

ISIMA Héritage

- Un des fondements de la POO
- L'héritage est à la base des possibilités de réutilisation des composants logiciels (en l'occurrence, des classes)
- Concept naturel de Généralisation / Spécialisation
 - classes représentées sous forme d'arbres généalogiques
- Idée fondamentale
 - classe B dérivant de classe A
 - B hérite de tous les attributs et méthodes de A
 - B ajoute ses propres attributs et méthodes

50

ISIMA Mise en œuvre de l'héritage simple en C++

- Dériver une classe d'une autre
- Syntaxe

```
class dérivée : modificateur d'accès
                {, modificateur d'accès}^n
```
- Modificateur conditionne la visibilité des attributs et méthodes de la classe mère.
- Le mot **public** signifie que les membres publics d'une classe base seront aussi des membres publics de la classe dérivée.

51

ISIMA Mise en œuvre de l'héritage simple en C++

```
class Point
{
    private:
        int absc;
        int ordo;
    public:
        Point(int x=0, int y=0);
        void move(int, int);
        void affiche();
};
```

```
...
Pixel p(10,20,5);
p.affiche();
...
```


Présence de lacunes puisque affiche() ne donne pas la couleur du pixel

```
class Pixel : public Point
{
    private:
        short couleur;
    public:
        Pixel(int x=0, int y=0,
            int c=1) : Point(x,y)
        void colore(short c)
        {
            couleur = c;
        }
};
```

52

ISIMA Mise en œuvre de l'héritage simple en C++

- Modificateur conditionne la visibilité des attributs et méthodes de la classe mère

Classe mère	Classe fille	
	héritage public	
public	public	
protected	protected	
private	inaccessible	

53

ISIMA Constructeur, destructeurs et l'héritage

- Pour créer un objet de type B, il faut d'abord créer un objet de type A puis ajouter les éléments spécifiques à B
- Pour la destruction d'un objet de type B, il faut d'abord détruire les éléments spécifiques à B puis A

```
class A
{
    ...
    public :
        A(...) {...}
        ~A() {...}
    ...
};
```

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

54

ISIMA Constructeur, destructeurs et l'héritage

```
main ()
{
    Pixel a(10,15,3);
    Pixel b(2,3);
    Pixel c(12);
    Pixel * d;
    d = new Pixel(12,25);
    delete d;
}
```

Création du point : 10 15
Création du pixel : 10 15 3
Création du point : 2 3
Création du pixel : 2 3 1
Création du point : 12 0
Création du pixel : 12 0 1
Création du point : 12 25
Création du pixel : 12 25 1
Destruction du pixel : 1
Destruction du point : 12 25
Destruction du pixel : 1
Destruction du point : 12 0
Destruction du pixel : 1
Destruction du point : 2 3
Destruction du pixel : 3
Destruction du point : 10 15

55

ISIMA Mise en œuvre de l'héritage simple en C++

- Modificateur d'accès **protected**
 - visible de classe fille mais non de l'extérieur
- Utilisation classique de l'héritage
 - attributs **protected** + héritage **public**
- Passer les attributs **private** en **protected** ?
 - 👉 resteront accessibles directement en cas de dérivation
 - 👉 si nécessité de mises à jour d'attribut en cascade ou critiques, utiliser des méthodes de la classe mère

56

ISIMA Mise en œuvre de l'héritage simple en C++

```
class Point
{
    protected:
        int absc;
        int ordo;

    public:
        Point(int x=0, int y=0);
        void move(int,int);
        void affiche();
};
```

```
...
Pixel p(10,20,1);
p.affiche();
...
```

```
class Pixel : public Point
{
    private:
        short couleur;
    public:
        Pixel(int x=0, int y=0,
            int c=1) : Point (x,y)
        void affiche()
        {
            cout << "Point :" <<
                abs
                << "\t" << ord << endl;
            cout << "couleur"
                << couleur << endl;
        }
};
```

57

ISIMA Intérêt du statut protégé (**protected**)

- Les membres privés d'une classe sont inaccessibles depuis l'extérieur de la classe (objets de cette classe, fonctions membres d'une classe dérivée, objets d'une classe dérivée, ...)
- Les membres protégés se présentent comme des membres privés pour l'utilisateur de la classe, ...
- ...mais ils sont comparables à des membres publics pour le concepteur d'une classe dérivée.

58

ISIMA Dérivation (héritage) privée « **private** »

- On interdit à l'utilisateur d'une classe dérivée l'accès aux membres publics de sa classe mère.

```
class Point
{
    ...
    public:
        Point(int x=0, int y=0);
        void move(int,int);
        void affiche();
};
```

```
class Pixel : private Point
{
    ...
    public:
        Pixel(int x=0, int y=0,
            int c=1) : Point (x,y)
        ...
};
```

```
...
Pixel p(10,20,1);
p.affiche();    // rejetés par le compilateur
...
```

59

ISIMA Dérivation (héritage) privée « **private** »

- Limite l'intérêt de l'héritage ?
- Sera employée dans des cas bien précis :
 - lorsque toutes les fonctions utiles de la classe de base sont redéfinies dans la classe dérivée.
 - lorsque l'on souhaite adapter l'interface d'une classe de manière à répondre à certaines exigences.
 - par exemple, la classe dérivée n'a pas de nouvelles données, ni fonctionnalités ; elle agit comme la classe mère, mais son utilisation est différente.

60

ISIMA Mise en œuvre de l'héritage simple en C++

- Modificateur conditionne la visibilité des attributs et méthodes de la classe mère

Classe mère	Classe fille	
	héritage public	héritage private
public	public	private
protected	protected	private
private	inaccessible	inaccessible

61

ISIMA Compatibilité entre classe de base et dérivée

- Un objet d'une classe dérivée « **peut** » être utilisé à la place d'un objet d'une classe de base.
- L'idée repose sur le fait que ce qu'on trouve dans une classe de base se trouve dans une classe dérivée.
- La réciproque de ces deux propositions est fausse.
- Cette compatibilité ne s'applique que dans le cas de dérivation « **public** »

62

ISIMA Conversion d'un type dérivé en type de base

```
class Point
{
    ...
};
```

```
class Pixel : public Point
{
    ...
};
```

```
Point a;
```

```
Pixel b;
```

```
a = b;           // OK (upcast)
```

```
b = a;           // échec (downcast)
```

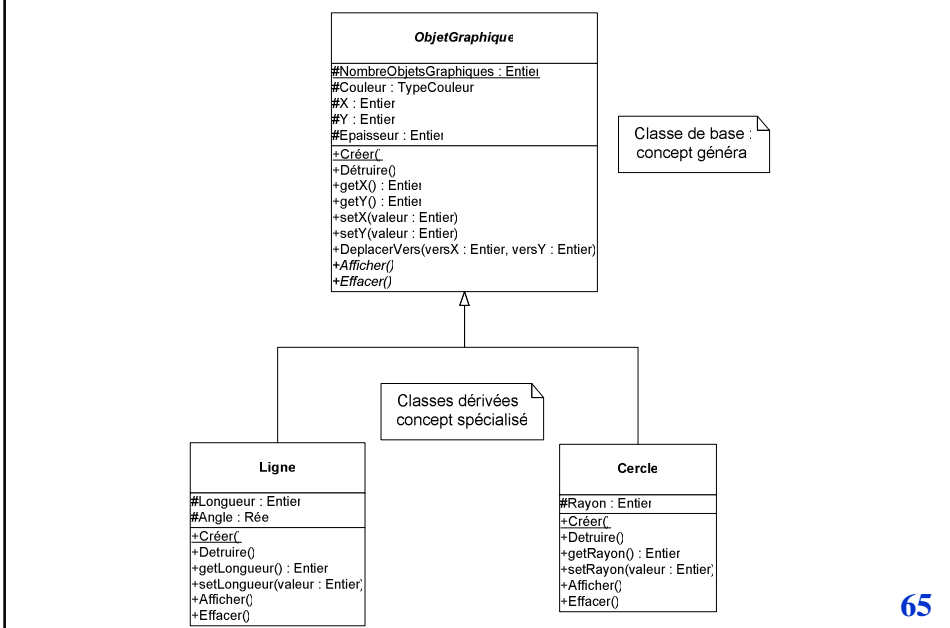
63

ISIMA Classes abstraites

- Rendre une classe abstraite
 - définir (au moins) une méthode virtuelle pure
`virtual type_retour methode(paramètres) = 0;`
 - une méthode virtuelle pure a sa définition nulle (0), et non plus seulement vide ({ }).
- à redéfinir impérativement dans les sous-classes
 - impossible d'instancier la classe abstraite
 - les sous-classes restent abstraites tant que la méthode ne reçoit pas une déclaration
 - possibilité de fournir du code en même temps ! D'autres méthodes dans la classe...cf. plus loin.

64

ISIMA Exemple de classe abstraite



65

ISIMA Exemple de classe abstraite

```
class objetGraphique {
    protected :
        static int nbGraphiques;
        int couleur;
        int x;
        int y;
        int epaisseur;

    public :
        ...
        virtual void afficher( ) const = 0;

        virtual void effacer( ) const = 0;

};
```

66

ISIMA Dans la classe dérivée

```
class ligne : public objetGraphique
{
    protected :
        ...

    public :
        ...
        void afficher( ) const
        {
            cout << "--Objet Graphique ligne--" << endl;
            cout << "coordonnees :" << x << "\t" << y << endl;
            cout << "longueur : " << longueur << endl;
            cout << "angle : " << angle << endl;
        }

        void effacer( ) const
        {
            cout << "J'efface la ligne" << endl;
        }

};
```

67

ISIMA Héritage et polymorphisme (rappel)

- une classe « est » un type
- deux catégories de langages à objets
 - typage fort (et statique) [C++, Java]
 - un objet est affecté à une classe au moment de la compilation
 - vérification de l'adéquation objet/message à la compilation
 - impossible d'envoyer un message non traitable
 - facile à déboguer et à mettre au point
 - le type est liée au nom (identificateur) de l'objet, donc à son adresse mémoire
 - typage faible (et dynamique) [Smalltalk, Objective C]
 - le type est lié au contenu de l'objet, pas à l'adresse
 - peut modifier le typage de l'objet au cours du temps
 - grande souplesse, notamment lors du prototypage d'une application
 - vérification à l'exécution de l'adéquation objet/message
 - erreur
 - transmission ou délégation vers un autre objet

68

ISIMA Héritage et polymorphisme (rappel)

```
class Point
{
    ...
    void affiche();
};
```

```
class Pixel : public Point
{
    ...
    void affiche();
};
```

```
Point a;
Pixel b;
Point * ptr = &a;
ptr->affiche(); /* appelle affiche() (classe Point) */
ptr = &b;      // autorisé
ptr->affiche();

/* appelle affiche() de la classe Point, alors que la classe
Pixel dispose aussi d'une méthode affiche */
```

69

ISIMA Héritage et polymorphisme

- Rendre une méthode polymorphe : **virtual**
 - indique au compilateur que les éventuels appels aux méthodes virtuelles doivent utiliser un typage dynamique (adéquation à l'exécution).
 - pas nécessaire de déclarer virtuelle dans les classes dérivées, une méthode déclarée virtuelle dans une classe de base.
 - à redéfinir dans les sous-classes.
 - destructeur doit devenir virtuel (destruction polymorphe).
 - réutilisation de l'implémentation de la classe mère dans la classe dérivée

`classe_mère::méthode(paramètres)`

70

ISIMA Héritage et polymorphisme

```
class Point
{
    ...
    virtual void affiche();
};
```

```
class Pixel : public Point
{
    ...
    void affiche();
};
```

71

ISIMA Polymorphisme : exemple 1

```
//méthode affiche() virtuelle dans la classe Point

main ()
{
    Point a(3,5);
    Point * ptr_point = &a;
    Pixel b(8,6,2);
    Pixel * ptr_pixel = &b;
    ptr_point -> affiche(); //méthode affiche de Point
    ptr_pixel -> affiche(); //méthode affiche de Pixel
    ptr_point = ptr_pixel; //autorisé
    ptr_point -> affiche(); //méthode affiche de Pixel
    ptr_pixel -> affiche(); //méthode affiche de Pixel
}
```

72

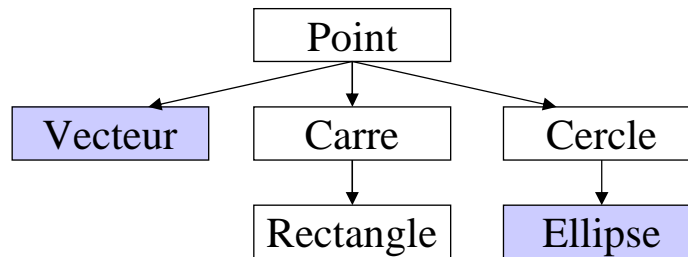
ISIMA Les propriétés des méthodes virtuelles

1. La redéfinition d'une méthode virtuelle n'est pas obligatoire.
2. Une méthode virtuelle peut être surdéfinie et peut être déclarée virtuelle ou non.
3. La redéfinition d'une méthode virtuelle doit respecter le type de la valeur de retour (une exception cf. plus loin).
4. Une méthode virtuelle peut être déclarée dans n'importe quel classe.

73

ISIMA Polymorphisme : exemple 2

- On suppose qu'affiche est redéfini dans les classes Vecteur et Ellipse. L'appel de la méthode affiche pour ces classes...



Vecteur Vecteur::affiche(...)
Carre Point::affiche(...)
...

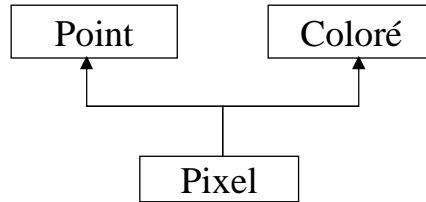
74

ISIMA Héritage multiple

- Son usage reste assez peu répandu
- Comment exprimer une dépendance «multiple» ?
- Comment sont appelés les constructeurs et les destructeurs concernés ?
- Comment régler les conflits, par exemple, dans un héritage en diamant?

75

ISIMA Mise en œuvre de l'héritage multiple



```
class Point
{
    protected :
        int x, y;
    public :
        Point(...) {...}
        ~Point() {...}
        void affiche() {...}
};
```

```
class Colore
{
    protected :
        short couleur;
    public :
        Colore(...) {...}
        ~Colore() {...}
        void affiche() {...}
};
```

76

ISIMA Mise en œuvre de l'héritage multiple

```
class Pixel : public Point, public Colore
{
    ...
    public :
        Pixel (int, int, int);
        void affiche();
};
```

```
Pixel::Pixel(int abs, int ord, int c) :
Point(abs, ord), Colore(c)
{
    ...
};
```

77

ISIMA Constructeur, destructeurs et l'héritage multiple

```
main ()
{
    Pixel a(3,9,2);
    a.affiche();
    a.Point::affiche();
    a.Colore::affiche();
}
```

Début de main

Appel constructeur de Point : 3 9

Appel constructeur de Colore : 2

Appel constructeur de Pixel : 3 9 2

Appel méthode affiche de Pixel

Appel méthode affiche de Point

Appel méthode affiche de Colore

Fin de main

Appel destructeur de Pixel

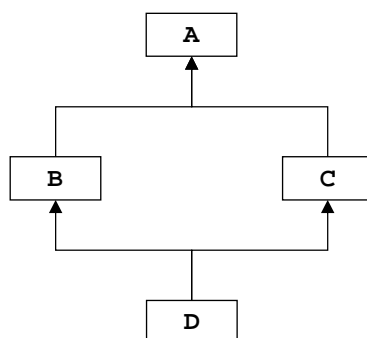
Appel destructeur de Colore

Appel destructeur de Point

78

ISIMA Conflits de l'Héritage multiple

- Héritage en diamant



- Collisions

- attributs de A dupliqués dans D
- comment appeler le constructeur de A dans D ?
- comment hériter proprement des méthodes ?

79

ISIMA Conflits de l'Héritage multiple

```
class B : public A {...};  
class C : public A {...};  
class D : public B, public C {...};
```

Appel du constructeur pour un objet du type **D** dans cette ordre :

A1 B A2 C D

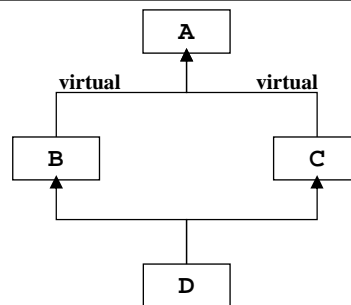
Duplication des données de la classe **A** en **D**. Situation pas souhaitable!

80

ISIMA Héritage virtuel

- Solution : héritage virtuel

```
class B : virtual public A  
{ B(...) : A(...) {} };  
class C : virtual public A  
{ C(...) : A(...) {} };  
class D : public B, public C  
{ D(...) : A(...), B(...), C(...) {} };
```



- spécifié A comme virtuelle dans la déclaration de B, signifie que A sera introduite une seul fois dans les **descendants** éventuels de B
- une seule copie de A dans D
- appel explicite au constructeur de A
- Autre solution : les interfaces...

81

ISIMA Relations fondamentales

- 3 grandes relations fondamentales
 - héritage : généralisation/spécialisation
symbolisé par « est une version spécialisée de » (Is A)
 - agrégation / composition
symbolisé par « contient », « regroupe » (Has A)
 - association : communication
symbolisé par « communique avec » (Uses A)

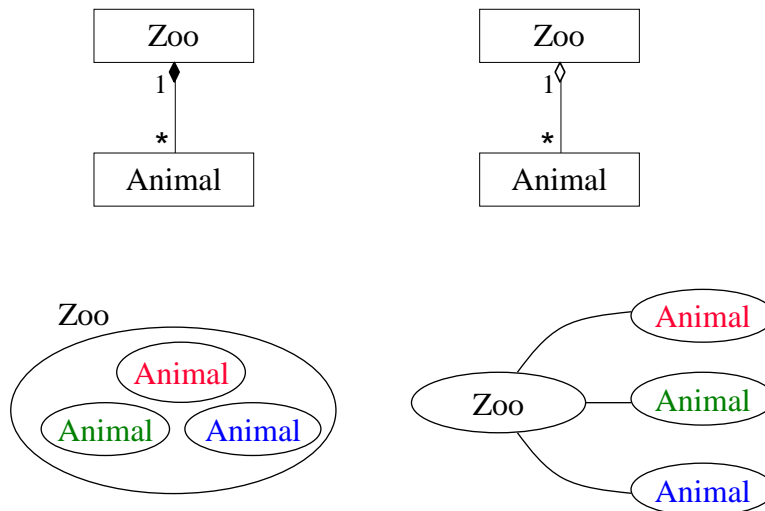
82

ISIMA Agrégation vs Composition

- L'agrégation et la composition sont deux versants d'un même type de relation « contient », « regroupe » (Has A)
- Dans la composition, la destruction du agrégateur entraîne celle des parties
- Dans l'agrégation, les composants sont autonomes

83

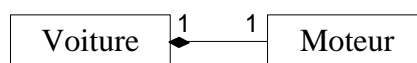
ISIMA Composition vs Agrégation



84

ISIMA Composition : exemple 1

- Définir dans la classe agrégateur, des attributs qui correspondent à des objets agrégés.

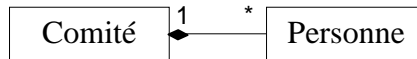


```
class Voiture
{
    protected :
        string immatriculation;
        Moteur m;
    public :
        Voiture(const Moteur & m1): m(m1)
        {...}
        ...
};
```

```
class Moteur
{
    protected :
        string type;
        int puissance;
    public :
        ...
};
```

85

ISIMA Composition : exemple 2



```

class Comite
{
    protected :
        Personne president;
        Personne secretaire;
        Personne tresorier;
    public :
        Comite(const Personne
        & p, const Personne &
        s, const Personne &
        t) :
            ...
};
  
```

```

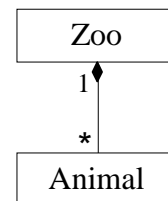
class Personne
{
    protected :
        string nom;
        ...
    public :
        Personne(string n): nom(n)
        {...}
        ...
};
  
```

86

ISIMA Composition : exemple 3

```

class Zoo
{
    protected :
        string nom;
        int code_postal;
        int taille;
        Animal * liste_a;
    public :
        Zoo(string n, int code, int t):
            nom(n), code_postal(code),
            taille(t){
                liste_a = new Animal[taille];
                ...
            }
        ~Zoo{
            ...
            delete [] liste_a;
        }
        ...
};
  
```



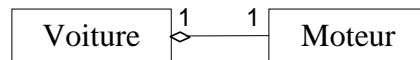
```

class Animal
{
    protected :
        string nom;
        int id;
        ...
};
  
```

87

ISIMA Agrégation : exemple 1

- Définir des pointeurs sur des classes agrégées

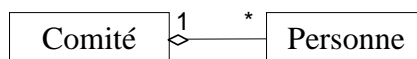


```
class Voiture
{
    protected :
        string immatriculation;
        Moteur * m;
    public :
        Voiture(Moteur * m1){
            m = m1;
        }
        ~Voiture(){
            ...
        }
};
```

```
class Moteur
{
    protected :
        string type;
        int puissance;
    public :
        Moteur(string t, int p)
        : type(t),
          puissance(p){}
        ...
};
```

88

ISIMA Agrégation : exemple 2



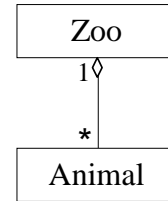
```
class Comite
{
    protected :
        Personne *president;
        Personne *secretaire;
        Personne *tresorier;
    public :
        Comite(Personne*,
              Personne*, Personne*)
        ...
};
```

```
class Personne
{
    protected :
        string nom;
        ...
    public :
        Personne(string n): nom(n)
        {...}
        ...
};
```

89

ISIMA Agrégation : exemple 3

```
class Zoo
{
    protected :
        string  nom;
        int     code_postal;
        int     taille;
        Animal  ** liste_a;
    public :
        Zoo(string n, int code, int t):
            nom(n), code_postal(code),
            taille(t){
            liste_a = new (Animal *)[taille];
            ...
        }
        ~Zoo() { delete [] liste_a; }
};
```



```
class Animal
{
    protected :
        string  nom;
        int     id;
    public :
        Animal(string,int);
        ...
};
```

90

ISIMA Agrégation vs composition

- Regrouper un ou plusieurs objets dans un autre
- Trois types d'agrégation/composition
 - objet direct
 - référence (attention à l'initialisation)
 - pointeur
- Vie de l'objet agrégé
 - objet construit par l'agrégeant
 - objet en provenance de l'extérieur
 - recopié
 - prise de pointeur ou de référence

91

ISIMA La classe Point

```
#ifndef __POINT_H__
#define __POINT_H__

class Point
{
private:
    int x;
    int y;

public:
    Point(int a=0, int b=0) :
        x(a), y(b) {}

    int getX(void) const
    { return x; }

    int getY(void) const
    { return y; }

    void move(int, int);

    void moveTo(int, int);
};

#endif
```

```
#include "point.h"

void Point::moveTo(int a, int b)
{
    x = a;
    y = b;
}

void Point::move(int a, int b)
{
    moveTo(x + a, y + b);
}
```

92

ISIMA La classe ObjetGraphique

```
#ifndef __OBJETGRAPHIQUE_H__
#define __OBJETGRAPHIQUE_H__

#include "point.h"

class ObjetGraphique
{
public:
    enum { COULEURFOND = 0 };

protected:
    Point base;
    int couleur;
    int epaisseur;

public:
    ObjetGraphique(int x, int y, int c = 0, int e = 0) :
        base(x,y), couleur(c), epaisseur(e)
    {}
    ... // autres méthodes
    virtual void afficher(void) const = 0;
};

#endif
```

93

ISIMA La classe **ObjetGraphique**

```
const Point & ObjetGraphique::getBase(void) const
{
    return base;
}

int ObjetGraphique::getCouleur(void) const
{
    return couleur;
}

int ObjetGraphique::getEpaisseur(void) const
{
    return epaisseur;
}

virtual void ObjetGraphique::effacer(void)
{
    int tmp = couleur;
    couleur = COULEURFOND;
    afficher();
    couleur = tmp;
}

void moveTo(int x, int y)
{
    effacer();
    base.moveTo(x, y);
    afficher();
}
```

94

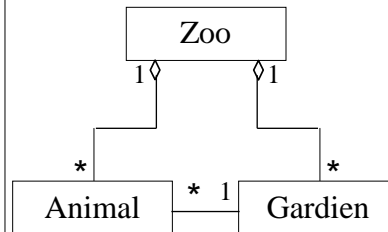
ISIMA Association

- L'association est une relation d'utilisation des services d'un objet par un autre.
- « uses a », est en association avec, communique avec
- L'association n'est pas transitive.

95

ISIMA Association

```
class Zoo
{
    protected :
        ...
        Animal ** liste_a;
        Gardien ** liste_g;
    public :
        Zoo(string n, int code, int t):
            nom(n), code_postal(code),
            taille(t)
        {
            liste_a = new (Animal *)[taille];
            liste_g = new (Gardien *)[taille];
            ...
        }
        ~Zoo() { delete [] liste_a; ... }
};
```



Nourrit <

96

ISIMA Association

```
class Gardien
{
    protected :
        int carte_id;
        int registre;
    public :
        Gardien(){...}
        Nourrir(Animal & a){
            inserer_nourriture();
            a.avancer();
            a.prendre_nourriture();
            a.manger();
            ...
        }
};
```

```
class Animal
{
    protected :
        string nom;
        int id;
    public :
        Animal(){...}
        avancer(){...}
        prendre_nourriture(){...}
        manger(){...}
        ...
};
```

97

ISIMA Récapitulatif

```
class A {  
    ...  
    public :  
        ...  
        manipuler(B & b){  
            b.move();  
            b.afficher();  
        }  
};  
class B { ...  
    public :  
        move();  
        afficher();  
};
```

```
class A {...};  
class B {...};  
class C : public A, public B {...};
```

```
class A {  
    protected :  
        B var;  
    ...  
};  
class B {...};
```

```
class A {  
    protected :  
        B * ptr;  
    ...  
};  
class B {...};
```

```
class A {...};  
class B : public A {...};
```