1. Définition

L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'héritage (pouvant parfois être appelé dérivation de classe) provient du fait que la classe dérivée (la classe nouvellement créée, ou classe fille) contient les attributs et les méthodes de sa classe mère (la classe dont elle dérive). L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles hérités.

Par ce moyen, une **hiérarchie de classes** de plus en plus spécialisées est créée. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante. De cette manière il est possible de récupérer des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir.

Une particularité de l'héritage est qu'un objet d'une classe dérivée est aussi du type de la classe mère...

2. Type d'héritage

En C++, il existe trois types d'héritage définis par les mots clés public, protected, private. Un type d'héritage définie les accès possibles qu'une classe dérivée a vis à vis des attributs et méthodes de la classe de base. Le prototype de l'héritage se fait à la définition de la classe :

```
class NomClasseDerivee : TypeHeritage NomClasseDeBase
{
// définition des attributs et méthodes
// de la classe dérivée
...
};
```

Héritage public :

L'héritage public se fait en C++ à partir du mot clé public. C'est la forme la plus courante de dérivation (et la seule décrite en UML). Les principales propriétés liées à ce type de dérivation sont les suivantes :

• les membres (attributs et méthodes) publics de la classe de base sont accessibles par **tout le monde**, c'est à dire à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de la classe dérivée;

• les membres protégés de la classe de base sont accessibles aux fonctions membres de la classe dérivée, mais pas aux utilisateurs de la classe dérivée;

• les membres privés de la classe de base sont inaccessibles à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de la classe dérivée.

De plus, tous les membres de la classe de base conservent dans la classe dérivée le statut qu'ils avaient dans la classe de base. Cette remarque n'intervient qu'en cas de dérivation d'une nouvelle classe à partir de la classe dérivée.

Héritage privé :

L'héritage privé se fait en C++ à partir du mot clé private. Les principales propriétés liées à ce type de dérivation sont les suivantes :

• les membres (attributs et méthodes) publics de la classe de base sont inaccessibles à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de cette classe dérivée ;

• les membres protégés de la classe de base restent accessibles aux fonctions membres de la classe dérivée mais pas aux utilisateurs de cette classe dérivée. Cependant, ils seront considéré comme privés lors de dérivation ultérieures.

Héritage protégé :

L'héritage protégé se fait en C++ à partir du mot clé protected. Cette dérivation est intermédiaire entre la dérivation publique et la dérivation privée. Ce type de dérivation possède la propriété suivante : les membres (attributs et méthodes) publics de la classe de base seront considérés comme protégés lors de dérivation ultérieures.

Résumé des héritages

Le tableau suivant récapitule toutes les propriétés liées aux différents types de dérivation.

Classe de base	Dérivée publique		Dérivée protégée		Dérivée privée	
Statut initial	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur
public	public	O	protégé	N	privé	N
protégé	protégé	N	protégé	N	privé	N
privé	privé	N	privé	N	privé	N

Voici un exemple d'utilisation d'un héritage public :

```
//fichier Point.h

class Point
{
 protected:
 float X,Y;
 public:
 Point(float a, float b) {
 X=a;
```

```
Y=a; }

void SetX(double a) {X=a;}
void SetY(double b) {Y=b;}
};

//fichier PointColor.h

class PointColor : public Point
{
 private:
 short couleur;
 public:
 void SetCouleur(short a)
 { couleur=a; }
 void Afficher()
 {
 cout << "La couleur en ";
 cout << X << " " << Y;
 cout << " = " << couleur;
 }
 };

 #include "PointColor.h"
 #include <iostream>

using namespace std;

int main()
{
 PointColor pt; // Objet instancie
pt.setX(3.4); // Utilisation d'une methode de la classe de base
pt.SetY(2.6);

pt.SetCouleur(1); // Utilisation de methodes de la classe derivee

pt.Afficher();


return 0;

}
```

3. Appel des constructeurs et des destructeurs

Hiérarchisation des appels

Soit l'exemple suivant :

```
class A

{ .....

 public:

 A();

 ~A();

 .....

};

class B : public A
```

```
{ .....
public:
 B();
 ~B();
 .....
 };
```

Pour créer un objet de type B, il faut tout d'abord créer un objet de type A, donc faire appel au constructeur de A, puis le compléter par ce qui est spécifique à B en faisant appel au constructeur de B. Ce mécanisme est pris en charge par le C++: il n'y aura pas à prévoir dans le constructeur de B l'appel au constructeur de A.

La même démarche s'applique aux destructeurs: lors de la destruction d'un objet de type B, il y aura automatiquement appel du destructeur de B, puis appel de celui de A. Il est important de noter que les destructeurs sont appelés dans l'ordre inverse de l'appel des constructeurs.

Transmission d'informations entre constructeurs

En C++, il est possible de spécifier, dans la définition d'un constructeur d'une classe dérivée, les informations que l'on souhaite transmettre au constructeur de la classe de base. Voici un exemple d'héritage avec transmission d'informations entre constructeurs :

```
class Point
{
 protected:
 float X,Y;
 public:
 Point(float a, float b)
{
 X=a;
 Y=a;
 }
 };

 class PointColor : public Point
{
 private:
 short couleur;
 public:
 PointColor(float a, float b,
 short c) : Point(a,b)  {
```

```
 couleur=c;
 }
};
```

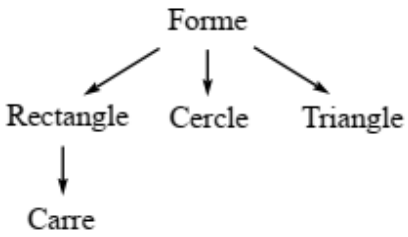
On rappelle aussi que l'on peut aller plus loin pour le passage des paramètres entre constructeur : un constructeur peut appeler explicitement chacun des constructeurs de ces champs. Voici comment la classe *Point* pourrait être écrite.

```
class Point
{
 protected:
 float X,Y;
 public:
 Point(float a, float b): X(a), Y(b)
{ } // il n'y a plus d'instructions !
};
```

Attention de bien utiliser le même ordre d'initialisation que celui de la déclaration des champs dans la classe, sinon un avertissement pourrait être donné par certains compilateurs.

4. Polymorphisme

Par défaut une méthode **AfficheAire**(), par exemple, d'une classe Carre dérivée d'une classe Rectangle de Carre réutilise le code de **AfficheAire**() de Rectangle. Mais dans d'autres cas, il peut être nécessaire d'écrire un code différent. Par exemple, dans une hiérarchie de classes de ce genre :



On a une version de **AfficheAire**() pour chacune de ces classes. Si ensuite on crée une collection d'objets de type Forme, en demandant **AfficheAire**() pour chacune de ces formes, ce sera automatiquement la version correspondant à chaque forme qui sera appelée et exécutée : on dit que **AfficheAire**() est polymorphe. Ce choix de la version adéquate de **AfficheAire**() sera réalisé au moment de l'exécution.

Toute fonction-membre de la classe de base devant être surchargée (c'est-à-dire redéfinie) dans une classe dérivée doit être précédée du mot virtual.

Exemple

```
# include <iostream.h>
class Forme {
// données et fonctions-membres....
public:
virtual void QuiSuisJe(void); // fonction destinée à être surchargée
};

void Forme::QuiSuisJe()
{
cout << "Je ne sais pas quel type de forme je suis !\n";
}

class Rectangle : public Forme {
// données et fonctions-membres....
public:
void QuiSuisJe(void);
};

void Rectangle::QuiSuisJe()
{
cout << "Je suis un rectangle !\n";
}

class Triangle : public Forme {
// données et fonctions-membres....
public:
void QuiSuisJe(void);
};

void Triangle::QuiSuisJe()
{
cout << "Je suis un triangle !\n";
}

void main() {
Forme *s;
char c;

cout << "Voulez-vous créer    1 : un rectangle ?\n";
cout << "                2 : un triangle ?\n";
cout << "                3 : une forme quelconque ?\n";
cin >> c;
switch (c)
{
case '1' :
case '2' :
case '3' :
}
s -> QuiSuisJe();

s = new Rectangle; break;
s = new Triangle; break;
s = new Forme;
 // (*) cet appel est polymorphe

 }
```

Remarque :

Pour le compilateur, à l'instruction marquée (*), il est impossible de savoir quelle version de QuiSuisJe() il faut appeler : cela dépend de la nature de la forme créée, donc le choix ne pourra être fait qu'au moment de l'exécution (on appelle cela *choix différé*, ou *late binding* en anglais).

Remarques :

- Une fonction déclarée virtuelle doit être définie, même si elle ne comporte pas d'instruction.
- Un constructeur ne peut pas être virtuel. Un destructeur peut l'être.