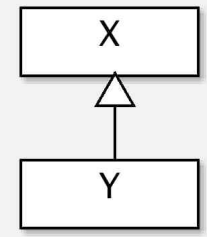
5 . HERITAGE, POLYMORPHISME, ABSTRACTION ET VIRTUALITE EN C++

➢ Héritage

Une classe peut hériter des propriétés (attributs et méthodes) d'une autre classe.

On dit que :
- Y hérite de, dérive, étend ou spécialise X
- Y est une clase fille ou sous classe de X
- X est une classe mère ou super classe ou classe de base de Y



On dit également :
- toute instance de Y *est aussi* ou *peut être manipulé comme* une instance de X

➢ Syntaxe

```
class <classe-dérivée> : <public,private,protected>
      <classe-de-base> {
      champs;
      fonctions membres;
};
```

Exemple :

```
// Y dérive de X
class Y : public X {
    ...
} ;
```

➢ Redéfinition d'une méthode

Une méthode définie dans une classe mère peut être *redéfinie* dans une classe fille (même signature).

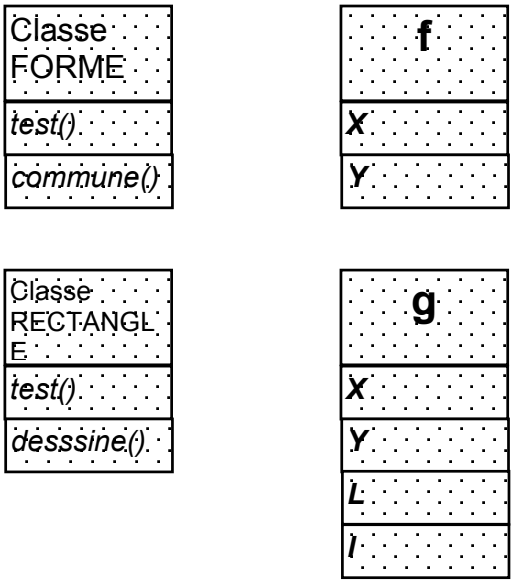
➢ Résolution de portée

Le code d'une méthode de la classe fille peut explicitement appeler une méthode de la classe mère avec la syntaxe :

```
<classe-base>::<nomDeMethode>()
```

EXEMPLE DE CLASSE HERITEE

```
class FORME {
    int X;
    int Y;
public:
    void aff()          { cout << "FORME" << X << Y ; }
    void commune()      {•••}
};

class RECTANGLE : public FORME {// RECTANGLE hérite de forme
    int L;
    int l;
public:
    void aff()     { FORME::aff() ; cout << "RECT" << L ;••• }
    void dessine()     {•••}
};

main() {
    FORME f;
    f. aff();       // FORME::aff()
    f.commune();        // FORME::commune()
    f.dessine() ;       // ERREUR :
                        //pas de méthode dessine() dans FORME
    RECTANGLE g;
    g.aff();        // RECTANGLE::aff ()
    g.commune();    // FORME::commune()
    g.dessinne();   // RECTANGLE::dessine()
}
```



POLYMORPHISME

➢ Polymorphisme

Capacité à voir un objet sous des formes différentes
En pratique, permet de manipuler un objet par un pointeur ou une référence d'une classe mere.

Le CERCLE c est manipulé via pf « comme » de FORME.

➢ Exemple

```
main() {
    FORME f, *pf=NULL, f2;
    CERCLE c, *pc=NULL, c2;
    f.aff();      // FORME::toString()
    f2 = c;                // OK ! Que se passe-t-il ?
        //1/ cast RECTANGLE -> FORME
        //2/ opérateur=(const FORME&)
        //   (opérateur d'affectation par défaut) de FORME !
        //Attention : f2 est un autre objet en mémoire.
    c2 = f;                // INTERDIT

    pc = &c;
    pc->aff();             // RECTANGLE::aff()
    pf = (FORME *) &c;     // OK un RECTANGLE est une FORME
                           // Le cast n'est pas obligatoire
    pf->aff();             // FORME::aff()

    pc = (CERCLE *) f;     // INTERDIT !
}
```

➢ Type statique, type dynamique

Type statique: type de la variable, de la réference ou du pointeur utilisé pour manipuler l'objet
Type dynamique: type effectif de l'objet manipulé en mémoire
:
Le type statique de pf est "pointeur sur FORME"..
Le type dynamique de l'objet pointé par pf est CERCLE.

CONVERSION CLASSE MERE <-> CLASSE FILLE

➢ Conversion entre classe fille et classe mère

Conversion **implicite** : lorsque la classe de base est dérivée publiquement, conversion implicite automatique entre un élément, un pointeur ou une référence à un élément de la classe dérivée vers la classe de base.

« Une instance de la sous classe *« est »* instance de la super-classe. »

La réciproque est fausse : un objet de base « *n'est pas* » un objet dérivé (un CERCLE est une FORME, mais une FORME n'est pas toujours un CERCLE).

➢ L'opérateur dynamic_cast< > : cast dans les hiérarchie de classes

Converti un pointeur sur un objet d'une classe de base vers un objet d'une classe de dérivée ou inversement.
Prend en compte le type dynamique (effectif) des objets manipulés.

- Si la conversion est possible, renvoie un pointeur valide
- Si la conversion est impossible, renvoie un pointeur nul sinon.

A éviter : souvent signe d'une mauvaise conception objet !!!

➢ Exemple

```
int main() {
   FORME f; RECTANGLE r; CERCLE c;
   FORME * tab_pf[4];
   tab_pf[0] = &f;
   tab_pf[1] = &r;
   tab_pf[2] = &c;
   tab_pf[3] = NULL;
   for (int i =0; i<4; i++) {
       CERCLE * pc  = dynamic_cast<CERCLE*> ( tab_pf[i] );
       if (pc)
          cout<<"CERCLE ; rayon=" << pc->rayon() << endl;
       else
          cout<<"L'objet pointé n'est pas un CERCLE"<<endl;
   }
}
```

Seul le troisième cast dynamique, sur tab_pf[2] renvoit un pointeur non NULL.
Tous les autres échouent (pointeur NULL).

HERITAGE, CONSTRUCTION, DESTRUCTION

➢ Construction et destruction d'un objet d'une classe dérivée

1. La mémoire est réservée pour stocker un objet de la classe dérivée.

2. Les objets sont construits et leurs constructeurs exécutés **de haut en bas** dans l'ordre de l'arbre d'héritage: l'objet de base, puis les membres de cet objet puis les aspects propres à la classe dérivée.

3. Les objets sont détruits dans l'ordre inverse : destructeur de la classe fille (destruction des membres de la classe fille) *puis* destructeur de la classe mère. L'appel du destructeur de la classe mère est automatique.

Remarque : le destructeur de la classe mère est en général déclaré virtual, cf ce qui suit.

➢ Exemple

```
class POINT { int X;   int Y;
public:
    POINT(int a, int b) {X=a; Y=b;}
    •••
};

class FORME { POINT origine;
public:
    FORME( int c, int d) : origine(c,d) {
      /* Rien a faire ici */
    }
};

// RECTANGLE derive de FORME
class RECTANGLE : public FORME { int L;  int l;
public:
    RECTANGLE(int a, int b, int c, int d) :
    FORME(a,b), // choix constructeur de la super classe
                // Toujours dans
                // la liste d'initialisation !
    L(c), l(d) {
          // ici le code
    }
};
```

DERIVATION ET PROTECTION

➢ Dérivation et protection

Soit une classe B qui hérite publiquement d'une classe A.

	Visivilité des membres de A dans...				
Membres de A	*...le code de A*	*... le code de B*	*... les fonctions Amies de A*	*... les fonctions Amies de B*	*... autre*
private	visible	**Non visible**	visible	Non visible	Non visible
protected	visible	**visible**	visible	visible	Non visible
public	visible	visible	visible	visible	visible

Ainsi:

- la classe dérivée n'a pas de droits privilégié sur la classe de base.
- les membres privés de la base sont inaccessibles

➢ Héritage `public` et `private`

La dérivation peut être privée ou publique.
L'évolution est toujours dans le sens de la restriction.

```
class A {...}

// héritage public. Le plus courant
class B  : public A { ....} ;

// héritage privé. Rarement utilisé
class C : A { ....} ;
// ou class C : private A { ....} ;
```

	Statut des membres de A dans la classe fille	
Membres de A	*Héritage public*	*Héritage privé*
private	*Non visible*	*Non visible*
protected	Membre protected	Membre private
public	Membre public	Membre private

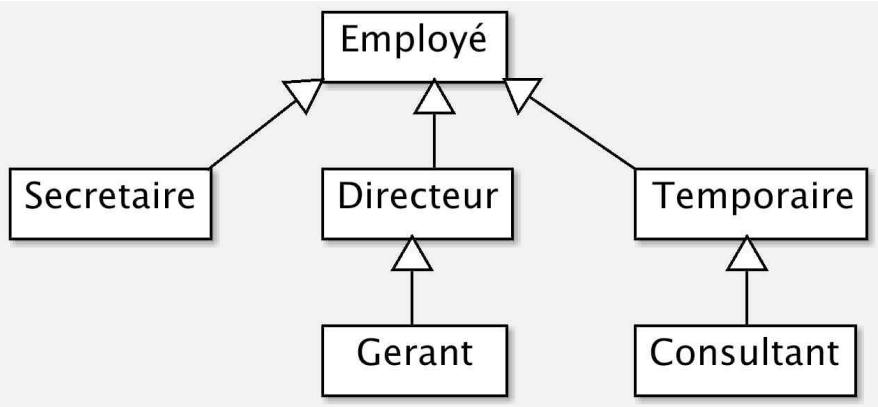
Notez qu'un attribut *public* ou *protected* dans la classe mère *reste accessible dans le code* de la classe fille, même avec l'héritage privé.

DERIVATION ET PROTECTION : EXEMPLE

```
class A {
      private :   int a;
      protected : int b;
      public :    int c;
};

class B : public A {
public:
      void f() {
          a=5;      // ERREUR : a private dans A => non visible dans B
          b=5;      // OK : b protected et héritage publique
          c=5;      // OK : c public
      }
      friend void g(B x);
};

class C : private A {
public :
      void f() {
          a=5;      // ERREUR : a private dans A => non visible dans B
          b=5;      // OK : héritage privé => b private dans C
          c=5;      // OK : héritage privé => c private dans C
      }
      friend void h (C x);
};

void g(B x) {
      x.a=5;        // ERREUR : a private dans A
      x.b=5;        // OK     : b protected dans B et g() amie de B
      x.c=5;        // OK     : c public dans B
}
void h(C x) {
      x.a=5;        // ERREUR : a private dans A
      x.b=5;        // OK     : b private dans C et h() amie de C
      x.c=5;        // OK     : c private dans C et h() amie de C
}

void r(B x) {
      x.a=5;        // ERREUR : a private dans A
      x.b=5;        // ERREUR : b protected dans B
      x.c=5;        // OK     : c public dans B
}
void r(C x) {
      x.a=5;        // ERREUR : a private dans A
      x.b=5;        // ERREUR : b private dans B (héritage privé)
      x.c=5;        // ERREUR : c private dans B (héritage privé)
}
```

HIERARCHIE DE CLASSES DERIVEES

Plusieurs classes peuvent dériver d'une même classe de base.
L'héritage est récursif : une classe dérivée peut servir de classe de base à une autre classe dérivée.




```
class Employe  {
protected:
    int salaire;
    char nom[50], prenom[50];
public:
    void paye()const { cout << salaire; }
    void aff()  const {
      cout << "Je suis "<<nom << " "<<prenom;
    }
};

class Secretaire : public Employe  {
    char nomdirecteur[50];
public:
    void aff() const {
      Employe::aff();
      cout<<" dans le service de "<< nomdirecteur;
    }
};
class Directeur : public Employe  {
    int primes;
public:
    void paye()const  {
      cout << " salaire : " << salaire+primes;
    }
};
```

HIERARCHIE DE CLASSES DERIVEES

```
class Temporaire : public Employe  {
    int date_debut, duree;
public:
    void aff() const {
      Employe::aff();
      cout << " embauché de "<< " date_debut "
           << " à "<< date_debut+duree;
    }
};

class Gerant : public Directeur  {
    char **societes;
public:
    void aff() const {
      Directeur::aff();
      cout << " des societes : " ;
      for (char *p=societes; p!=NULL; )
          cout << *p++;
    }
};



main() {
    Secretaire a1,a2;
    Directeur b1,b2,b3;
    Gerant c1;
    Temporaire x1,x2,x3,x4;
    Employe * tab[4];

    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&b2;


    // Valeurs des a1,a2...
    a1.aff(); b1.aff();
    // Ce qui suit n'exécute que Employe::aff()
    // car le type statique de tab[i] est Employe !
     for (int i=0; i<4; i++) tab[i]->aff();
}
```