HERITAGE

HERITAGE 1
8. L'héritage simple. 2
8.1 Héritage simple : classe dérivée : 2
8.2 Statuts d'accès dans une classe de base (rappel) 3
8.3 Nouveau statut : protected. 3
8.4 Dérivation publique (la plus courante) 4
8.5 Dérivation privée 5
8.6 Dérivation protégée 6
8.7 Quelle dérivation utiliser ? (qq. idées) 7
8.8 Constructeur et destructeur d'une classe dérivée. 8
8.9 Constructeur de recopie 8
8.10 Opérateur d'affectation = ... 9
8.11 Redéfinition des membres 9

8. L'héritage simple.

8.1 Héritage simple : classe dérivée :

A partir d'une classe A, on définit une nouvelle classe *dérivée* B

A est la classe de **base**

B est la classe **dérivée** de A.

La classe dérivée B

hérite de tous les membres de A (données ou fonctions)

peut *ajouter* de nouveaux membres (données ou fonctions)

peut *redéfinir* des membres existants dans A (données ou fonctions).

```
class  Point                                  // classe de base
{
    int x, y;
public :
    Point(  int  abs = 0 ,  int ord = 0 )  { x = abs; y = ord ; }
    void affiche( );
    void deplace( int,  int );
    ...
} ;

class  PointColore : public Point       // classe dérivée publiquement
{
    int couleur;              // attribut supplémentaire
public :
    PointColore( int abs = 0 , int ord = 0 , int col = 0 ) ;   // nouveau
constructeur
    void affiche( );                  // rédéfinition de la fonction
    ...
} ;
```

Par dérivation la classe dérivée récupère tous les membres de la classe de base
sauf les constructeurs, le destructeur et l'opérateur =.

On peut dériver de nouveau à partir de la classe dérivée etc....

3 types de dérivations possibles : *publique*, *protégée* ou *privée*

8.2 Statuts d'accès dans une classe de base (rappel)

spécificateur d'accès :
Public : les membres publics de la classe **sont accessibles dans la classe et à l'extérieur de la classe**.
Private : les membres privés d'un objet **ne sont accessibles qu'à l'intérieur de la classe** par les objets de la classe (les fonctions membres de la classe).

8.3 Nouveau statut : protected.

Private et **Protected** :

- A l'extérieur de la classe : *private* et *protected* sont équivalents, les membres sont toujours inaccessibles.
- Seul un objet de la classe a accès aux membres privés et protégés (données ou fonctions) de sa classe.

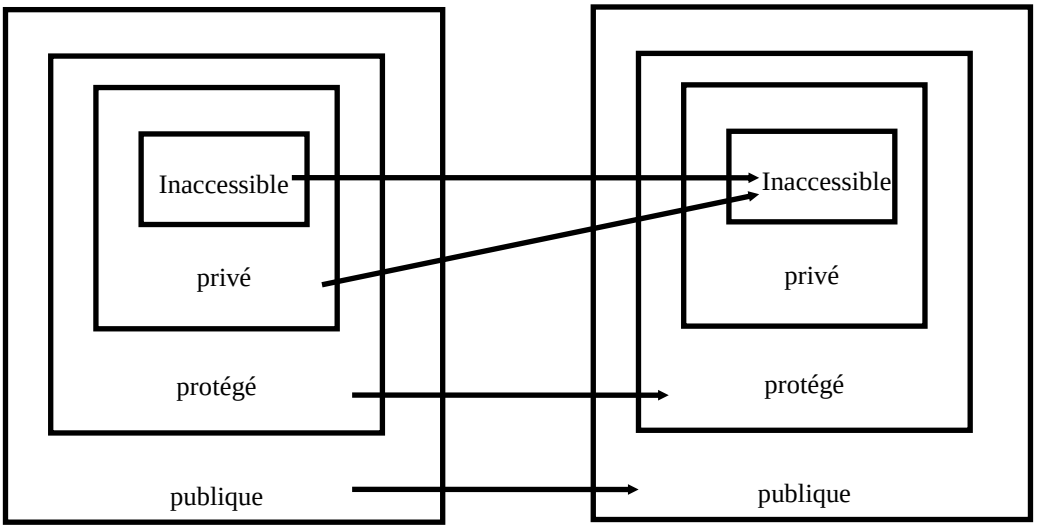
la différence est visible uniquement dans les opérations de dérivation.

- Une zone privée d'une classe devient toujours *inaccessible* (plus fort que privé) dans une classe dérivée, quelque soit la dérivation publique, protégée ou privée.
- Une zone protégée reste accessible (protégée) dans une classe dérivée publiquement.

		Statut des membres de base		
		public	**protected**	**private**
Mode de dérivation	**Public**	public	protected	*inaccessible*
	Protected	protected	protected	*inaccessible*
	Private	private	private	*inaccessible*

8.4 Dérivation publique (la plus courante)

Les membres privés de la classe de base deviennent inaccessibles dans la classe dérivée, les autres membres de la classe de base conservent leur statut dans la classe dérivée.



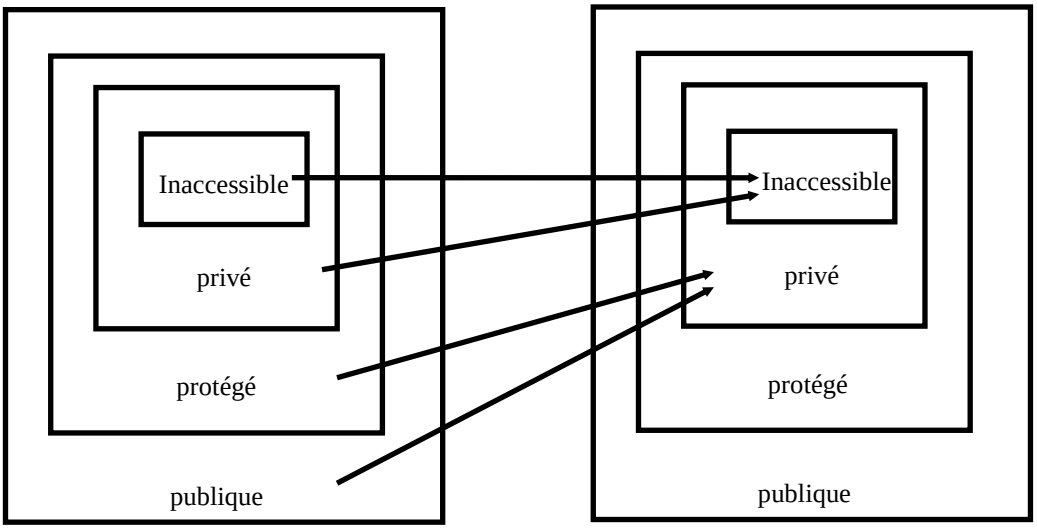
Classe de base **C. dérivée par héritage publique**

```
class A {
private  :    int x;
protected :  int y;
public :     int z ;
     void fa( );
} ;


class B : public A {
public : void fb( );
} ;

void A :: fa ( ) {
   x = 1;
   y = 1;
   z = 1;
}
```

```
void B :: fb ()  {
   x = 1;      // interdit
   y = 1;
   z = 1;
}

int main()  {
     A a;
     B b;

     a.x = 1;      // interdit
     a.y = 1;      // interdit
     a.z = 1;
     a.fa( ) ;

     b.x = 1;      // interdit
     b.y = 1;      // interdit
     b.z = 1;
     b.fa( ) ;
     b.fb( ) ;
}
```

8.5 Dérivation privée

Les membres privés de la classe de base deviennent inaccessibles dans la classe dérivée, les autres membres de la classe de base deviennent privés dans la classe dérivée.



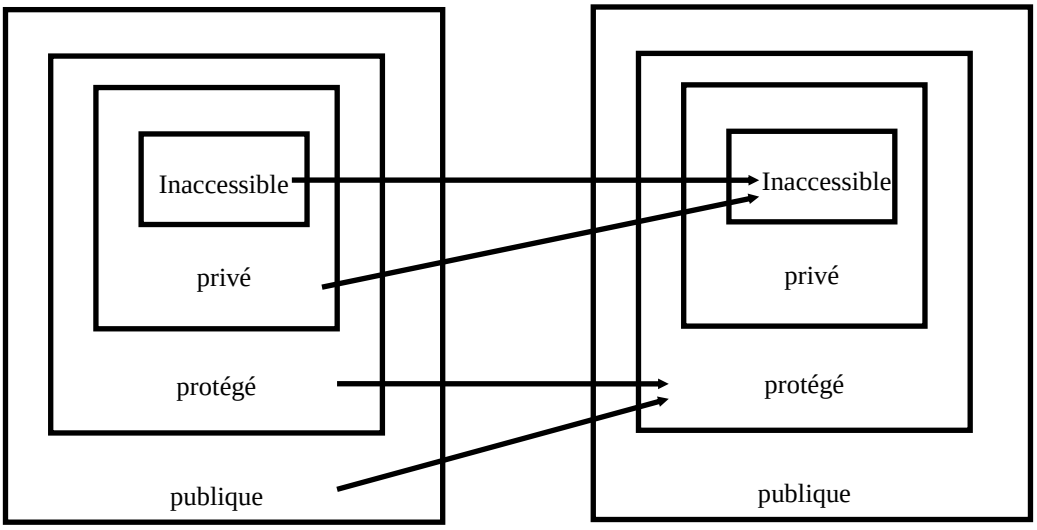
Classe de Base **C. dérivée par héritage privée**

```
class A {
private  :    int x;
protected :  int y;
public :     int z ;
     void fa( );
} ;


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

void A :: fa ( ) {
   x = 1;
   y = 1;
   z = 1;
}
```

```
void B :: fb ()  {
   x = 1;      // interdit
   y = 1;
   z = 1;
}

int main()  {
     A a;
     B b;

     a.x = 1;      // interdit
     a.y = 1;      // interdit
     a.z = 1;
     a.fa( ) ;

     b.x = 1;      // interdit
     b.y = 1;      // interdit
     b.z = 1;      // interdit
     b.fa( ) ;     // interdit
     b.fb( ) ;
}
```

8.6 Dérivation protégée

Les membres privés de la classe de base deviennent inaccessibles dans la classe dérivée, les autres membres de la classe de base deviennent protégés dans la classe dérivée.



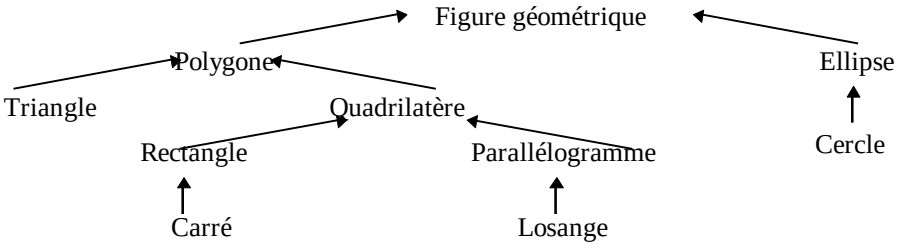
Classe de base **C. dérivée par héritage protégée**

```
class A {
private  :    int x;
protected :  int y;
public :     int z ;
     void fa( );
} ;


class B : protected A {
public : void fb( );
} ;

void A :: fa ( ) {
   x = 1;
   y = 1;
   z = 1;
}
```

```
void B :: fb ()  {
   x = 1;      // interdit
   y = 1;
   z = 1;
}

int main()  {
     A a;
     B b;

     a.x = 1;      // interdit
     a.y = 1;      // interdit
     a.z = 1;
     a.fa( ) ;

     b.x = 1;      // interdit
     b.y = 1;      // interdit
     b.z = 1;      // interdit
     b.fa( ) ;     // interdit
     b.fb( ) ;
}
```

8.7 Quelle dérivation utiliser ? (qq. idées)

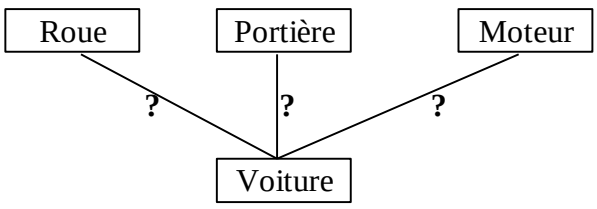
La dérivation publique s'utilise quand B *est une sorte de* A (AKO = A Kind Of)
(B est un cas particulier de A)



La dérivation privée s'utilise quand B *n'est pas* une sorte de A, la classe dérivée B restreint les fonctionnalités de A, on veut alors tout masquer à l'utilisateur extérieur.

exemple : Une classe Pile est créé à partir d'une classe Tableau. On se sert d'un tableau pour gérer une pile, mais une pile n'est pas véritablement une sorte de tableau.
On pourrait tout aussi bien utiliser une liste chainée pour gérer une pile, mais une pile n'est pas non plus une sorte de liste chaînée.

```
class Tableau
{    int *tab, nb;
public :
     Tableau( int );
     Tableau( const Tableau & );
     ~Tableau( );
     Tableau & operator = ( const Tableau & );
     int & operator [ ] ( int ) ;
     int taille( )  { return nb ; }
}

class Pile : private Tableau
{    int sommet;
public :
     Pile( int n )  : Tableau( n )    { sommet = -1 ; }
     void empile( int e )     { (*this)[ ++sommet ] = e ; }
     int depile( )            { return (*this)[ sommet-- ] ; }
     BOOL vide( )             { return sommet == -1 ; }
     BOOL plein( )            { return taille() == sommet ; }
}
```



Ne pas utiliser l'héritage mais l'inclusion
car une voiture *n'est pas une sorte de* roue ou de portière ou de moteur

8.8 Constructeur et destructeur d'une classe dérivée.

Lors de l'héritage, tous les constructeurs de la hiérarchie sont appelés du plus général au plus particulier (de la classe de base à la classe dérivée).

Deux possibilités :
1 - Il y a appel *implicite* des *constructeurs par défaut* de toutes les classes de base *avant* l'appel du constructeur de la classe dérivée.

2 - On peut faire un appel *explicite* à un *autre constructeur* d'une classe de base. On peut alors passer des paramètres. Pour faire un appel explicite il faut le signaler au niveau du constructeur.

```
// constructeur de la la classe PointColore

PointColore :: PointColore( int abs = 0, int ord = 0, int col = 0 ) : Point( abs, ord )
{
    couleur = col;                  //  ou couleur( col );
}
```

Le constructeur Point est d'abord appelé (initialisation de x et y)
puis le reste du constructeur est exécuté. (initialisation de couleur).

Pour le destructeur : même mécanisme mais dans l'ordre inverse.

8.9 Constructeur de recopie

Le constructeur de recopie comme tout constructeur n'est pas transmis par héritage.

- Si B, classe héritée de A ne définit pas de constructeur de recopie, les clones de B sont fabriquées par le constructeur *de recopie par défaut de B* qui fait appel au constructeur de *recopie de A (par défaut ou redéfini)* pour les membres hérités de A.

- Par contre, si B redéfinit un constructeur de recopie, le constructeur de recopie de A n'est pas automatiquement appelé. Il faut faire un appel explicite dans le constructeur de recopie de B d'un constructeur (de recopie ou un autre) de A. Si on ne fait pas d'appel explicite d'un constructeur de A, c'est le constructeur *par défaut de A* qui est appelé pour construire la partie héritée. S'il n'existe pas il y a erreur de compilation.

```
B :: B( const B &b ) : A( b )  { }
        // ici appel explicite d'un constructeur de recopie de A
```

8.10 Opérateur d'affectation =

L'opérateur d'affectation = n'est pas transmis par héritage.

- Si B, classe héritée de A n'a pas défini l'opérateur =, les affectations de B sont réalisées par *l'opérateur = par défaut de B*. Cet opérateur appelle l'opérateur = de A (par défaut ou redéfini) pour tous les membres hérités de A, *puis* exécute une affectation membre à membre pour tous les autres composants de B.

- Par contre si B, classe héritée de A, a défini l'opérateur =, les affectations de B sont réalisées par *l'opérateur = défini de B*, mais l'opérateur = de A ne sera pas appelé même s'il a été redéfini. L'opérateur = défini de B doit prendre en charge toute l'affectation, par exemple en appelant *explicitement* l'opérateur = de A.

8.11 Redéfinition des membres

```
int x;                  // x global

Class Base
{ public : int x ;
     void f( ) ;
}

class Derivee : public Base
{ public : int x ;
     void f( ) ;
}

void f( )
{
     x ++;              // incrémente x global
}

void Base :: f( )
{
     x ++;              // incrémente Base :: x
     :: x ++;           // incrémente  x global
}

void Derivee :: f( )
{    x ++;              // incrémente Derivee:: x
     Base :: x ++;      // incrémente Base :: x
     :: x ++;           // incrémente  x global
}
```