L’HERITAGE

- **L’héritage simple**
- **Mode de dérivation**
- **Redéfinition de méthodes dans la classe dérivée**
- **Ajustement d’accès**
- **Héritage des constructeurs/destructeurs**
- **Héritage et amitié**
- **Conversion de type dans une hiérarchie de classes**
- **Héritage multiple**
- **Héritage virtuel**
- **Polymorphisme**
- **Classes abstraites**

L’HERITAGE SIMPLE

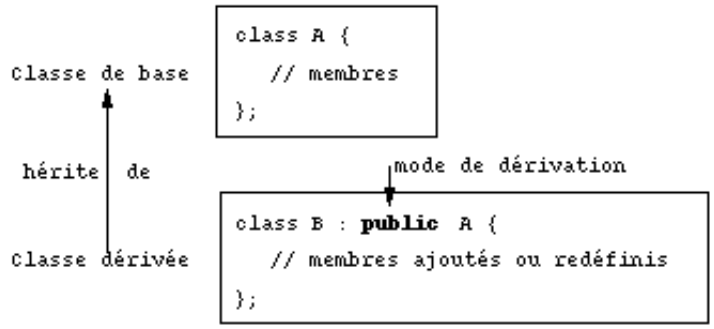
L’héritage, également appelé **dérivation**, permet de créer une nouvelle classe à partir d’une classe déjà existante, **la classe de base** (ou super classe).

"**Il est plus facile de modifier que de réinventer**"

La nouvelle classe (ou **classe dérivée** ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la classe de base et ainsi réutiliser le code déjà ecrit pour la classe de base.

On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.

Syntaxe :



La classe *B* hérite de façon publique de la classe *A*.

Tous les membres publics ou protégés de la classe *A* font partis de l'interface de la classe *B*.

MODE DE DERIVATION

Lors de la définition de la classe dérivée il est possible de spécifier le **mode de dérivation** par l'emploi d'un des mots−clé suivants :

public, *protected* ou *private*.

Ce mode de dérivation détermine quels membres de la classe de base sont accessibles dans la classe dérivée.

Au cas où aucun mode de dérivation n'est spécifié, le compilateur C++ prend par défaut le mot−clé *private* pour une classe et *public* pour une structure.

Les membres privés de la classe de base ne sont jamais accessibles par les membres des classes dérivées.

Héritage public :

Il donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée.

C'est la forme la plus courante d'héritage, car il permet de modéliser les relations "Y est une sorte de X" ou "Y est une spécialisation de la classe de base X".

Exemple :

```
class Vehicule {
  public:
```

```
    void pub1();
  protected:
    void prot1();
  private:
    void priv1();
};

class Voiture : public Vehicule {
  public:
    int pub2() {
      pub1();      // OK
      prot1();     // OK
      priv1();     // ERREUR
    }
};

Voiture safrane;
safrane.pub1();  // OK
safrane.pub2();  // OK
```

Héritage privé :

Il donne aux membres publics et protégés de la classe de base le statut de membres privés dans la classe dérivée.

Il permet de modéliser les relations "Y est composé de un ou plusieurs X" .

Plutôt que d'hériter de façon privée de la classe de base X, on peut faire de la classe de base une donnée membre (composition).

Exemple :

```
class String {
  public:
    int length();
  // ...
};

class Telephone_number : private String {
   void f1() {
     // ...
     l = length();  // OK

};

Telephone_number tn;
cout << tn.length();      // ERREUR
```

Héritage protégé :

Il donne aux membres publics et protégés de la classe de base le statut de membres protégés dans la classe dérivée.

L'héritage fait partie de l'interface mais n'est pas accessible aux utilisateurs.

Exemple :

```
class String {
  protected:
    int n;
};

class Telephone_number : protected String {
  protected:
    void f2() { n++; }  // OK
};

class Local_number : public Telephone_number {
  protected:
    void f3() { n++; } // OK
};
```

Tableau résumé de l'accès aux membres

		Statut dans la classe de base	Statut dans la classe dérivée
mode de dérivation	**public**	**public**	**public**
		protected	**protected**
		private	**inaccessible**
	protected	**public**	**protected**
		protected	**protected**
		private	**inaccessible**
	private	**public**	**private**
		protected	**private**
		private	**inaccessible**

REDEFINITION DE METHODES DANS LA CLASSE DERIVEE

On peut redéfinir une fonction dans une classe dérivée si on lui donne le même nom que dans la classe de base.

Il y aura ainsi, comme dans l'exemple ci après, deux fonctions *f2()*, mais il sera possible de les différencier avec l'opérateur **::** de résolution de portée.

Exemple :

```
class X {
  public:
    void f1();
    void f2();
  protected:
    int xxx;
};

class Y : public X {
  public:
    void f2();
    void f3();
};

void Y::f3() {
  X::f2();     // f2 de la classe X
  X::xxx = 12; // accès au membre xxx de la classe X
  f1();        // appel de f1 de la classe X
  f2();        // appel de f2 de la classe Y
}
```

AJUSTEMENT D'ACCES

Lors d'un héritage protégé ou privé, nous pouvons spécifier que certains membres de la classe ancêtre conservent leur mode d'accès dans la classe dérivée.

Ce mécanisme, appelé déclaration d'accès, ne permet en aucun cas d'augmenter ou de diminuer la visibilité d'un membre de la classe de base.

Exemple :

```
class X {
  public:
    void f1();
    void f2();
  protected:
    void f3();
    void f4();
};

class Y : private X {
  public:
    X::f1; // f1() reste public dans Y
    X::f3; // ERREUR: un membre protégé ne peut pas devenir public
  protected:
    X::f4; // f3() reste protégé dans Y
    X::f2; // ERREUR: un membre public ne peut pas devenir protégé
```

```
};
```

HERITAGE DES CONSTRUCTEURS/DESTRUCTEURS

Les constructeurs, constructeur de copie, destructeurs et opérateurs d'affectation ne sont jamais hérités.

Les constructeurs par défaut des classes de bases sont automatiquement appelés avant le constructeur de la classe dérivée.

Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, vous devez employer une **liste d'initialisation**.

L'appel des destructeurs se fera dans l'ordre inverse des constructeurs.

Exemple 1 :

```
class Vehicule {
  public:
    Vehicule() { cout<< "Vehicule" << endl; }
    ~Vehicule() { cout<< "~Vehicule" << endl; }
};

class Voiture : public Vehicule {
  public:
    Voiture() { cout<< "Voiture" << endl; }
    ~Voiture() { cout<< "~Voiture" << endl; }
};

void main() {
  Voiture *R21 = new Voiture;
  // ...
  delete R21;
}
/********** se programme affiche :
Vehicule
Voiture
~Voiture
~Vehicule
*****************************************/
```

Exemple d'appel des constructeurs avec paramètres :

```
class Vehicule {
  public:
     Vehicule(char *nom, int places);
     //...
```

```
};

class Voiture : public Vehicule {
  private:
    int _cv;    // puissance fiscale
  public:
    Voiture(char *n, int p, int cv);
    // ...
};

Voiture::Voiture(char *n, int p, int cv): Vehicule(n, p), _cv(cv)
{ /* ... */ }
```

HERITAGE ET AMITIE

- L'amitié pour une classe s'hérite, mais uniquement sur les membres de la classe hérités, elle ne se propage pas aux nouveaux membres de la classe dérivée et ne s'étend pas aux générations suivantes.

  Exemple :

```
class A {
  friend class test1;
  public:
    A( int n= 0): _a(n) {}
  private:
    int _a;
};

class test1 {
  public:
    test( int n= 0): a0(n) {}
    void affiche1() {
      cout << a0._a << // OK: test1 est amie de A
    }
  private:
    A a0;
};

class test2: public test {
  public:
    test2( int z0= 0, int z1= 0): test( z0), a1( z1) {}
    void Ecrit() {
      cout << a1._a; // ERREUR: test2 n'est pas amie de A
    }
  private:
    A a1;
};
```

- L'amitié pour une fonction ne s'hérite pas.

A chaque dérivation, vous devez redéfinir les relations d'amitié avec les fonctions.

CONVERSION DE TYPE DANS UNE HIERARCHIE DE CLASSES

Il est possible de convertir implicitement une instance d'une classe dérivée en une instance de la classe de base si l'héritage est public.

L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres de la classe dérivée.

Exemple :

```
class Vehicule {
  public:
    void f1();
  // ...
};

class Voiture : public Vehicule {
  public:
    int f1();
  // ...
};

void traitement1(Vehicule v) {
  // ...
  v.f1();   // OK
  // ...
}

void main() {
  Voiture R25;

  traitement1( R25 );
}
```

De la même façon on peut utiliser des pointeurs :

Un pointeur (ou une référence) sur un objet d'une classe dérivée peut être implicitement converti en un pointeur (ou une référence) sur un objet de la classe de base.

Cette conversion n'est possible que si l'héritage est public, car la classe de base doit posséder des membres public accessibles (ce n'est pas le cas d'un héritage protected ou private).

C'est le type du pointeur qui détermine laquelle des méthodes f1() est appelée.

```
void traitement1(Vehicule *v) {
  // ...
  v->f1();   // OK
  // ...
}

void main() {
  Voiture R25;

  traitement1( &R25 );
}
```

HERITAGE MULTIPLE

En langage C++, il est possible d'utiliser l'héritage multiple.

Il permet de créer des classes dérivées à partir de plusieurs classes de base.

Pour chaque classe de base, on peut définir le mode d'héritage.

```
class A {
  public:
    void fa() { /* ... */ }
  protected:
    int _x;
};

class B {
  public:
    void fb() { /* ... */ }
  protected:
    int _x;
};

class C: public B, public A {
  public:
    void fc();
};

void C::fc() {
  int i;
  fa();
  i = A::_x + B::_x; // résolution de portée pour lever l'ambiguïté
}
```

Ordre d'appel des constructeurs

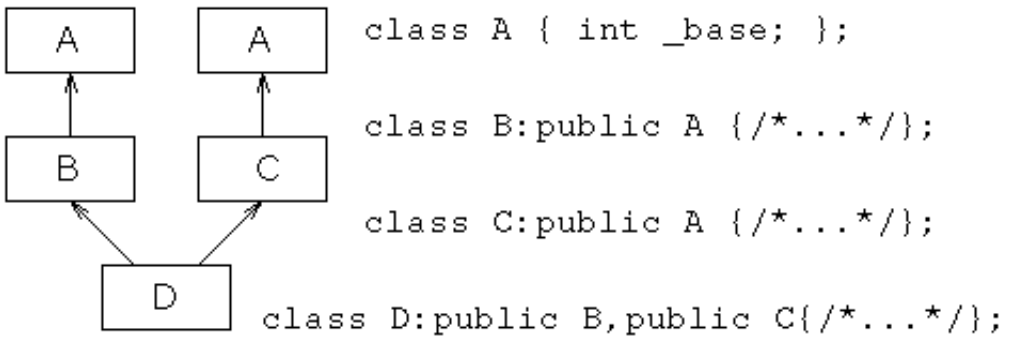
Dans l'héritage multiple, les constructeurs sont appelés dans l'ordre de déclaration de l'héritage.

Dans l'exemple suivant, le constructeur par défaut de la classe C appelle le constructeur par défaut de la classe B, puis celui de la classe A et en dernier lieu le constructeur de la classe dérivée, même si une liste d'initialisation existe.

```
class A {
  public:
    A(int n=0) { /* ... */ }
  // ...
};

class B {
  public:
    B(int n=0) { /* ... */ }
  // ...
};

class C: public B, public A {
//       ^^^^^^^^^^^^^^^^^
//       ordre d'appel des constructeurs des classes de base
//
  public:
    C(int i, int j) : A(i) , B(j) { /* ... */ }
  // ...
};

void main() {
  C objet_c;
  // appel des constructeurs B(), A() et C()
  // ...
}
```

Les destructeurs sont appelés dans l'ordre inverse de celui des constructeurs.

HERITAGE VIRTUEL



Un objet de la classe D contiendra deux fois les données héritées de la classe de base A, une fois par héritage de la classe B et une autre fois par C.

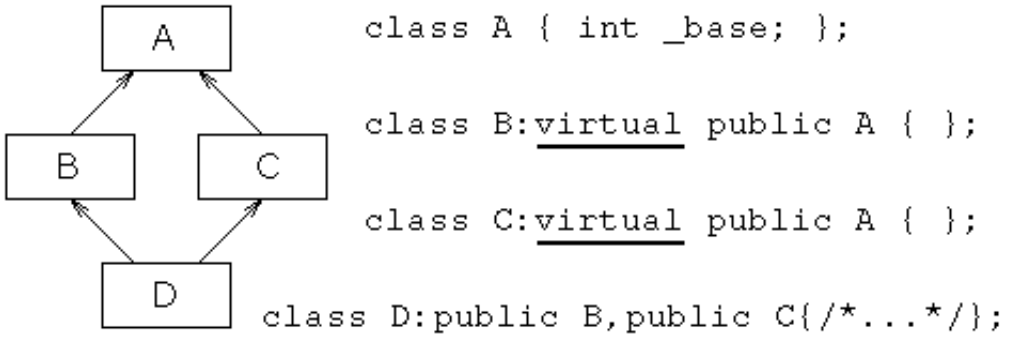
Il y a donc deux fois le membre _base dans la classe D.

L'accès au membre _base de la classe A se fait en levant l'ambiguïté.

```
void main() {
  D od;

  od._base = 0;     // ERREUR, ambiguïté
  od.B::_base = 1; // OK
  od.C::_base = 2; // OK
}
```

Il est possible de n'avoir qu'une occurence des membres de la classe de base, en utilisant l'**héritage virtuel**.

Pour que la classe D n'hérite qu'une seule fois de la classe A, il faut que les classes B et C héritent virtuellement de A.



Permet de n'avoir dans la classe D qu'une seule occurence des données héritées de la classe de base A.

```
void main() {
  D od;

  od._base = 0; // OK, pas d'ambiguïté
}
```

Il ne faut pas confondre ce statut "virtual" de déclaration d'héritage des classes avec celui des membres virtuels que nous allons étudier. Ici, Le mot-clé *virtual* précise au compilateur les classes à ne pas dupliquer.

POLYMORPHISME

L'**héritage** nous permet de réutiliser le code écrit pour la classe de base dans les autres classes de la hiérarchie des classes de votre application.

Le **polymorphisme** rendra possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérachie des classes.

En C++, le polymorphisme est mis en oeuvre par l'utilisation des **fonctions virtuelles**.

Fonctions virtuelles

```
class ObjGraph {
 public:
   void print() const { cout <<"ObjGraph::print()"; }
};

class Bouton: public ObjGraph {
  public:
    void print() const { cout << "Bouton::print()"; }
};

class Fenetre: public ObjGrap {
  public:
    void print() const { cout << "Fenetre::print()"; }
};

void traitement(const ObjGraph &og) {
  // ...
  og.print();
  // ...
}

void main() {             // Qu'affiche ce programme ???
  Bouton OK;
  Fenetre windows97;

  traitement(OK);         // affichage de .....
  traitement(Window97);   // affichage de .....
}
```

Comme nous l'avons déjà vu, l'instruction *og.print()* de *traitement()* appelera la méthode *print()* de la classe *ObjGraph*.

La réponse est donc :

```
  traitement(OK);          // affichage de ObjGraph::print()
  traitement(Window97);    // affichage de ObjGraph::print()
```

```
}
```

Si dans la fonction *traitement()* nous voulons appeler la méthode *print()* selon la classe à laquelle appartient l'instance, nous devons définir, dans la classe de base, la méthode *print()* comme étant **virtuelle** :

```
class ObjGraph {
  public:
    // ...
    virtual void print() const {
       cout<< "ObjetGraphique::print()" << endl;}
};
```

Pour plus de clarté, le mot−clé *virtual* peut être répété devant les méthodes *print()* des classes *Bouton* et *Fenetre* :

```
class Bouton: public ObjGraph {
  public:
    virtual void print() const {
      cout << "Bouton::print()";
    }
};

class Fenetre: public ObjGrap {
  public:
    virtual void print() const {
      cout << "Fenetre::print()";
    }
};
```

On appelle ce comportement, le **polymorphisme**.

Lorsque le compilateur rencontre une méthode virtuelle, il sait qu'il faut attendre l'exécution pour déterminer la bonne méthode à appeler.

Destructeur virtuel

Il ne faut pas oublier de définir le destructeur comme "virtual" lorsque l'on utilise une méthode virtuelle :

```
class ObjGraph {
  public:
    //...
    virtual ~ObjGraph() { cout << "fin de ObjGraph\n"; }
};

class Fenetre : public ObjGraph {
  public:
    // ...
    ~Fenetre() { cout << "fin de Fenêtre  "; }
};
```

```
void main() {
  Fenetre *Windows97 = new Fenetre;
  ObjGraph *og = Windows97;
  // ...
  delete og;  // affichage de :    fin de Fenêtre  fin de ObjGraph
              // si le destructeur n'avait pas été virtuel,
              // l'affichage aurait été :          fin de ObjGraph
}
```

- Un constructeur, par contre, ne peut pas être déclaré comme virtuel.
- Une méthode statique ne peut, non plus, être déclaré comme virtuelle.
- Lors de l'héritage, le statut de l'accessibilité de la méthode virtuelle (public, protégé ou privé) est conservé dans toutes les classes dérivée, même si elle est redéfinie avec un statut différent. Le statut de la classe de base prime.

CLASSES ABSTRAITES

Il arrive souvent que la méthode virtuelle définie dans la classe de base serve de cadre générique pour les méthodes virtuelles des classes dérivées. Ceci permet de garantir une bonne homogénéité de votre architecture de classes.

Une classe est dite abstraite si elle contient au moins une méthode virtuelle pure.

Une classe abstraite ne peut instancier aucun objet.

Méthode virtuelle pure

une telle méthode se déclare en ajoutant un **= 0** à la fin de sa déclaration.

```
class ObjGraph {
  public:
    virtual void print() const = 0;
};

void main() {
  ObjGraph og;  // ERREUR
  // ...
```

```
}
```

- On ne peut utiliser une classe abstraite qu'à partir d'un pointeur ou d'une référence.
- Contrairement à une méthode virtuelle "normale", une méthode virtuelle pure n'est pas obligé de fournir une définition pour ObjGraph::print() .
- Une classe dérivée qui ne redéfinit pas une méthode virtuelle pure est elle aussi abstraite.
