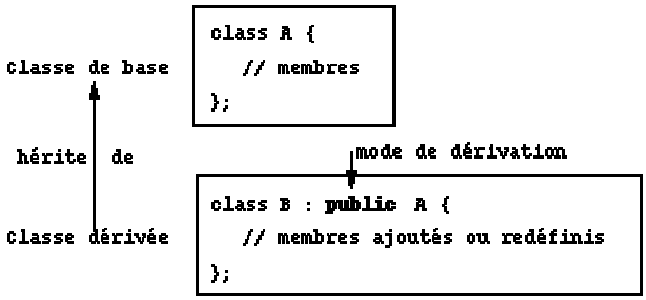
Héritage simple

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 de la classe de base, qui ne sont pas *privés*, et ainsi réutilise le code déjà écrit 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 partie de l'interface de la classe *B*.

Mode de dérivation

Lors de la définition de la classe dérivée il est possible de spécifier le **mode de dérivation** par l'emploi d'un des mots-clés suivants: *public*, *protected* ou *private*.

Le mode de dérivation détermine quels membres de la classe de base seront 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".

Héritage public

```
class Vehicule // classe de base
{
        public:
          void pub1();
        protected:
          void prot1();
        private:
          void priv1();
};
class Voiture : public Vehicule // classe dérivée
{
        public:
          int pub2() // méthode de la classe dérivée
          {
            pub1(); // OK
            prot1(); // OK
            priv1(); // ERREUR
          }
};

Voiture safrane; // instance de la classe Voiture
safrane.pub1(); // appel de la fonction pub1, OK
safrane.pub2(); // appel de la fonction 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".

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

class Telephone_number : private String // classe dérivée
{
      void f1()
      {   l = length();   }// OK
};
Telephone_number tn; // instance de la classe dérivée dans main
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 protégé fait partie de l'interface mais n'est pas accessible aux utilisateurs.

```
class String // classe de base
{
   protected:
      int n;
};
class Telephone_number : protected String // classe dérivée
{
   protected:
      void f2() { n++; } // OK
};
class Local_number : public Telephone_number // classe dérivée
{
   protected:
      void f3() { n++; } // OK
};
```

Tableau résumé d'accès aux membres des classes

		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

Redéfinition des méthodes dans la classe dérivée

On peut redéfinir une fonction dans la classe dérivée en lui donnant 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 à l'aide de l'opérateur de la résolution de portée **::** .

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

class Y : public X    // classe dérivée
{ public:
    void f2(); // méthode redéfinissant la fonction f2 de X
    void f3();
};

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

Ajustement d'accès

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.

```
class X        // classe de base
{    public:
        void f1();
        void f2();
     protected:
        void f3();
        void f4();
};
class Y : private X       // classe dérivée
{ public:
     X :: f1; // la fonction f1() reste publique dans Y
     X :: f3; // ERREUR: un membre protégé ne peut pas devenir public
   protected:
     X :: f4; // la fonction f4() reste protégée dans Y
     X :: f2; // ERREUR: un membre public ne peut pas devenir protégé
};
```

Héritage des constructeurs et destructeurs

Les constructeurs, destructeurs et opérateurs d'affectation ne sont jamais hérités. Les constructeurs par défaut des classes de base sont automatiquement appelés avant le constructeur de la classe dérivée. Pour ne pas appeler les constructeurs par défaut, mais les constructeurs avec des paramètres, il faut employer une *liste d'initialisation*. L'appel des destructeurs se fera dans l'ordre inverse des constructeurs.

```
class Vehicule    //classe de base
{ public:
    Vehicule() { cout<< "Vehicule" << endl; }
    ~Vehicule() { cout<< "~Vehicule" << endl; }
};
class Voiture : public Vehicule    // classe dérivée
{ public:
    Voiture() { cout<< "Voiture" << endl; }
    ~Voiture() { cout<< "~Voiture" << endl; }
};
void main ()
{   Voiture *R21 = new Voiture; // création d'un objet de la classe dérivée
    delete R21; // destruction de cet objet
} // Ce programme affiche:   Vehicule   Voiture   ~Voiture   ~Vehicule
```

Héritage et amitié (1)

L'amitié pour une classe peut être héritée, mais uniquement pour les membres de la classe dérivée qui ont été hérités de la classe de base. Elle ne se propage pas aux nouveaux membres de la classe dérivée et ne s'étend pas aux générations suivantes.

```
class A { // classe A

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

};

class test1 { // classe test1

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

};
```

Héritage et amitié (2)

```
class test2 : public test1 // classe test2, dérivée de test1
{
 public:
   test2 ( int z0 = 0, int z1 = 1): a0( z0 ), a1( z1 ) { }
   void ecrit() {cout << a1._a;}//ERREUR: test2 n'est pas amie de A
 private:
   A a0, a1;
};
```

L'amitié pour une fonction ne peut pas être héritée. À chaque dérivation, il faut redéfinir les relations d'amitiés avec les fonctions.

Conversion des types dans une hiérarchie (1)

Il est possible de convertir implicitement une instance de la 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.

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

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


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

void main()
{   Voiture R;
    traitement1( R );
}
```

Conversion des types dans une hiérarchie (2)

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 soit *public*, car la classe de base doit posséder des membres publics 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 );
}
```

Héritage multiple (1)

Il est possible d'utiliser l'héritage multiple en langage C++. 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   // classe de base A
{
    public:
      void fa() { /* ... */ }
    protected:
      int _x;
};

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

Héritage multiple (2)

```
class C : public B, public A // héritage multiple
// la classe C hérite des classes A et B
{
    public:
      void fc();
};

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

Héritage multiple (3)

Ordre d'appel des constructeurs

En cas d'héritage multiple, les constructeurs sont appelés dans l'ordre de déclaration d'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 // classe de base A
{
    public:
      A (int n = 0) { /* ... */ } // constructeur de A
      // ...
};
class B // classe de base B
{
    public:
      B (int n = 0) { /* ... */ } // constructeur de B
      // ...
};
```

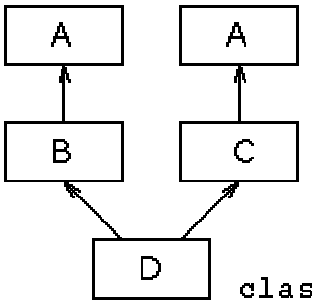
Héritage multiple (4)

```
class C : public B, public A // héritage multiple
// la classe C hérite des classes A et B
{
// 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.

Héritage virtuel (1)




```
class A { int _base; };

class B:public A {/*...*/};

class C:public A {/*...*/};

class D:public B,public C{/*...*/};
```

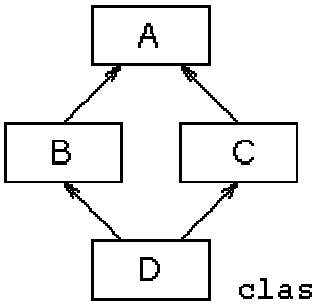
Un objet de la classe D contiendra deux fois les données héritées de la classe de base A, une fois par l'héritage de la classe B et une autre fois par C. Il y a donc deux fois le membre _base dans la classe D. Accès au membre _base de la classe A se fait en levant l'ambiguïté.

```
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 seule occurrence des membres de la classe de base, en utilisant l'**héritage virtuel**.

Héritage virtuel (2)

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. Cela permet d'avoir dans la classe D une seule occurrence des données héritées de la classe de base A.




```
class A { int _base; };

class B:virtual public A { };

class C:virtual public A { };

class D:public B,public C{/*...*/};
```

```
void main()
{
  D od;
  od._base = 0; // OK, pas d'ambiguïté
}
```

Il ne faut pas confondre le *statut virtuel* de déclaration d'héritage des classes avec celui des membres virtuels d'une classe (voir les transparents ci-après).

Polymorphisme (1)

Le mécanisme d'*héritage* nous permet de réutiliser le code écrit pour la classe de base dans les autres classes de la hiérarchie des classes de l'application. Le *polymorphisme* rendra possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérarchie des classes. En C++, le polymorphisme est mis en oeuvre par l'utilisation des **fonctions virtuelles**.

Fonctions virtuelles

```
class ObjGraph        // classe de base
{
  public:
  void print() const { cout <<"ObjGraph::print()"; }
};
class Bouton: public ObjGraph // première classe dérivée
{
public:
  void print() const { cout << "Bouton::print()"; }
};
```

Polymorphisme (2)

```
class Fenetre: public ObjGraph // deuxième classe dérivée
{ public:
  void print() const { cout << "Fenetre :: print()"; }
};

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

void main() { // Qu'affiche ce programme ???
  Bouton OK;
  Fenetre Window;
  traitement(OK); // affichage de .....
  traitement(Window); // affichage de .....
}
```

L'instruction *og.print()* de *traitement()* appellera la méthode *print()* de la classe *ObjGraph*. La réponse est donc:

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

traitement(Window); // affichage de ObjGraph :: print()
```

Polymorphisme (3)

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 // la méthode virtuelle
  {
       cout<< "ObjetGraphique::print()" << endl;
  }
};
```

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.

Polymorphisme (4)

Pour plus de clarté, le mot-clé *virtual* peut aussi 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()";
  }
};
```

Destructeurs virtuels (1)

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

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

};

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

// destructeur de classe dérivée

};
```

Destructeurs virtuels (2)

```
void main()
{
  Fenetre *Windows = new Fenetre;
  ObjGraph *og = Windows;
  // ...
  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 déclarée 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 dérivées, même si elles sont redéfinies avec un statut différent. Le statut de la classe de base prime.

Classes abstraites (1)

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; // méthode virtuelle pure
};
void main()
{
  ObjGraph og; // ERREUR
  // ...
}
```

Méthodes virtuelles pures

- 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 obligée de fournir une définition pour ObjGraph::print() .

- Une classe dérivée qui ne redéfinit pas une méthode virtuelle pure est aussi une classe abstraite.