LIAISON DYNAMIQUE (1)

Problème : avec un pointeur de base (Employe *), on aurait besoin que la « bonne » méthode soit exécutée, en fonction du type dynamique de l'objet pointé.

Comment savoir quel est le type dynamique (« réel ») de l'objet pointé (Secretaire, Directeur, ..) ?

Solution 1 : RTTI et dynamic_cast<>. A éviter !

```
class Employe  {
protected:
    int salaire; int prime;
    char nom[50], prenom[50];
public:
    void paye() const {
      if ( dynamic_cast<Secretaire *>(this) ) {
           cout << salaire;
      } else if ( dynamic_cast<Directeur *>(this) ) {
           cout << salaire + prime ;
      }
      •••
    }
};

class Secretaire : public Employe  {
    char directeur[50];
}

main() {
    Secretaire a1,a2;
    Directeur b1,b2;
    employe * tab[4];
    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&b2;

    // Valeurs des a1,a2...
    a1.paye(); b1.paye();

    // OK: Employe::paye() affiche
    // ce qu'il faut grace au dynamic_cast<>
    for (int i=0; i<4; i++) tab[i]->paye();
}
```

LIAISON DYNAMIQUE (2)

Solution 2 : champ de type (à la C) : à éviter

```
typedef enum { SECRETAIRE, DIRECTEUR, •••} TYPE_EMPLOYE ;

class Employe  {
protected:
    TYPE_EMPLOYE quijesuis;
    int salaire; int prime;
    char nom[50], prenom[50];
public:
    Employe(TYPE_EMPLOYE _quijesuis, •••) {•••}
    void paye() const {
     switch (quijesuis) {
          case SECRETAIRE : cout << salaire; break;
          case DIRECTEUR : cout << salaire+primes; break;
     }
    }
};

class Secretaire : public Employe  {
    char directeur[50];
public:  // ATTENTION au constructeur
    Secretaire(): Employe( SECRETAIRE, •••) { ••• ; }
};

main() {
    Secretaire a1,a2;
    Directeur b1,b2;
    employe * tab[4];
    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&b2;

    // Valeurs des a1,a2...
    a1.paye(); b1.paye();

    // OK: Employe::paye() sait toujours ce qui est manipulé
    for (int i=0; i<4; i++) tab[i]->paye();
}
```

Aucune de ces solutions n'est satisfaisante !

LIAISON DYNAMIQUE. METHODE VIRTUELLE. (1)

➢ Liaison dynamique : mot clé *virtual*.

1. Déclarer une méthode d'une super classe *virtuelle*
2. Redéfinir cette méthode dans les sous classes.

➔ la méthode exécutée sera celle de la classe du type **dynamique** de l'objet et sera choisie automatiquement à l'exécution.

Syntaxe :

```
class >ClasseMere> {
    virtual  <prototype de méthode>
} ;
```

➢ Mécanisme sous-jacent

Les fonctions virtuelles et le mécanisme de liaison dynamique sont implémentées par un tableau de pointeurs sur des fonctions et un champ entier indiquant le type de l'objet.

LIAISON DYNAMIQUE. METHODE VIRTUELLE. (2)

➢ Exemple

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

Les autres classes ne changent pas ! Eg, la méthode aff() de Secretaire reste :

```
void Secretaire::aff() const {
      Employe::aff();
      cout<<" dans le service de "<< nomdirecteur;
    }
```

Le main ne change pas :

```
main() {
    Secretaire a1, a2;
    Directeur b1;
    Gerant c1;
    Employe * tab[4];
    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&c1;

    // Employe::aff() est virtuelle
    // => la méthode aff() exécutée est celle
    // du type dynamique («  effectif ») de l'objet
    // quel que soit le type statique utilisé pour
    // manipuler l'objet
    for (int i=0; i<100; i++) tab[i]->aff();}
    // les méthodes aff() exécutées sont celles de
    // Secretaire, Secretaire, Directeur puis Gerant.
}
```

CLASSE ABSTRAITE, METHODE VIRTUELLE PURE (1)

➢ Méthode virtuelle pure

Méthodes qui ne peuvent avoir de sens ou dont le sens est incomplet

Syntaxe :

```
virtual type nom_methode (paramètres) = 0;
```

➢ Notion de classe abstraite

Classe dont l'implantation n'est pas complète:

- Elle représente un concept abstrait
- Elle possede au moins une méthode virtuelle pure
- Il est impossible de créer un objet de cette classe.
- On peut utiliser un pointeur ou une référence sur une classe abstraite (polymorphisme).

➢ Exemples de classes abstraites

Dans les exemples précédents du cours, typiquement :

- **FORME** serait une classe abstraite.
  FORME::dessiner() serait virtuelle pure.
  - On ne sait pas dessiner() une forme à ce niveau de la hiérarchie.
  - On ne peut écrire le code de dessiner que dans les sous-classes.

- **Employe** serait une classe abstraite
  Il n'est pas possible d'instancier un « Employe » sans plus de précision ; seules les sous classes peuvent être instanciées.
  Par exemple, Employe::afficher() serait virtuelle pure.

CLASSE ABSTRAITE, METHODE VIRTUELLE PURE (2)

➢ Exemple

```
class FORME { protected: int X; int Y;
public:
   virtual ~FORME() {} ;
   // Impossible de dessiner à ce niveau => virtuelle pure
   virtual void dessine() = 0;
   virtual void aff()=0 { cout << "forme " << X << x ; }
}; // FORME est abstraite car a une méthode virtuelle pure

class RECTANGLE : public FORME { int l; int L;
public:
   void dessine(){ goto(X,Y); draw_box(X,Y,X+L,Y+l); }
   void aff(){
        FORME::aff() ;
        cout << "rectangle " << l << L ; }
   }
};

class ELLIPSE : public FORME { int ga; int pa
public:
   void dessine(){ goto(X,Y); draw_circle(X,Y,ga,pa); }
   void aff(){
        FORME::aff() ;
        cout << "ellipse " << l << L ;
   }
};

main(){
   FORME a;  // IMPOSSIBLE : FORME est une classe abstraite
   ELLIPSE b;    // OK
   b.dessine();  // ELLIPSE::dessine()
   FORME *p = &b;
   p->dessine();     // OK, ELLIPSE::dessine()
}
```

CLASSE ABSTRAITE PURE

➢ Classe abstraite pure

- Classe qui n'a aucune méthode concrete et ne declare aucun attribut.
- Définit un contrat : oblige à avoir une API commune
  - toutes les sous classes doivent implanter toutes les méthodes déclarées dans la super classe.
- *Equivalente à la notion d'Interface en UML ou Java...*

➢ Exemple

La classe abstraite pure Motorisé declare une méthode allumerMoteur().
On peut avoir un conteneur d'objets motorisés et "démarrer" tous ces objets.

```
class Motorise {
public:
   virtual ~Motorise() {} // nécessaire...
   virtual void demarrer() = 0 ;
};
class Voiture : public Motorise {
public:
   Voiture() { cout<<"Constructeur Voiture "<<endl;}
   ~Voiture() { cout<<"Destructeur Voiture "<<endl;}
   virtual void demarrer(){cout<<"Demarrer Voiture"<<endl;}
};

class Perceuse : public Motorise {
public:
 virtual void demarrer(){cout<<"Demarrer Perceuse"<<endl;}
};
void demarrerTabMotorises(Motorise * tab[], int n) {
   for (int i=0; i<n ; i++) { tab[i]-> demarrer(); }
}
int main() {
   Motorise * tabMotorise[2];
   tabMotorise[0] = new Voiture();
   tabMotorise[1] = new Perceuse ();
   demarrerTabMotorises(tabMotorise, 2);
   for (int i=0; i<2 ; i++) { delete tabMotorise[i]; }
}
```

HERITAGE ET CYCLE DE VIE : DESTRUCTEUR VIRTUEL

➢ Exemple problématique

```
class Base {
public:
    Base() { cout<<"Constructeur Base "<<endl; }
    ~Base( ){ cout<<"Destructeur Base"<<endl; }

};

class Derivee : public Base {
public:
    Derivee() { cout<<"Constructeur Derivee "<<endl;}
    ~Derivee() { cout<<"Destructeur Derivee "<<endl;}
};

int main() {
    Base * b = new Derivee; // OK !
    delete b;      // delete d'un objet Derivee
                   // au moyen d'un pointeur de type Base!
}
```

Que se passe-t-il ? Ou est le problème ?

➢ Destructeur virtuel

Conseil : le destructeur d'une classe de base (dont derivent d'autre classes) doit avoir un destructeur *virtuel*.

```
class Base {
public:
    Base() { cout<<"Constructeur Base "<<endl; }
    virtual ~Base( ){ cout<<"Destructeur Base"<<endl; }

};
```

Le programme precedent affiche alors :

```
Constructeur Base
Constructeur Derivee
Destructeur Derivee
Destructeur Base
```

HERITAGE ET CYCLE DE VIE : COPIE ET AFFECTATION

➢ Constructeur par copie et opérateur d'affectation

Imposer et respecter l'ordre hierarchique de construction :

- le constructeur par copie de la classe dérivée **doit** appeler le contructeur par copie de la classe mère.
- l'opérateur d'affectation de la classe dérivée **doit** appeler l'opérateur d'affectation de la classe mère.

➢ Exemple

```
class A {
public:
   A() { cout <<"A::A ; " ;}
   A(const A & other) { cout <<"A::A(copy) ; " ;}
   A& operator=(const A& other) {
        cout <<"A::operator= ; ";
        return *this ;
}
   virtual ~A() { cout <<"A::~A " <<endl ;}
};
class B : public A{
public:
   B() : A() { cout <<"B::B" <<endl;}
   B(const B & other): A(other) {cout <<"B::B(copy)"<<endl;}
   B& operator=(const B& other) {
        A::operator=(other);
        cout <<"B::operator=" <<endl;
        return *this
   }
   ~B() { cout << "B::~B ; "  ;}
};
```

```
main() {
   B b;
   B b2(b) ;
   B b3;
   b3 = b;
 }
```

```
A::A ; B::B
A::A(copy) ; B::B(copy)
A::A ; B::B
A::operator= ;
B::operator=
B::~B ; A::~A
B::~B ; A::~A
B::~B ; A::~A
```

HERITAGE ET CYCLE DE VIE : POLYMORHISME ET METHODE CLONE()

➢ Le problème

Avec les classes A et B precedents, polymorphisme **et** copies d'objets…

```
main() {
    B b;
    A * pa = &b;//Polymorphisme: on manipule B comme un A
    B b5(*pa);  //ERREUR ! Pas de B::B(const A &) !
}
```

➢ Méthode vritual clone()

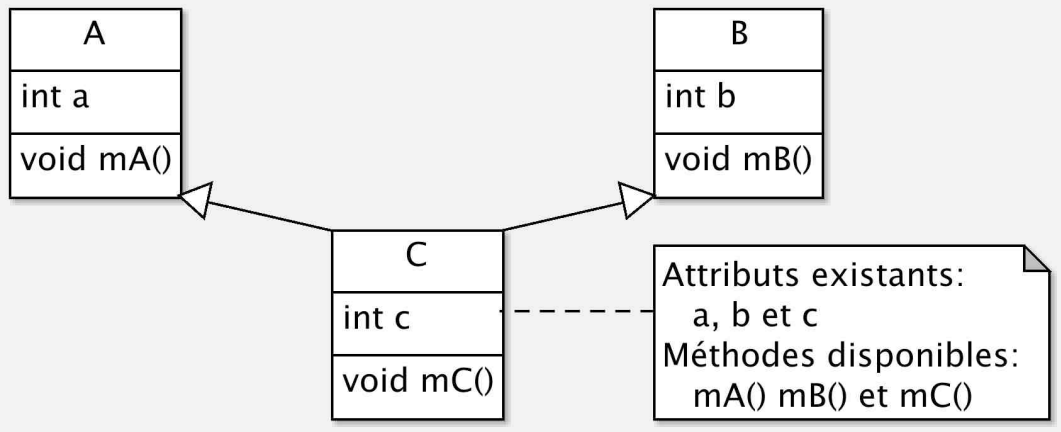
Dans les hiérarchie de classe, pouvoir faire des copies d'objets polymorphes, on définit souvent une méthode clone() comme suit :

```
class FIGURE {
public:
    virtual FIGURE * clone() const = 0 ;
};

class CERCLE : public FIGURE {
public:
    virtual FIGURE * clone() const {
          return new CERCLE(*this);
    }

};

main() {
    FIGURE * pf = new CERCLE() ;
    FIGURE * pf2 = pf ->clone();
    // le type dynamique de *pf2 est bien CERCLE  !
    // combien y a t il d'objets CERCLE en mémoire ?
}
```

HERITAGE MULTIPLE

Un concept peut avoir des points communs avec plusieurs autres concepts : il doit alors hériter de ces divers concepts : **héritage multiple**.

L'héritage multiple concerne aussi bien les champs que les méthodes.



➢ Syntaxe :

```
class derivee :
     <public,private> base1, <public,private> base2
     { •••
};
```

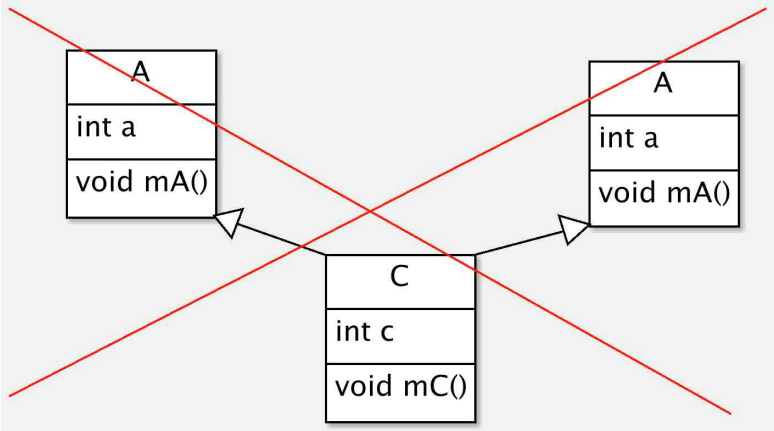
Attention : l'héritage est statique : il ne doit pas y avoir d'ambiguïté sur les méthodes. Les classes A et B ne doivent pas avoir une méthode de nom identique; Dans le cas contraire, il faut spécifier complètement le nom de fonction lors de l'appel.

➢ Exemple

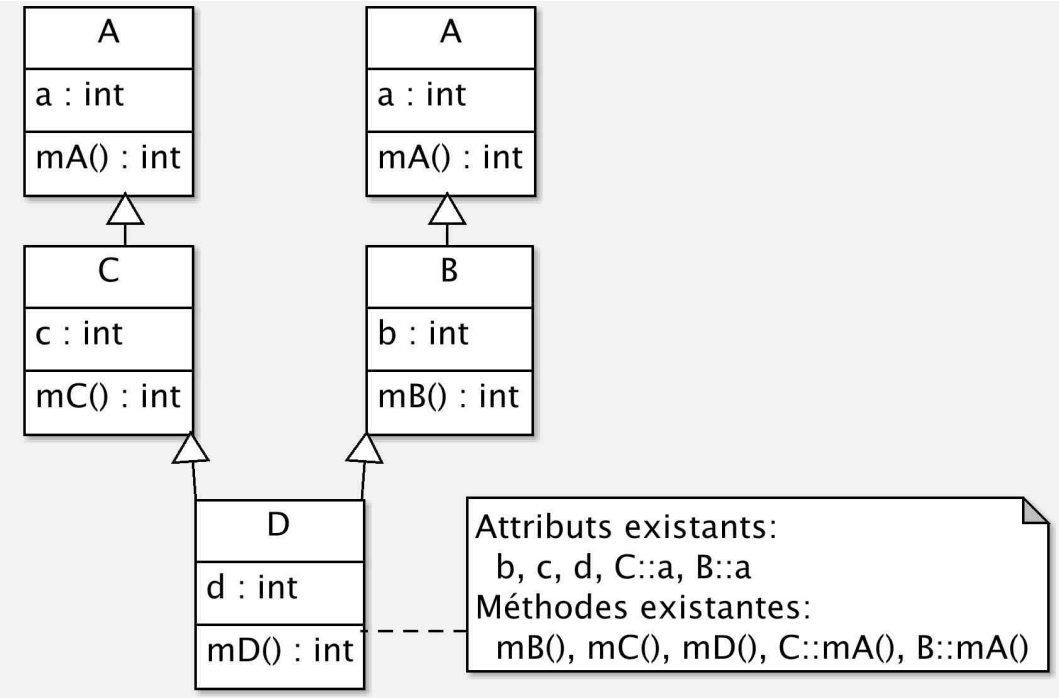
```
class A { public : void m1() { •••} ; };
class B { public : void m1() { •••} ; };

class C : public A, public B { •••};

main() {  C x;
    x.m1();          // ERREUR : ambiguïté
    x.A::m1();       // OK
    x.B::m1();       //OK
}
```

HERITAGE MULTIPLE D'UNE MEME CLASSE

L'héritage multiple d'une même classe est interdit.



L'héritage **indirect** multiple d'une même classe est autorisé. La classe de base est alors dupliquée 2 fois.




```
main() {
     C x;
     x.c = x.d= x.b = 0;    // OK
     x.a=0;                 //ERREUR : ambiguïté
     x.D::a = x.B::a = 0;   // OK
}
```

HERITAGE MULTIPLE ET PARTAGE. HERITAGE VIRTUEL

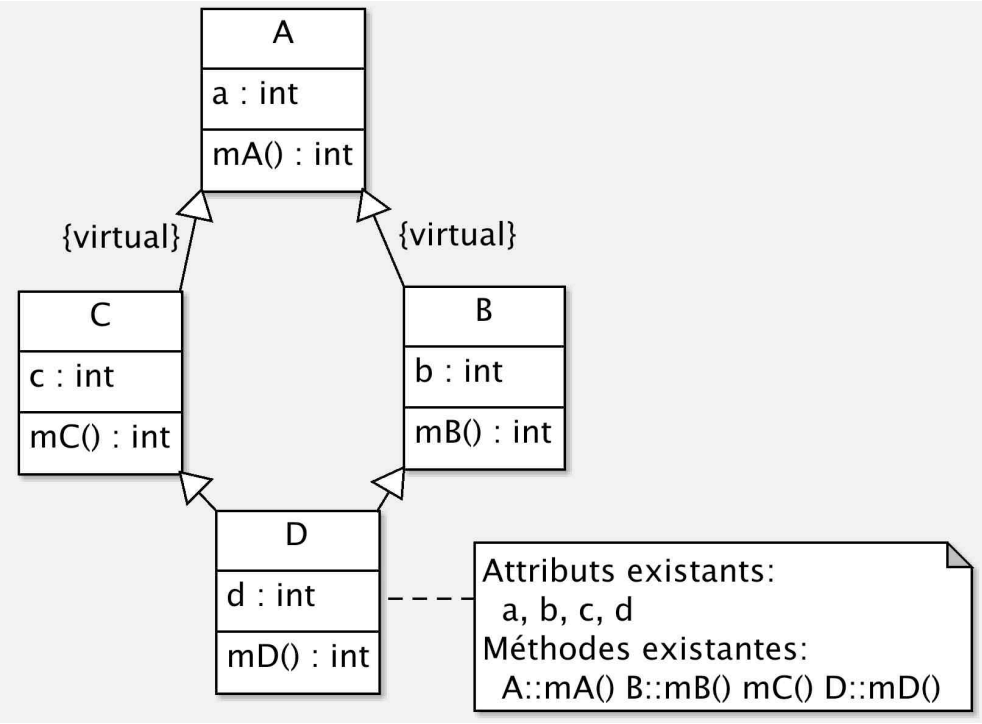
L'héritage multiple peut être vu comme une manière de rassembler des classes pour en créer de nouvelles plus complexes, pour intégrer des outils développés séparément.

Autre situation : développer des classes fortement liées entre elles, partageant des propriétés => héritage indirect d'une même classe de base, sans duplication de l'objet de base (classe iostream qui dérive de istream et ostream)

Pour ce cas : **héritage *virtuel***

➢ Syntaxe

```
class classe_dérivée : public virtual classe_base {•••};
                     : private
```



➢ Remarques

Constructeur : le constructeur de C doit passer les arguments aux constructeurs de D et B, mais aussi aux constructeurs de la classe de base virtuelle A.

Le constructeur de la classe de base virtuelle est toujours appelé avant les autres

HERITAGE MULTIPLE ET PARTAGE. HERITAGE VIRTUEL

➢ Exemple

```
class FENETRE { public : int X; int Y; •••};

class MENU : public virtual FENETRE  {
public:
    int nb_lignes;
    MENU (..) { •••}    // Constructeur
    •••
};

class F_COUL : public virtual FENETRE  {
public:
    int couleur;
    F_COUL (..) {•••} // Constructeur
    •••
};

class F_MENU_COUL :  public MENU, public F_COUL  {
public :
    F_MENU_COUL (...):
      FENETRE (...) {
      F_COUL (...),
      MENU (...),
           •••
    }
    •••
} ;

};
```