b;
    C c;
    A & ab {b};
    A & ac {c};
    B & bc {c};

    a.f();
    b.f();
    c.f();
    ab.f();
    ac.f();
    bc.f();

    ((B&) ac).f();
}

```

```

dans A
dans B
dans C
dans A
dans A
dans C

dans C

```

virtual étant déclaré dans la classe intermédiaire B, les liaisons sont parfois early, parfois late ...

Pour info : ensuite (pour C et au delà) le caractère virtuel sera transmis implicitement.

## Plan :

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- visibilité de l'héritage
- liaison dynamique ou pas
- **cas des destructeurs virtuels**
- cas des copies/affectation
- petite chose sur final

# Cas des destructeurs / virtual

```
class A {
public:
    A() { cout << "A() "; }
    ~A() { cout << "~A() "; }
};
class B : public A {
public:
    B() { cout << "B() "; }
    ~B() { cout << "~B() "; }
};
class C : public B {
public:
    C() { cout << "C() "; }
    ~C() { cout << "~C() "; }
};
```

```
int main() {
    C x

    return EXIT_SUCCESS;
}
```

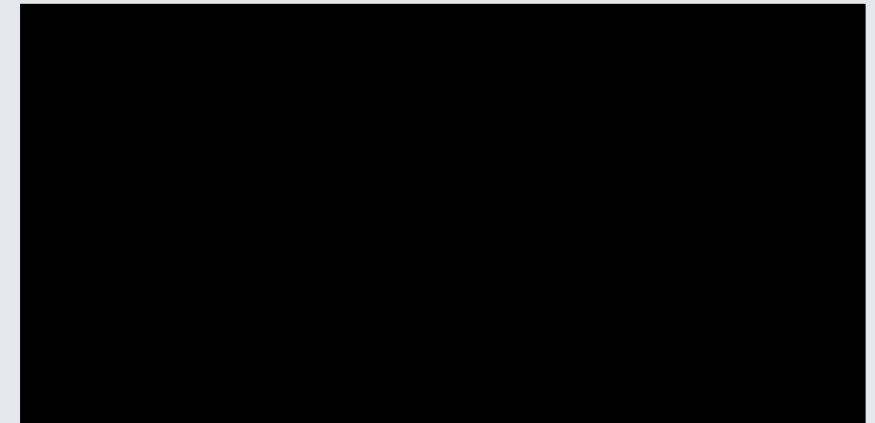
```
A()
B()
C()
~C()
~B()
~A()
```

On avait vu, dans le cas des objets déclarés avec des variables que les destructions s'enchaînaient en cascade

# Cas des destructeurs / virtual

```
class A {  
    public:  
    A() { cout << "A() "; }  
    ~A() { cout << "~A() "; }  
};  
class B : public A {  
    public:  
    B() { cout << "B() "; }  
    ~B() { cout << "~B() "; }  
};  
class C : public B {  
    public:  
    C() { cout << "C() "; }  
    ~C() { cout << "~C() "; }  
};
```

```
int main() {  
    A *p = new C;  
    delete p;  
    return EXIT_SUCCESS;  
}
```



# Cas des destructeurs / virtual

```
class A {
public:
    A() { cout << "A() "; }
    ~A() { cout << "~A() "; }
};
class B : public A {
public:
    B() { cout << "B() "; }
    ~B() { cout << "~B() "; }
};
class C : public B {
public:
    C() { cout << "C() "; }
    ~C() { cout << "~C() "; }
};
```

```
int main() {
    A *p = new C;
    delete p;
    return EXIT_SUCCESS;
}
```

```
A()
B()
C()
~A()
```

On voudrait que le destructeur appelé soit "au plus près" de l'objet réel, pas celui déclaré : c'est exactement le comportement lié à **virtual**

# Cas des destructeurs / virtual

```
class A {  
    public:  
    A() { cout << "A() "; }  
    virtual ~A() { cout << "~A() "; }  
};  
class B : public A {  
    public:  
    B() { cout << "B() "; }  
    ~B() { cout << "~B() "; }  
};  
class C : public B {  
    public:  
    C() { cout << "C() "; }  
    ~C() { cout << "~C() "; }  
};
```

```
int main() {  
    A *p = new C;  
    delete p;  
    return EXIT_SUCCESS;  
}
```

```
A()  
B()  
C()  
~C()  
~B()  
~A()
```

Le destructeur de A étant virtual (avec un caractère transmis aux sous classes), on descendra bien jusqu'à celui de C. Il fera son travail propre, puis détruira ses parents en remontant la hiérarchie de classe

# Cas des destructeurs / virtual

```
class A {
public:
    A() { cout << "A() "; }
    virtual ~A() { cout << "~A() "; }
};
class B : public A {
public:
    B() { cout << "B() "; }
    ~B() { cout << "~B() "; }
};
class C : public B {
public:
    C() { cout << "C() "; }
    ~C() { cout << "~C() "; }
};
```

```
int main() {
    B *p = new C;
    delete p;
    return EXIT_SUCCESS;
}
```

```
A()
B()
C()
~C()
~B()
~A()
```

Remarquez qu'une fois que le destructeur est déclaré virtual, ce caractère est bien propagé aux sous classes.

Pour éviter toute ambiguïté un conseil : faites apparaître les virtual implicitement hérités, pour les destructeurs comme pour les méthodes.



# Cas des destructeurs / virtual

```
class A {
public:
    A() { cout << "A() "; }
    virtual ~A() { cout << "~A() "; }
};
class B : public A {
public:
    B() { cout << "B() "; }
    virtual ~B() { cout << "~B() "; }
};
class C : public B {
public:
    C() { cout << "C() "; }
    virtual ~C() { cout << "~C() "; }
};
```

```
int main() {
    B *p = new C;
    delete p;
    return EXIT_SUCCESS;
}
```

```
A()
B()
C()
~C()
~B()
~A()
```

Remarquez qu'une fois que le destructeur est déclaré virtual, ce caractère est bien propagé aux sous classes.

Pour éviter toute ambiguïté un conseil : faites apparaître les virtual implicitement hérités, pour les destructeurs comme pour les méthodes.

Dans 99 % des cas les destructeurs sont virtual.  
Il reste 1% ... à illustrer.

Un petit mot sur « final » :

On peut déclarer l'impossibilité de définir une sous classe

```
class X final {};  
class Y : public X {}; // bloqué
```

Dans un cas de figure comme celui ci on ne voit pas à quoi pourrait servir virtual (qui est destiné aux sous classes)

Dans 99 % des cas les destructeurs sont virtual.  
Il reste 1% ... à illustrer :

```
class A {
public:
    A() { }
    ~A() { }
};
class B {
public:
    B() { }
    virtual ~B() { }
};
class C final {
public:
    C() { }
    virtual ~C() { }
};
class D final {
public:
    D() { }
    ~D() { }
};
```

```
int main() {
    cout << sizeof(A);
    cout << sizeof(B);
    cout << sizeof(C);
    cout << sizeof(D);
    return EXIT_SUCCESS;
}
```

Dans 99 % des cas les destructeurs sont virtual.  
Il reste 1% ... à illustrer :

```
class A {
public:
    A() { }
    ~A() { }
};
class B {
public:
    B() { }
    virtual ~B() { }
};
class C final {
public:
    C() { }
    virtual ~C() { }
};
class D final {
public:
    D() { }
    ~D() { }
};
```

```
int main() {
    cout << sizeof(A);
    cout << sizeof(B);
    cout << sizeof(C);
    cout << sizeof(D);
    return EXIT_SUCCESS;
}
```

```
1
4
4
1
```

cette déclaration est  
inutile v.s final ...  
virtual a visiblement un coût

Dans 99 % des cas les destructeurs sont virtual.  
Il reste 1% ... à illustrer :

On peut aussi se permettre d'avoir des destructeurs non virtual si toute la hiérarchie inférieure n'introduit pas de nouveaux attributs : dans ce cas on est sûr qu'il n'y a rien à détruire "plus bas".

# Questions courantes

Partons de :

```
class A {  
    public :  
        int x;  
        A() :x{1} {}  
};  
class B : public A {};
```

```
int main() {  
    B b;  
    A& a{b};  
    cout << a.x;  
    cout << b.x;  
}
```

le rappel qu'un accès à un attribut d'une classe mère est possible

1  
1

# Questions courantes :

```
class A {  
    public :  
        int x;  
        A() :x{1} {}  
};  
class B : public A {  
    public :  
        int x;  
        B():A(),x{2} {}  
};
```

```
int main() {  
    B b;  
    A& a{b};  
    cout << a.x;  
    cout << b.x;  
}
```

le rappel qu'un accès à un attribut n'est pas "virtual"  
(on ne peut pas déclarer un attribut virtual)

1  
2

# Questions courantes :

```
class A {  
    public :  
        int x;  
        A() :x{1} {}  
};  
class B : public A {  
    public :  
        int x;  
        B():A(),x{2} {}  
};
```

```
int main() {  
    B b;  
    A& a{b};  
    cout << a.x;  
    cout << b.x;  
    cout << (A &)b).x;  
    cout << b.A::x;  
    cout << b.B::x;  
}
```

le rappel qu'un accès à un attribut n'est pas "virtual"  
(on ne peut pas déclarer un attribut virtual)

1  
2  
1  
1  
2



# Questions courantes :

```
class A {  
    public :  
        int x;  
        A() :x{1} {}  
};  
class B : public A {  
    private :  
        int x;  
    public :  
        B():A(),x{2} {}  
};
```

```
int main() {  
    B b;  
    A& a{b};  
    cout << a.x;  
    cout << b.x;  
}
```

deux x coexistent dans b.  
L'un public, l'autre privé.

...

..?  
..?

# Questions courantes :

```
class A {  
    public :  
        int x;  
        A() :x{1} {}  
};  
class B : public A {  
    private :  
        int x;  
    public :  
        B():A(),x{2} {}  
};
```

```
int main() {  
    B b;  
    A& a{b};  
    cout << a.x;  
    cout << b.x;  
}
```

deux x coexistent dans b.  
L'un public, l'autre privé.

...

1  
erreur

# Cas limite :

```
class A {  
    public :  
        virtual void f() {cout << "de A";};  
};  
class B : public A {  
    private :  
        void f() { cout << "de B"; }  
};
```

```
int main() {  
    B b;  
    A & a{b};  
    a.f();  
}
```

difficile de trancher ...

?

# Cas limite :

```
class A {  
    public :  
        virtual void f() {cout << "de A";};  
};  
class B : public A {  
    private :  
        void f() { cout << "de B"; }  
};
```

```
int main() {  
    B b;  
    A & a{b};  
    a.f();  
}
```

difficile de trancher ...  
en tout cas ca doit  
compiler

?

# Cas limite :

```
class A {  
    public :  
        virtual void f() {cout << "de A";};  
};  
class B : public A {  
    private :  
        void f() { cout << "de B"; }  
};
```

```
int main() {  
    B b;  
    A & a{b};  
    a.f();  
}
```

difficile de trancher ...  
en tout cas ca doit compiler ...  
et donc s'exécuter si on le laisse  
passer

?

# Cas limite :

```
class A {  
    public :  
        virtual void f() {cout << "de A";};  
};  
class B : public A {  
    private :  
        void f() { cout << "de B"; }  
};
```

```
int main() {  
    B b;  
    A & a{b};  
    a.f();  
}
```

difficile de trancher ...  
en tout cas ca doit compiler ...  
et donc s'exécuter si on le laisse  
passer  
c'est le choix fait par les concepteurs

de B

# Cas limite :

```
class A {  
    public :  
        virtual void f() {cout << "de A";};  
};  
class B : public A {  
    private :  
        void f() { cout << "de B"; }  
};
```

```
int main() {  
    B b;  
    A & a{b};  
    a.f();  
    b.f();  
}
```

de B  
?

# Cas limite :

```
class A {  
    public :  
        virtual void f() {cout << "de A";};  
};  
class B : public A {  
    private :  
        void f() { cout << "de B"; }  
};
```

```
int main() {  
    B b;  
    A & a{b};  
    a.f();  
    b.f();  
}
```

de B  
erreur à la  
compilation...



## Plan :

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- visibilité de l'héritage
- liaison dynamique ou pas
- cas des destructeurs virtuels
- **cas des copies/affectation**
- petite chose sur final

On rappelle qu'en c++, lors de passages d'arguments ou de return (non référence), des copies sont faites.

Lorsqu'on fait une affectation, il y a aussi qq chose de similaire.

Qu'est-il fait par défaut lorsqu'il y a héritage ?  
Comment les redéfinir si besoin ?

```

class A {
    public:
        A() { cout << "A() "; }
        A(const A &a) { cout << "A(const A&) "; }
        const A& operator=(const A &) {
            cout << "operator=(A) "; return *this; }
};

class B : public A {
    public:
        B() { cout << "B() "; }
};

void f(B b) { }

int main()
{
    B b1, b2;
    f(b1);
    b1=b2;
}

```

```

A()
B()
A()
B()
A(const A&)
operator=(A)

```

Illustration du comportement par défaut des constructeurs de copie et d'affectation d'une classe B fille de A

```

class A {
public:
    A() { cout << "A() "; }
    A(const A &a) { cout << "A(const A&) "; }
    const A& operator=(const A &) {
        cout << "operator=(A) "; return *this; }
};

class B : public A {
public:
    B() { cout << "B() "; }
};

void f(B b) { }

int main()
{
    B b1, b2;
    f(b1);
    b1=b2;
}

```

une copie (par défaut) de B est faite. Elle fait aussi celle du A sous-jacent.

```

A()
B()
A()
B()
A(const A&)
operator=(A)

```

Illustration du comportement par défaut des constructeurs de copie et d'affectation d'une classe B fille de A

```

class A {
    public:
        A() { cout << "A() "; }
        A(const A &a) { cout << "A(const A&) "; }
        const A& operator=(const A &) {
            cout << "operator=(A) "; return *this; }
};

class B : public A {
    public:
        B() { cout << "B() "; }
};

void f(B b) { }

int main()
{
    B b1, b2;
    f(b1);
    b1=b2;
}

```

l'affectation (par défaut)  
de B fait appel à celle du  
A sous-jacent.

```

A()
B()
A()
B()
A(const A&)
operator=(A)

```

Illustration du comportement par défaut des constructeurs de copie et d'affectation d'une classe B fille de A

# Si on redéfinit l'affectation

```
class A {
public:
    A() { cout << "A() "; }
    A(const A &a) { cout << "A(const A&) "; }
    const A& operator=(const A &) {
        cout << "operator=(A) "; return *this; }
};

class B : public A {
public:
    B() { cout << "B() "; }
    const B& operator=(const B &x) {
        cout << "operator=(B) "; return *this; }
};

void f(B b) { }

int main()
{
    B b1, b2;
    b1=b2;
}
```

on perd l'appel à  
l'affectation de A sous  
jacente.  
(Indispensable s'il y a des  
champs privés dans A)

```
A()
B()
A()
B()
operator=(B)
```

# Si on redéfinit l'affectation

```
class A {
public:
    A() { cout << "A() "; }
    A(const A &a) { cout << "A(const A&) "; }
    const A& operator=(const A &) {
        cout << "operator=(A) "; return *this; }
};

class B : public A {
public:
    B() { cout << "B() "; }
    const B& operator=(const B &x) {
        A::operator=(x); // il faut l'écrire si on la veut
        cout << "operator=(B) "; return *this; }
};

void f(B b) { }

int main()
{
    B b1, b2;
    b1=b2;
}
```

```
A()
B()
A()
B()
operator=(A)
operator=(B)
```

# Si on redéfinit la copie ? On avait :

```
class A {
public:
    A() { cout << "A() "; }
    A(const A &a) { cout << "A(const A&) "; }
};

class B : public A {
public:
    B() { cout << "B() "; }
};

void f(B b) { }

int main()
{
    B b;
    f(b);
}
```

une copie (par défaut) de B est faite. Elle fait aussi celle du A sous-jacent.

```
A()
B()
A(const A&)
```



# Si on redéfinit la copie :

```
class A {
public:
    A() { cout << "A() "; }
    A(const A &a) { cout << "A(const A&) "; }
};

class B : public A {
public:
    B() { cout << "B() "; }
    B(const B &b) { cout << "B(const &B) "; }
};

void f(B b) { }

int main()
{
    B b;
    f(b1);
}
```

c'est le constructeur A() qui est choisi... Probablement qu'on aurait voulu A(const A&)

```
A()
B()
A()
B(const &B)
```

# Si on redéfinit la copie :

```
class A {
    public:
        A() { cout << "A() "; }
        A(const A &a) { cout << "A(const A&) "; }
};

class B : public A {
    public:
        B() { cout << "B() "; }
        B(const B &b) : A{b} { cout << "B(const B &) "; }
};

void f(B b) { }

int main()
{
    B b;
    f(b);
}
```

```
A()
B()
A(const A&)
B(const B&)
```

## Plan :

- généralités (spécialisation/généralisation)
- syntaxe, exemple : construction / destruction
- visibilité des membres/méthodes, amitié
- polymorphisme, casting
- visibilité de l'héritage
- liaison dynamique ou pas
- cas des destructeurs virtuels
- cas des copies/affectation
- **on finit par une petite chose sur final**

On a déjà vu qu'on utilise **final** pour déclarer l'impossibilité de dériver une sous classe :

```
class X final {};
```

```
class Y : public X {}; // bloqué
```

**final** sert aussi pour empêcher une redéfinition d'une méthode virtual (dynamiquement liée)

```
class A {  
    public:  
        virtual void f(){};  
  
};  
class B : public A {  
    public:  
        void f() final {};  
  
};  
class C : public B {  
    public:  
        void f(){}; // non  
};
```

**final** sert aussi pour empêcher une redéfinition d'une méthode virtual (dynamiquement liée)

```
class A {  
    public:  
        virtual void f(){};  
        virtual void h() final {}; // ok aussi  
};  
class B : public A {  
    public:  
        void f() final {};  
  
};  
class C : public B {  
    public:  
  
};
```

**final** sert aussi pour empêcher une redéfinition d'une méthode virtual (dynamiquement liée)

```
class A {
    public:
        virtual void f(){};
        virtual void h() final {}; // ok aussi
        void g() final {}; // non, final est associé à virtual
};
class B : public A {
    public:
        void f() final {};
};
class C : public B {
    public:
};
```