SURCHARGE

SOMMAIRE ....Erreur ! Signet non défini.
7. La surdéfinition d'opérateurs (ou surcharge). 2
7.1 Introduction : 2
7.2 Définition par une fonction membre... 2
7.3 Définition par une fonction amie. 2
7.4 L'opérateur = (affectation) 3

7. La surdéfinition d'opérateurs (ou surcharge).

7.1 Introduction :

Surdéfinition d'opérateur = donner une *nouvelle signification* à un opérateur sur des objets.

```
class String  {  .... } ;    // et on voudrait pouvoir faire (comme en Pascal)

String s1, s2 , s3 ;
s1 = s2 + s3 ;
if ( s1 == s2 ) {   ...    }                // etc....
```

Quelques restrictions:

la pluralité doit être respectée (unaire, binaire, ternaire) .
La priorité et l'associativité des opérateurs ne peuvent pas être modifiées.
La commutativité n'est pas forcément conservée.
La redéfinition doit porter au moins sur un opérande de type classe.

Pour définir un opérateur *op* il faut définir une fonction *operator op*.
Définition par une fonction membre ou amie, qqfois une seule forme est possible.

7.2 Définition par une fonction membre.

Le premier paramètre implicite est l'objet courant.
a op b génère un appel du type **a.operator op (b)**

```
class Complex
{      double x, y ;
public :
       Complex( double = 0, double = 0 );
       Complex operator + ( Complex );
       void operator  += ( Complex );
       ...
};
Complex  Complex :: operator + ( Complex z )
{
       return Complex( x + z.x, y + z.y );
};

void Complex :: operator += ( Complex z )
{
       x += z.x;
       y += z.y;
} ;

Complex z1, z2, z;
....z = z1 + z2 ;             // interprété comme z = z1.operator + ( z2 );
z1 += z2 ;              // interprété comme z1.operator += ( z2 );
```

7.3 Définition par une fonction amie.

une fonction amie (a op b)   génère un appel du type   **operator op(a, b)**

```
class Complex
{      double x, y ;
public :
       Complex( double = 0, double = 0 );
       friend Complex operator + ( Complex, Complex );
       void operator  += ( Complex );
       ...
```

```
}
Complex  operator + ( Complex a, Complex b )
{
	return Complex( a.x + b.x, a.y + b.y );
}

Complex z1, z2, z;
....z = z1 + z2 ;		// interprété comme z = operator + ( z1,  z2 );
```

7.4 L'opérateur = (affectation)

L'opérateur = a une signification prédéfinie, il réalise la copie membre à membre de l'objet.

La fonction *operator =* doit être obligatoirement une fonction membre.

```
class Tableau
{
	int *ptr, nb;	// nb = nombres d'éléments effectifs
public :
	Tableau ( int n ) { nb = n; ptr = new int[ nb ]; }
	Tableau ( const Tableau &t );
	~Tableau ( )  {  delete [ ] ptr ; }
	void operator = ( const Tableau &t );
	...
} ;

Tableau :: Tableau ( const Tableau &t )
{
	nb = t.nb;	ptr = new int [ nb ] ;
	memcpy( ptr, t.ptr, nb*sizeof( int ) ) ;
}

void Tableau :: operator = ( const Tableau &t )
{
	if ( &t == this )   return;		// !!!!
	delete [ ] ptr;
	nb = t.nb ;
	ptr = new int [ nb ] ;
	memcpy( ptr, t.ptr, nb*sizeof( int ) ) ;
}

Tableau t1, t2;
t1 = t2 ;		// équivalent à  t1.operator = ( t2 );
```

⇒ Il y a une différence importante entre le constructeur de recopie et l'opérateur =. Le constructeur de recopie manipule un objet *neuf* alors que l'opérateur = manipule un objet qui *existe déjà*. L'opérateur d'affectation doit donc nettoyer proprement l'objet courant avant de le réinitialiser, c'est un traitement supplémentaire par rapport au constructeur de recopie.

HERITAGE

HERITAGE 5
8. L'héritage simple. 6
8.1 Héritage simple : classe dérivée : 6
8.2 Statuts d'accès dans une classe de base (rappel) 7
8.3 Nouveau statut : protected. 7
8.4 Dérivation publique (la plus courante) 8
8.5 Dérivation privée 9
8.6 Dérivation protégée 10
8.7 Quelle dérivation utiliser ? (qq. idées) 11
8.8 Constructeur et destructeur d'une classe dérivée. 12
8.9 Constructeur de recopie 12
8.10 Opérateur d'affectation = ... 13
8.11 Redéfinition des membres 13

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.

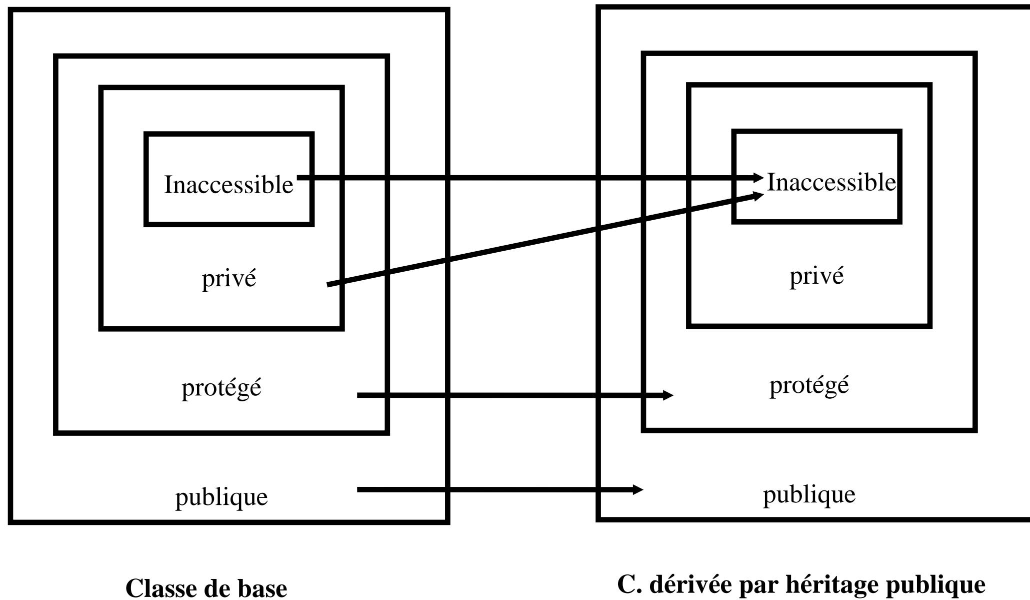
la différence est visible uniquement dans les opérations de dérivation.

- 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.



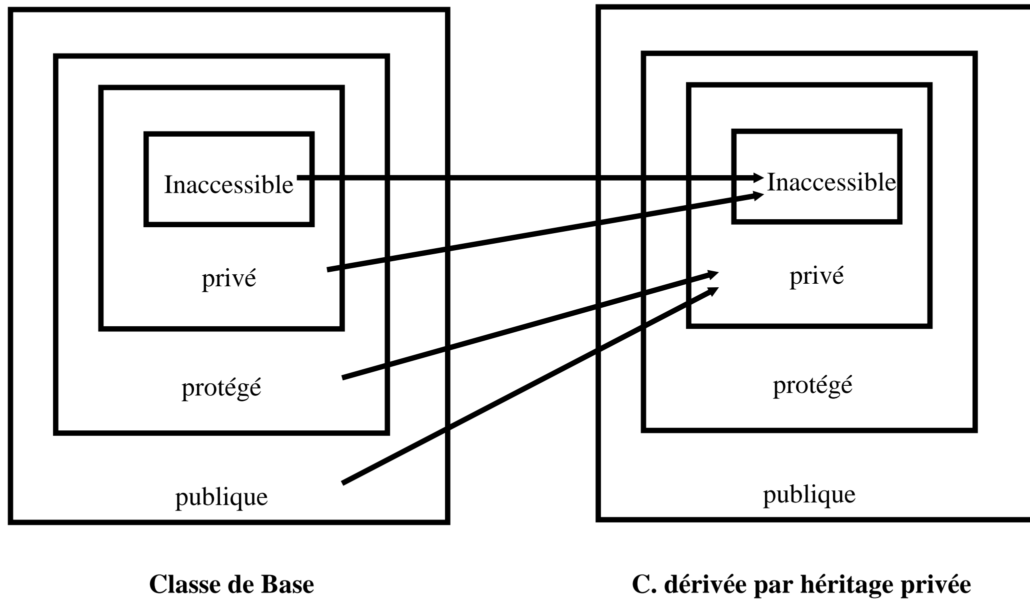
Classe de base **C. dérivée par héritage publique**

```
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.



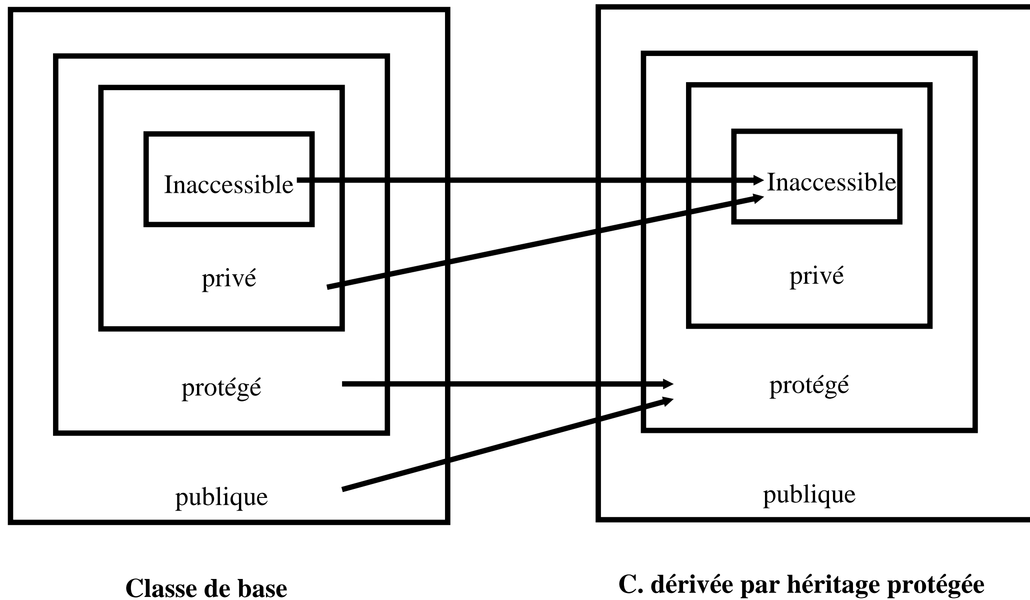
Classe de Base **C. dérivée par héritage privée**

```
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.

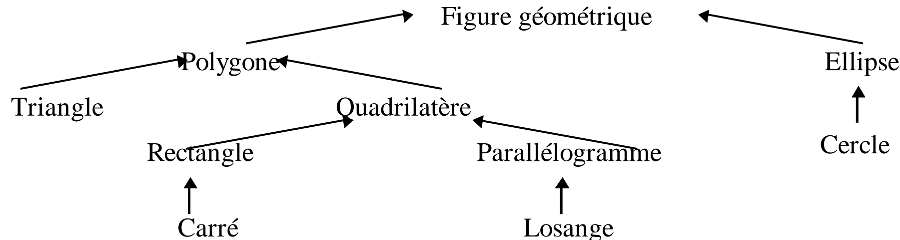



```
class A {
private  :    int x;
protected : int y;
public :      int z ;
      void fa( );
} ;


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)

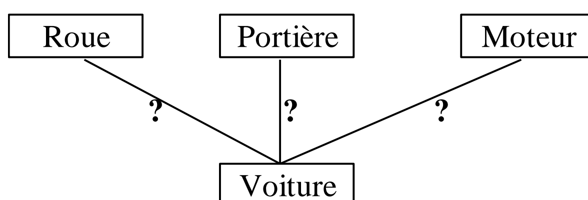
La dérivation publique s'utilise quand B *est une sorte de* A (AKO = A Kind Of)
(B est un cas particulier de A)



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
}
```

9. Polymorphisme et fonction virtuelle

9.1 Compatibilité entre objets d'une classe de base et objets d'une classe dérivée.

Si l'héritage est public

```
Class A  { ..... }
Class B : public A  { .... }

A a;          // a du type de base A
B b;          // b est du type B dérivé de A

A *pa;        // pa est un pointeur sur un objet de type A
B *pb;        // pb est un pointeur sur un objet de type B

a = b;        // ok
b = a;        // erreur de compilation

pa = pb;          // ok
pb = pa;          // erreur de compilation, mais ok pour :  pb = (B*)pa ;
```

a = b revient à convertir b dans le type A, en ne conservant que les membres hérités de A, et à affecter le résultat à *a*.
L'inverse est impossible.

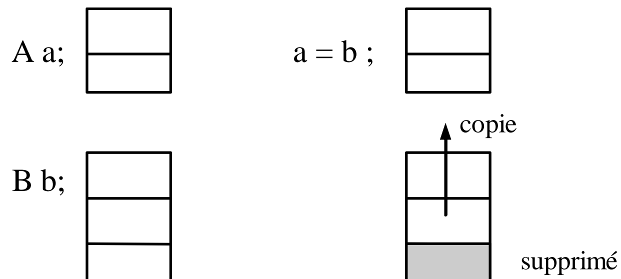


Figure 1

pa = pb : l'affectation se fait sans conversion de type (l'objet pointé par *pa* est toujours de type B)

soit A une classe et *p* un pointeur de type A*
Tout objet pointé par *p* est soit de type A, soit d'un type dérivé (publiquement) de A

soit A une classe et *r* une référence de type A&
Tout objet référencé par *r* est soit de type A, soit d'un type dérivé (publiquement) de A

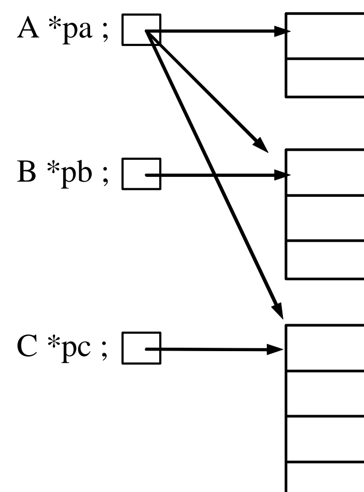


Figure 2

9.1.1 Polymorphisme

Par un héritage public, une classe B dérivée de A est considérée comme une sorte de A. Un objet de type B est aussi un objet de type A. Tous les comportements de A sont aussi les comportements de B.

9.1.2 type statique, type dynamique.

```
class  FigureGeometrique { .... } ;
class  Polygone : public FigureGeometrique { .... } ;
class  Triangle : public Polygone { .... } ;
class  Quadrilatere : public Polygone { .... } ;
class  Rectangle : public Quadrilatere { .... } ;
class  Carre: public Rectangle  { .... } ;
......
{
     FigureGeometrique *figure[ 5 ] ;
     figure[ 0 ] = new Triangle(…);
     figure[ 1 ] = new Carre(…);
     figure[ 2 ] = new Rectangle(…);
     figure[ 3 ] = new Polygone(…);
     figure[ 4 ] = new Quadrilatere(…);
}
```

soit un pointeur pa tel que : A *pa
et pa = &b ou pa = &c etc ....
le pointeur pa est de type A * : c'est le type statique, connu à la compilation.
Mais l'objet pointé *pa est soit de type A, B ou C : c'est le type dynamique, connu seulement à l'exécution.

9.2 Lien statique

```
Class A  { public : int i; }
Class B : public A  { public : int i; }

B b ;                              // même raisonnement avec les références ...
A *pa = &b;                        // ...  A &ra = b ;
int c = pa->i ;                    // ...  int c = ra.i ;

// Quelle est la valeur de  c ?  celle du membre i de A ou   i de B ?
```

Lors d'une indirection, est-ce le type du pointeur ou celui de l'objet pointé qui est pris en compte ?
réponse : en ce qui concerne les variables et les constantes, c'est le type du pointeur qui est pris en compte (type statique).

Par défaut la réponse est la même pour les fonctions.

```
class  FigureGeometrique { public : void tracer( )  { }  } ;

class  Polygone : public FigureGeometrique { public : void tracer( )  { .... }  } ;
class  Triangle : public Polygone { public : void tracer( )  { .... }   } ;
class  Quadrilatere : public Polygone { public : void tracer( )  { .... }    } ;
class  Rectangle : public Quadrilatere { public : void tracer( )  { .... }    } ;
class  Carre: public Rectangle  { public : void tracer( )  { .... }    } ;

{
     FigureGeometrique *figure[ 5 ] ;
     figure[ 0 ] = new Triangle(…);
     figure[ 1 ] = new Carre(…);
     figure[ 2 ] = new Rectangle(…);
     figure[ 3 ] = new Polygone(…);
     figure[ 4 ] = new Quadrilatere(…);

     for ( int i = 0; i < 5 ;  i++ ) figure[ i ]->tracer( ) ;
}
```

Ici la détermination de la fonction appelée est faite à la compilation.
Donc le compilateur installe dans l'appel la fonction définie par le pointeur, donc ici la fonction *tracer* de la classe FigureGeometrique !
Ce n'est pas ce qu'on espérait !

9.3 Lien dynamique

On souhaite que la fonction appelée soit celle correspondant au type de l'objet pointé

```
class  FigureGeometrique
{ public :
     virtual void tracer( )  { }                         // fonction vide
} ;

class  Polygone : public FigureGeometrique
{ public :
     virtual void tracer( )  { .... }                    // fonction redéfinie
} ;
class  Triangle : public Polygone
{ public :
     virtual void tracer( )  { .... }                    // fonction redéfinie
} ;
class  Quadrilatere : public Polygone
{ public :
     virtual void tracer( )  { .... }                    // fonction redéfinie
} ;
class  Rectangle : public Quadrilatere
{ public :
     virtual void tracer( )  { .... }                    // fonction redéfinie
} ;
class  Carre: public Rectangle
{ public :
     virtual void tracer( )  { .... }                    // fonction redéfinie
```

```
} ;

{
     FigureGeometrique *figure[ 5 ] ;
     figure[ 0 ] = new Triangle(…);
     figure[ 1 ] = new Carre(…);
     figure[ 2 ] = new Rectangle(…);
     figure[ 3 ] = new Polygone(…);
     figure[ 4 ] = new Quadrilatere(…);

     for ( int i = 0; i < 5 ;  i++ ) figure[ i ]->tracer( ) ;
}
```

ici la fonction appelée est véritablement la fonction appartenant à l'objet pointé.

Cette fonction ne peut être déterminée qu'*à l'exécution* en fonction de l'objet pointé : il s'agit d'un lien dynamique (par opposition au lien statique réalisé par l'éditeur de liens après la phase de compilation).

Remarques sur les fonctions virtuelles :

1. Si une fonction est déclarée virtuelle dans une classe de base, elle est *obligatoirement virtuelles dans les classes dérivées*.
2. Une fonction virtuelle de la classe de base sert de *fonction par défaut* dans les classes dérivées ou elle n'a pas été redéfinie.
3. Un constructeur ne peut pas être virtuel, mais il peut appeler une fonction virtuelle.
4. Un destructeur peut être virtuel (doit être déclaré virtuel en général).
5. L'appel d'une fonction virtuelle s'appuie sur une indirection, cette indirection consomme un peu de temps et d'espace mémoire supplémentaire par opposition à l'appel d'une fonction non virtuelle.
6. Les fonctions virtuelles permettent de définir des mécanismes généraux dans les classes de base. Ces mécanismes sont redéfinis (précisés) plus tard par les concepteurs des classes dérivées.
7. Mieux vaut mettre plus de fonctions virtuelles que le contraire.
8. Si une classe ne peut pas avoir de classes dérivées, inutile de déclarer les fonctions virtuelles.
9. Si une classe peut avoir des classes dérivées et si une fonction est très dépendante du type de la classe, (elle sera alors redéfinie dans la classe dérivée) alors elle doit être virtuelle.
10. Une classe possédant au moins une fonction virtuelle s'appelle une *classe polymorphe*.

Le mécanisme des fonctions virtuelles permet d'implémenter des *comportements généraux* dans la classe de base, les détails du comportement sont précisés dans les classes dérivées.

```
class FigureGéometrique {
      public :
            virtual void affiche() ;
            virtual void efface() ;
            void clignote() { while( 1 ) { affiche(); efface() ; } }
} ;

class Rectangle : public FigureGeometrique  {
      public :
            virtual void affiche() ;
            virtual void efface() ;
} ;

Rectangle r ;

r.clignote() ;
```

Figure 3

9.4 Fonctions virtuelles pures.

```
class  FigureGeometrique
{ public :
      virtual void tracer( )  = 0 ;
} ;
```

Une fonction virtuelle pure ne fait rien et ne doit jamais être appelée.
Elle sert de modèle aux fonctions de même nom des classes dérivées.
Les classes dérivées doivent *obligatoirement* redéfinir la fonction.

9.5 Classes abstraites

Une classe ayant au moins une fonction virtuelle pure est une classe abstraite.
Une classe ayant chacun de ses constructeurs soit protégé soit privé est une classe abstraite.
Une classe abstraite ne peut pas être instanciée (on ne peut pas créer un objet de son type).
Elle sert de moule pour dériver d'autres classes.

```
class  Figure
{ public :
      virtual void tracer( )  = 0 ;              // syntaxe !!!
} ;
Figure figure ;                    // erreur à la compilation
Figure *pfig ;                     // ok
pfig = new Figure ;                // erreur à la compilation
```

9.6 Table des méthodes virtuelles.

Les appels à une fonction membre *non virtuelle* sont « résolus » à la compilation. Les adresses réelles et définitives sont connues par le compilateur.

Les appels à une fonction membre *virtuelle* sont « résolus » à l'exécution en allant chercher l'adresse courante dans la table des méthodes virtuelles (TMV) de l'objet.

Class Point	Class Ligne : public Point
```Class Point { protected : int x, y; public : Point(int, int); ~Point() virtual void affiche(); virtual void efface(); void clignote(); }```	```Class Ligne : public Point { int xl, yl; public : Ligne(int, int, int, int); virtual void affiche(); virtual void efface(); }```

```
int main()
{
     Point p1, p2( 10, 20 );                // sizeof( p1 ) ---> 6
     Ligne l1, l2( 10, 10, 50, 50 );        // sizeof( l1  ) ---> 10
}
```

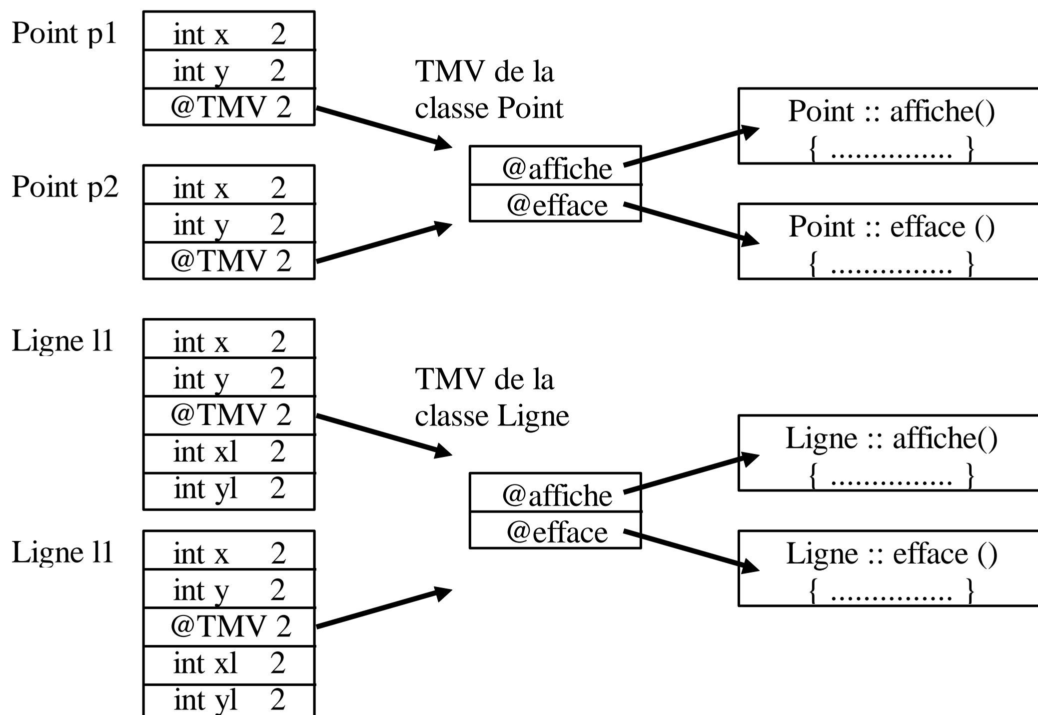


Figure 4