ISEN
ALL IS DIGITAL!
OUEST

Programmation orientée objet en C++
- Composition de classes -

Groupe : CIR2

Nils Beaussé
E-Mail : nils.beausse@isen-ouest.yncrea.fr
Numéro de bureau : A2-78 ou A1-50 (Salle robotique/IA)

Introduction à UML

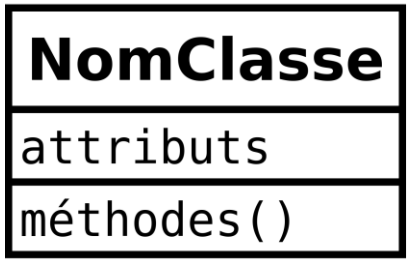
Définition

- UML = Unified Modeling Language
- L'UML est un standard défini par OMG (Object Management Group)
- L'UML est utilisé pour spécifier, visualiser, modifier et construire les documents nécessaires
  au bon développement d'un logiciel orienté objet
- L'UML est un langage graphique composé de plusieurs diagrammes

ISEN
ALL IS DIGITAL!
OUEST

Diagrammes UML

- **Statique** : Définition de la structure statique du système : relation entre les objets
  - Exemple : **Diagramme de classe**, Diagramme des cas d'utilisation

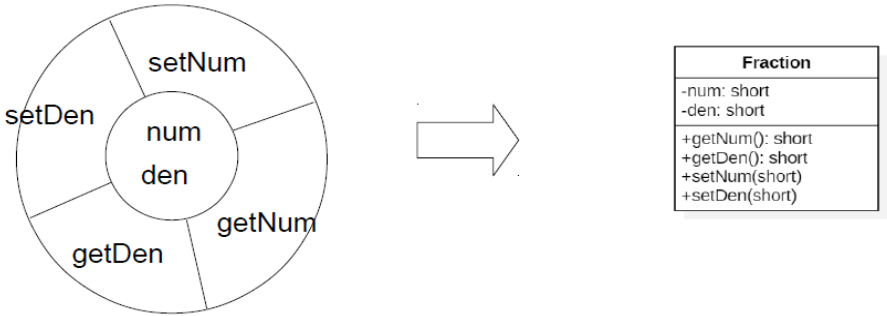


- **Dynamique** : Définition du comportement dynamique d'un système : changement de l'état interne des objets

ISEN
ALL IS DIGITAL!
OUEST

Diagramme de classe

- Représente les classes qui modélisent le système et les différentes relations entre elles
- La relation entre les classes est appelée association
- La **composition** et l'**agrégation** sont deux types d'association
- Certains logiciels IDE permettent de générer automatiquement le code source correspondant au diagramme de classes
- Exemple du diagramme de classe :

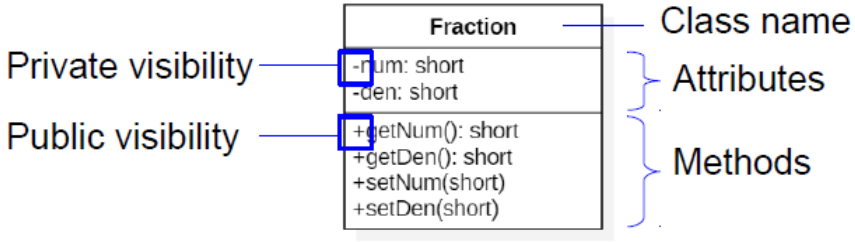


ISEN
ALL IS DIGITAL!
OUEST

Diagramme de classe

- Représente les classes qui modélisent le système et les différentes relations entre elles
- La relation entre les classes est appelée association
- La **composition** et l'**agrégation** sont deux types d'association
- Certains logiciels IDE permettent de générer automatiquement le code source correspondant au diagramme de classes

- Exemple du diagramme de classe :

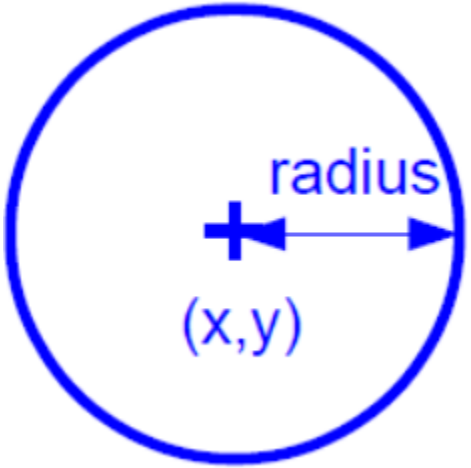


Composition

ISEN
ALL IS DIGITAL!
OUEST

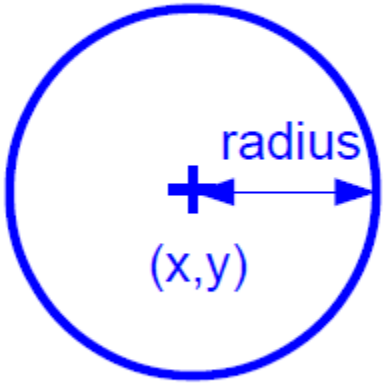
Modélisation d'un cercle

- **Première solution**

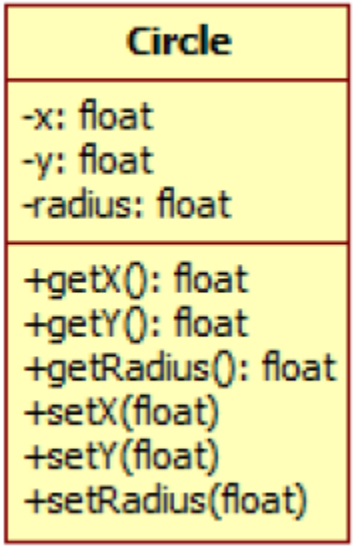


Modélisation d'un cercle

- **Première solution**

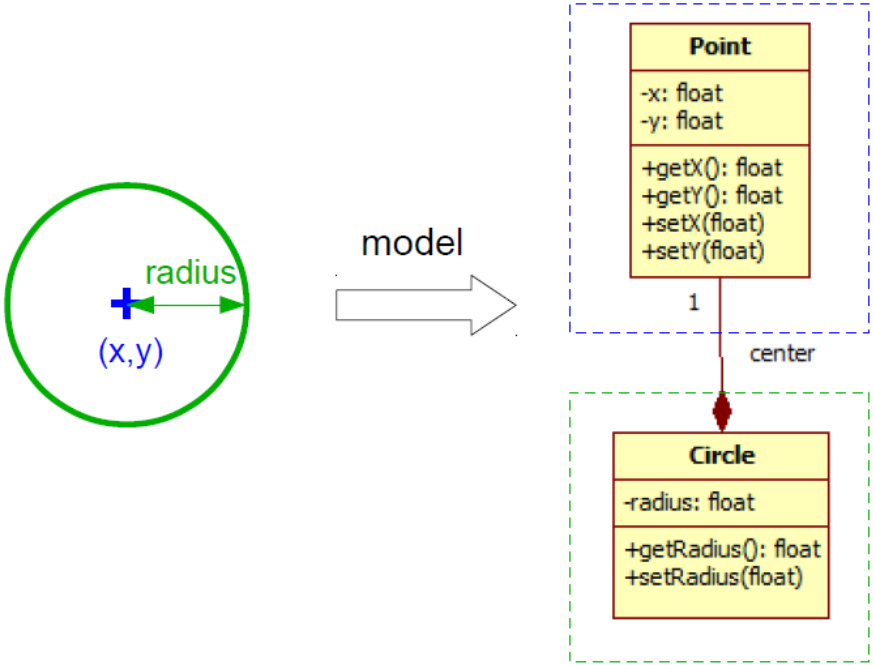


model



Modélisation d'un cercle

- **Première solution**

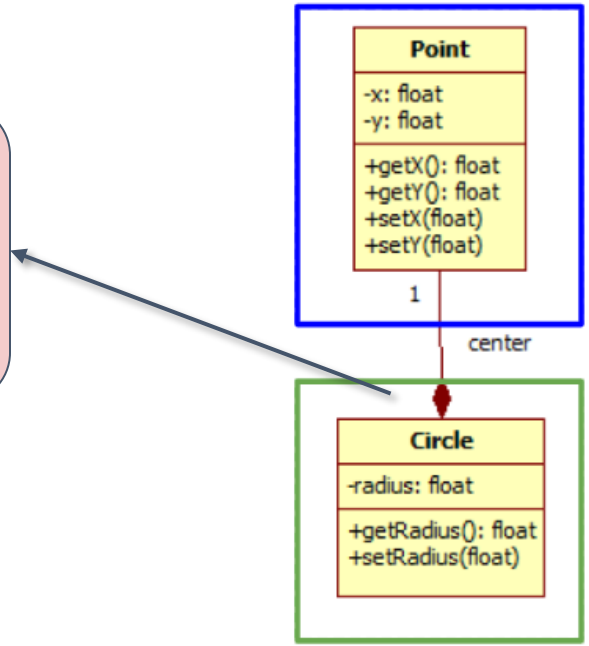


ISEN
ALL IS DIGITAL!
OUEST

Modélisation d'un cercle

- **Deuxième solution**

Composition = une classe a comme attribut l'objet d'une autre classe
Exemple : un Cercle encapsule un objet (center) de la classe Point



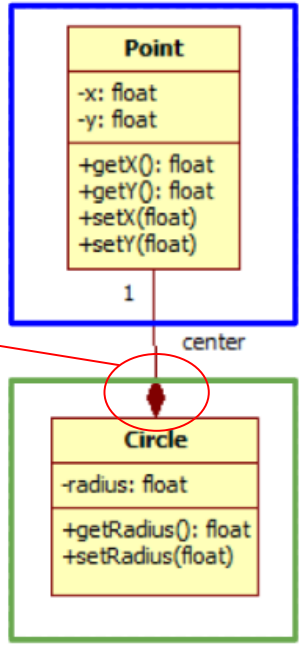
ISEN
ALL IS DIGITAL!
OUEST

Implémentation d'un cercle

- **Deuxième solution**

Circle.h

```
#include "Point.h"
class Circle{
private:
 Point center;
 float radius;
public:
 Circle();
 Circle(float);
 float getRadius() const;
 void setRadius(float);
};
```



Point.h

```
class Point{
private:
  float x;
  float y;
public:
  Point();
  Point(float, float);
  float getX() const;
  float getY() const;
  void setX(float);
  void setY(float);
};
```

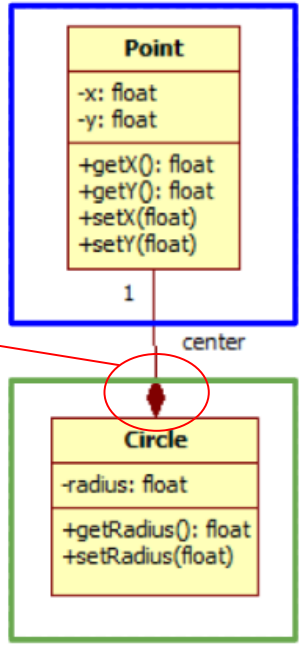
Implémentation d'un cercle

- **Deuxième solution**

Chaque instance de la classe Cercle aura une instance de la classe Point

Circle.h

```
#include "Point.h"
class Circle{
private:
 Point center;
 float radius;
public:
 Circle();
 Circle(float);
 float getRadius() const;
 void setRadius(float);
};
```



Point.h

```
class Point{
private:
  float x;
  float y;
public:
  Point();
  Point(float, float);
  float getX() const;
  float getY() const;
  void setX(float);
  void setY(float);
};
```

Implémentation d'un cercle

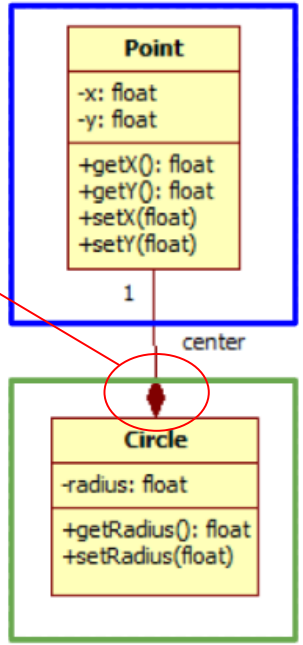
ISEN ALL IS DIGITAL! OUEST

Circle.h

```
#include "Point.h"
class Circle{
private:
 Point center;
…};
```

main.cpp

```
#include "Circle.h"
int main() {
  Circle c;
  return 0;
}
```



Circle.cpp

```
…
Circle::Circle(){
  cout << "Circle()" << endl;
  setRadius(0);
}
Circle::Circle(float radius){
  cout << "Circle(float)" << endl;
  setRadius(radius);
}
…
```

Point.cpp

```
…
Point::Point(){
  cout << "Point()" << endl;
  setX(0);
  setY(0);}
Point::Point(float x, float y){
  cout << "Point(float, float)" <<
endl;
  setX(x);
  setY(y);}
```

Implémentation d'un cercle

ISEN ALL IS DIGITAL! OUEST

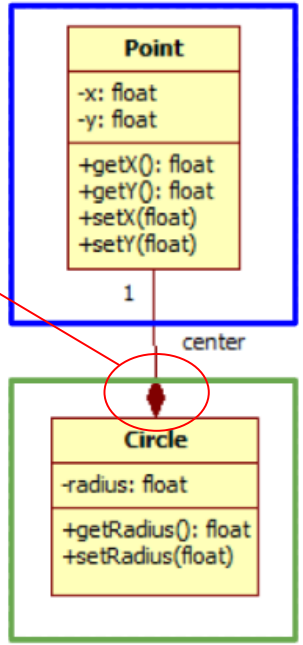
Circle.h

```
#include "Point.h"
class Circle{
private:
 Point center;
…};
```

main.cpp

```
#include "Circle.h"
int main() {
  Circle c;
  return 0;
}
```

```
Point()
Circle()
```



Circle.cpp

```
…
Circle::Circle(){
  cout << "Circle()" << endl;
  setRadius(0);
}
Circle::Circle(float radius){
  cout << "Circle(float)" << endl;
  setRadius(radius);
}
…
```

Point.cpp

```
…
Point::Point(){
  cout << "Point()" << endl;
  setX(0);
  setY(0);}
Point::Point(float x, float y){
  cout << "Point(float, float)" <<
endl;
  setX(x);
  setY(y);}
```

Implémentation d'un cercle

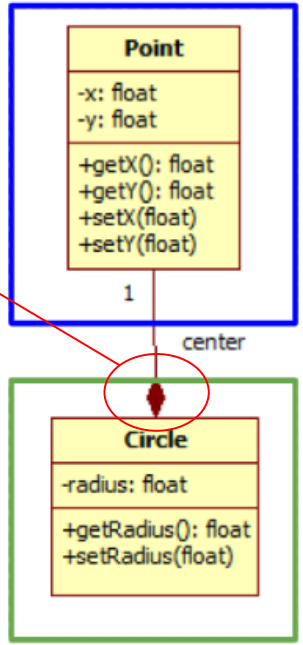
ISEN ALL IS DIGITAL! OUEST

Circle.h

```
#include "Point.h"
class Circle{
private:
 Point center;
…};
```

main.cpp

```
#include "Circle.h"
int main() {
  Circle c(1.0);
  return 0;
}
```



Circle.cpp

```
…
Circle::Circle(){
  cout << "Circle()" << endl;
  setRadius(0);
}
Circle::Circle(float radius){
  cout << "Circle(float)" << endl;
  setRadius(radius);
}
…
```

Point.cpp

```
…
Point::Point(){
  cout << "Point()" << endl;
  setX(0);
  setY(0);}
Point::Point(float x, float y){
  cout << "Point(float, float)" <<
endl;
  setX(x);
  setY(y);}
```

Implémentation d'un cercle

ISEN ALL IS DIGITAL! OUEST

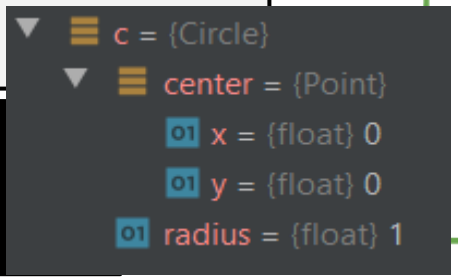
Circle.h

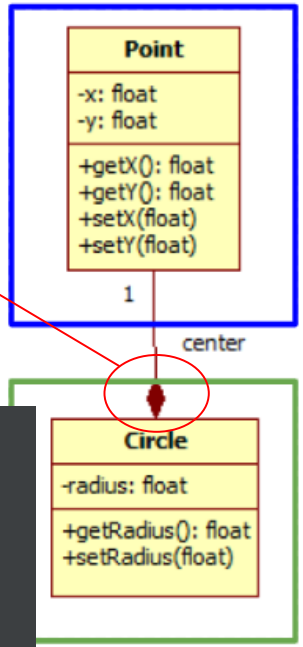
```
#include "Point.h"
class Circle{
private:
 Point center;
…};
```

main.cpp

```
#include "Circle.h"
int main() {
  Circle c(1.0);
  return 0;
}
```

```
Point()
Circle(float)
```





Circle.cpp

```
…
Circle::Circle(){
  cout << "Circle()" << endl;
  setRadius(0);
}
Circle::Circle(float radius){
  cout << "Circle(float)" << endl;
  setRadius(radius);
}
…
```

Point.cpp

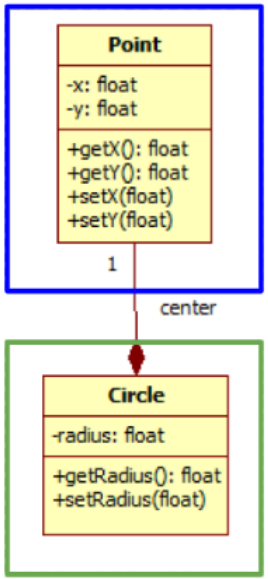
```
…
Point::Point(){
  cout << "Point()" << endl;
  setX(0);
  setY(0);}
Point::Point(float x, float y){
  cout << "Point(float, float)" <<
endl;
  setX(x);
  setY(y);}
```

ISEN
ALL IS DIGITAL!
OUEST

Implémentation d'un cercle

- Quand une instance de la classe Cercle est créée :
  1. Le constructeur par défaut de la classe "Point" est **implicitement** appelé
  2. Le constructeur approprié de la classe "Circle" est **explicitement** appelé

```
Point()
Circle()
```



ISEN
ALL IS DIGITAL!
OUEST

Généralisation

Soit une classe A (classe composite = container class) qui encapsule des instances des classes B1, B2, …, Bn (classe membre = content class). Si on crée une instance de la classe A :

1. Les constructeurs par défaut des classes B1, B2, …, Bn sont implicitement appelés (dans l'ordre de déclaration)
2. Le constructeur approprié de la classe A est explicitement appelé

ISEN ALL IS DIGITAL! OUEST

Généralisation

Soit une classe A (classe composite = container class) qui encapsule des instances des classes B1, B2, …, Bn (classe membre = content class). Si on crée une instance de la classe A :

1. Les constructeurs par défaut des classes B1, B2, …, Bn sont implicitement appelés (dans l'ordre de déclaration)
2. Le constructeur approprié de la classe A est explicitement appelé

Inconvénients :

- Les autres constructeur des classes membres ne sont pas utilisés
- **Modification d'un attribut d'une instance de la classe = double affectation**

main.cpp

```
#include "Circle.h"
int main() {
 Circle c(1.0);
 c.getCenter().setX(5.1);
 c.getCenter().setY(2.3);
 return 0;
}
```

Circle.h

```
#include "Point.h"
class Circle{
private:
 Point center;
public:
 Point& getCenter();
 ...
…};
```

ISEN
ALL IS DIGITAL!
OUEST

Généralisation

Soit une classe A (classe composite = container class) qui encapsule des instances des classes B1, B2, …, Bn (classe membre = content class). Si on crée une instance de la classe A :

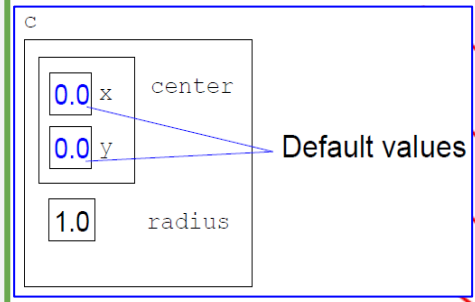
1. Les constructeurs par défaut des classes B1, B2, …, Bn sont implicitement appelés (dans l'ordre de déclaration)
2. Le constructeur approprié de la classe A est explicitement appelé

Inconvénients :

- Les autres constructeur des classes membres ne sont pas utilisés
- **Modification d'un attribut d'une instance de la classe = double affectation**

main.cpp

```
#include "Circle.h"
int main() {
 Circle c(1.0);
 c.getCenter().setX(5.1);
 c.getCenter().setY(2.3);
 return 0;
}
```



ISEN
ALL IS DIGITAL!
OUEST

Généralisation

Soit une classe A (classe composite = container class) qui encapsule des instances des classes B1, B2, …, Bn (classe membre = content class). Si on crée une instance de la classe A :

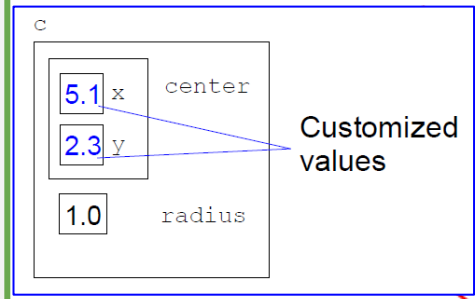
1. Les constructeurs par défaut des classes B1, B2, …, Bn sont implicitement appelés (dans l'ordre de déclaration)
2. Le constructeur approprié de la classe A est explicitement appelé

Inconvénients :

- Les autres constructeur des classes membres ne sont pas utilisés
- **Modification d'un attribut d'une instance de la classe = double affectation**

main.cpp

```
#include "Circle.h"
int main() {
 Circle c(1.0);
 c.getCenter().setX(5.1);
 c.getCenter().setY(2.3);
 return 0;
}
```



ISEN
ALL IS DIGITAL!
OUEST

Généralisation

Soit une classe A (classe composite = container class) qui encapsule des instances des classes B1, B2, …, Bn (classe membre = content class). Si on crée une instance de la classe A :

1. Les constructeurs par défaut des classes B1, B2, …, Bn sont implicitement appelés (dans l'ordre de déclaration)
2. Le constructeur approprié de la classe A est explicitement appelé

Inconvénients :

- Les autres constructeur des classes membres ne sont pas utilisés
- Modification d'un attribut d'une instance de la classe = double affectation
- **Problème de compilation en cas de suppression du constructeur par défaut de la classe membre**

main.cpp

```
#include "Circle.h"
int main() {
 Circle c(1.0);
 return 0;
}
```

```
error: no matching
function for call
to Point::Point()
```

```
…
/* Point::Point(){
  cout << "Point()" << endl;
  setX(0);
  setY(0);} */
Point::Point(float x, float y){
  cout << "Point(float, float)" <<
endl;
  setX(x);
  setY(y);}
…
```

Appel du constructeur de la classe membre

A.h

```
class A composite
{
        ...
        private:
        ...
        B1 b1;
        B2 b2;
        ...
        Bn bn;
        ...
        public
        A(...)
        …
};
```

membre

A.cpp

```
A::A(type1 arg1, type2 arg2, ..., typen argn)
: b1(arg1, arg2), b2(argn), ..., bn(arg6)
{
  ...
}
```

Appel du constructeur de la classe membre

A.h

```
class A composite
{
        ...
        private:
        ...
        B1 b1;
        B2 b2;
        ...
        Bn bn;
        ...
        public
        A(...)
        …
};
```

membre

A.cpp

```
A::A(type1 arg1, type2 arg2, ..., typen argn)
: b1(arg1, arg2), b2(argn), ..., bn(arg6)
{
  ...
}
```

Liste d'initialisation !

On ne peut pas appeler le constructeur de façon classique dans le code : Problème de la poule et de l'œuf.

Appel du constructeur de la classe membre

Circle.h

```
class Circle
{
    private:

    Point center;
    float radius;

    public:
    ...
    Circle(float,float,float);

};
```

container

content

Circle.cpp

```
Circle::Circle(float x, float y, float r)

: center(x,y)
{
    cout << "Circle(float, float, float)"
    << endl;
    setRadius(r);
}
```

Appel du constructeur de la classe membre

Circle.h

```
class Circle
{
    private:

    Point center;
    float radius;

    public:
    ...
    Circle(float,float,float);

};
```

container

content

Circle.cpp

```
Circle::Circle(float x, float y, float r)

: center(x,y)
{
    cout << "Circle(float, float, float)"
    << endl;
    setRadius(r);
}
```

Main.cpp

```
#include "Circle.h"

int main() {

    Circle c(5.1, 2.3, 1.0);

    return 0;

}
```

Appel du constructeur de la classe membre

Circle.h

```
class Circle
{
    private:

    Point center;
    float radius;

    public:
    ...
    Circle(float,float,float);

};
```

container

content

Circle.cpp

```
Circle::Circle(float x, float y, float r)

: center(x,y)
{
    cout << "Circle(float, float, float)"
    << endl;
    setRadius(r);
}
```

Main.cpp

```
#include "Circle.h"

int main() {

    Circle c(5.1, 2.3, 1.0);

    return 0;

}
```

```
Point(float, float)
Circle(float, float, float)
```

ISEN ALL IS DIGITAL! OUEST

Appel au destructeur

- Soit une classe A (classe composite = container class) qui encapsule des instances des classes B1, B2, …, Bn (classe membre = content class). Si on détruit une instance de la classe A :
  1. Le destructeur de A est appelé
  2. Les destructeurs des classes Bn, Bn-1, …, B1 sont implicitement appelés (dans l'ordre inverse de leurs déclarations)

Point.h

```
...
class Point{
...
public:
...
~Point(){
cout << "~Point()" <<
endl;
}
};
```

```
...
class Circle{
...
public:
...
~Circle(){
cout << "~Circle()" <<
endl;
}
};
```

Circle.h

main.cpp

```
#include "Circle.h"
int main() {
 Circle c(1.0);
return 0;
}
```

ISEN ALL IS DIGITAL! OUEST

Appel au destructeur

- Soit une classe A (classe composite = container class) qui encapsule des instances des classes B1, B2, …, Bn (classe membre = content class). Si on détruit une instance de la classe A :
  1. Le destructeur de A est appelé
  2. Les destructeurs des classes Bn, Bn-1, …, B1 sont implicitement appelés (dans l'ordre inverse de leurs déclarations)

Point.h

```
...
class Point{
...
public:
...
~Point(){
cout << "~Point()" <<
endl;
}
};
```

Circle.h

```
...
class Circle{
...
public:
...
~Circle(){
cout << "~Circle()" <<
endl;
}
};
```

main.cpp

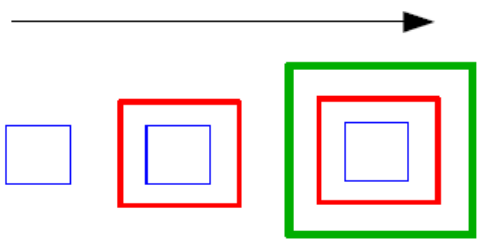
```
#include "Circle.h"
int main() {
 Circle c(1.0);
return 0;
}
```

```
Point(float, float)
Circle(float, float, float)
~Circle()
~Point()
```

ISEN
ALL IS DIGITAL!
OUEST

Synthèse

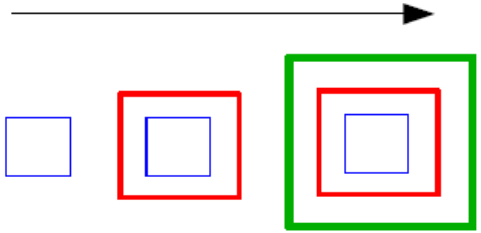
- Quand un objet composé est créé :
  - Les constructeurs de la classe membre sont appelés avant les constructeurs de la classe composite :



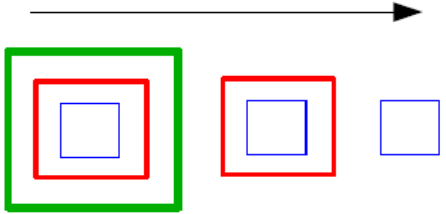
ISEN
ALL IS DIGITAL!
OUEST

Synthèse

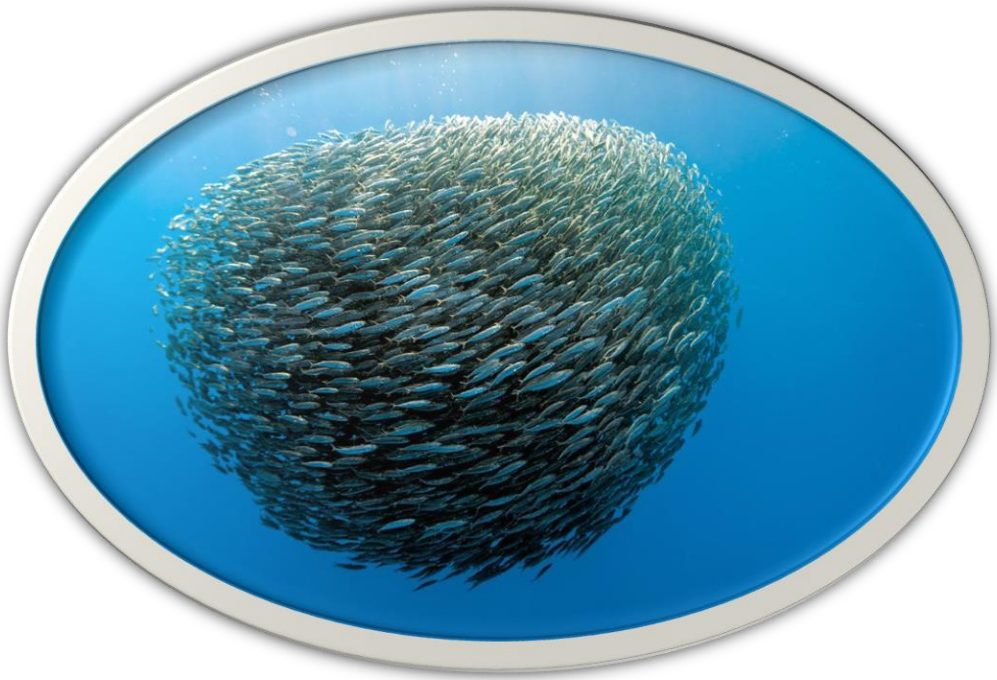
- Quand un objet composé est créé :
  - Les constructeurs de la classe membre sont appelés avant les constructeurs de la classe composite :



- Quand un objet est détruit :
  - Le destructeur de la classe composite est appelé avant les destructeurs de la classe membres :



Agrégation

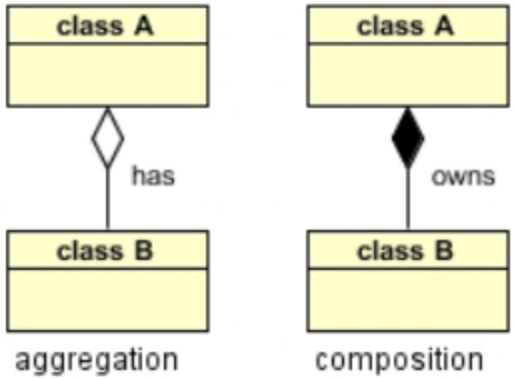


ISEN
ALL IS DIGITAL!
OUEST

Composition vs Agrégation

Composition = Relation "**fait partie de**" : si un objet B **fait partie d**'un objet A alors B ne peut pas exister sans A. Si A est détruit alors B également

Agrégation = Relation "**a un** " : si un objet A **a un** objet B alors B peut vivre sans A



La composition est une agrégation forte

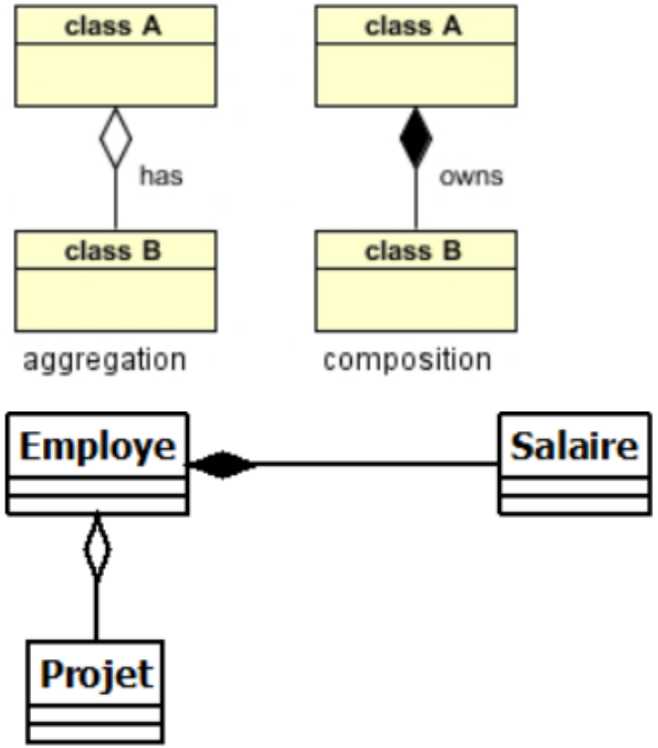
ISEN
ALL IS DIGITAL!
OUEST

Composition vs Agrégation

Composition = Relation "**fait partie de**" : si un objet B **fait partie d**'un objet A alors B ne peut pas exister sans A. Si A est détruit alors B également

Agrégation = Relation "**a un** " : si un objet A **a un** objet B alors B peut vivre sans A

La composition est une agrégation forte



ISEN
ALL IS DIGITAL!
OUEST

Appel au destructeur

Circle.h

```
class Circle{
private:
Point* center;
float radius;
...
};
```

Circle.cpp

```
...
Circle::Circle(){
cout << "Circle()" <<
endl;
setRadius(0);
}
...
Circle::~Circle(){
cout << "~Circle()" <<
endl;
}
```

main.cpp

```
#include "Circle.h"
int main() {
 Circle c;
return 0;
}
```

ISEN
ALL IS DIGITAL!
OUEST

Appel au destructeur

Circle.h

```
class Circle{
private:
Point* center;
float radius;
...
};
```

Circle.cpp

```
...
Circle::Circle(){
cout << "Circle()" <<
endl;
setRadius(0);
}
...
Circle::~Circle(){
cout << "~Circle()" <<
endl;
}
```

main.cpp

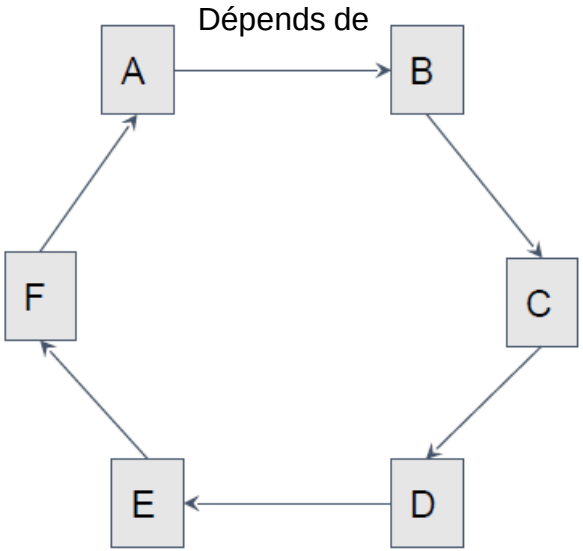
```
#include "Circle.h"
int main() {
 Circle c;
return 0;
}
```

```
Circle()
~Circle()
```

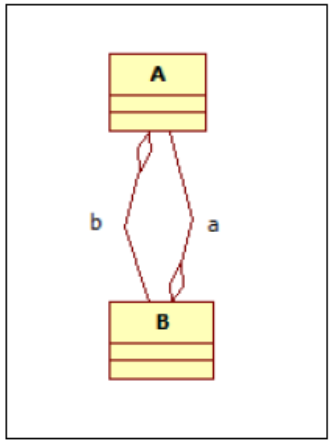
Dépendance circulaire

Définition

- La dépendance circulaire est la relation dans laquelle chaque classe dépend de l'autres classe



Exemple

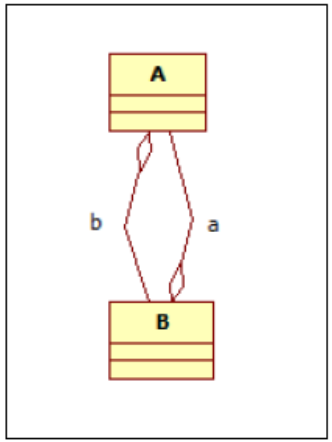



```
class A() {
   public :
        B b;
        int c;
}
```

```
class B() {
   public :
        A a;
        int c;
}
```

Évidemment impossible, car si taille de B = X alors :
A fait une taille de **X + int**
→ alors B fera une taille de **X + int + int**, mais alors A sera égal à **X + int + int +int** etc. à l'infini.

Exemple



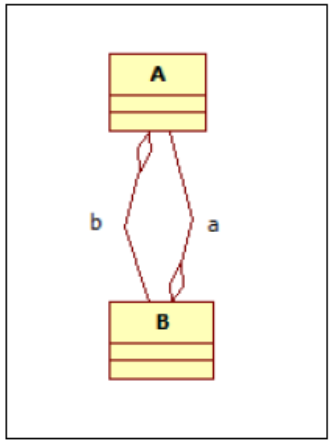

```
class A() {
   public :
        B b;
        int c;
}
```

```
class B() {
   public :
        A a;
        int c;
}
```

Impossible pour le compilateur de créer ce genre d'objet, alors comment faire ?

Évidemment impossible, car si taille de B = X alors :
A fait une taille de **X + int**
→ alors B fera une taille de **X + int + int**, mais alors A sera égal à **X + int + int +int** etc. à l'infini.

Exemple

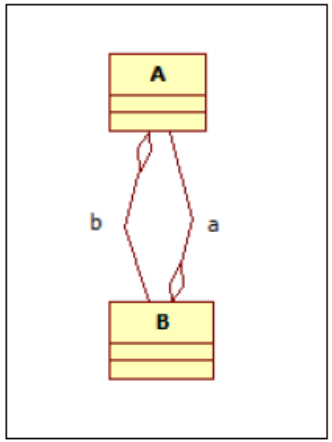



```
class A() {
   public :
        B* b = nullptr;
        int c;
}
```

```
class B() {
   public :
        A* a = nullptr;
        int c;
}
```

Solution : on peut utiliser des pointeurs car un pointeur a toujours une taille fixe (la taille d'une adresse en mémoire souvenez vous → 32 bit sur un système 32bit et 64bit sur un système 64bit)

Exemple




```
class A() {
   public :
        B* b = nullptr;
        int c;
}
```

```
class B() {
   public :
        A* a = nullptr;
        int c;
}
```

Mais ! Il y a quelques pièges !

ISEN
ALL IS DIGITAL!
OUEST

Exemple

ABTest.cpp

```
#include <iostream>
#include "A.h"
using std::cout;
using std::endl;

int main() {

    A* a = new A();

➔   cout << a->b << end;

    return 0;

}
```

Standard output

ISEN
ALL IS DIGITAL!
OUEST

Exemple

ABTest.cpp

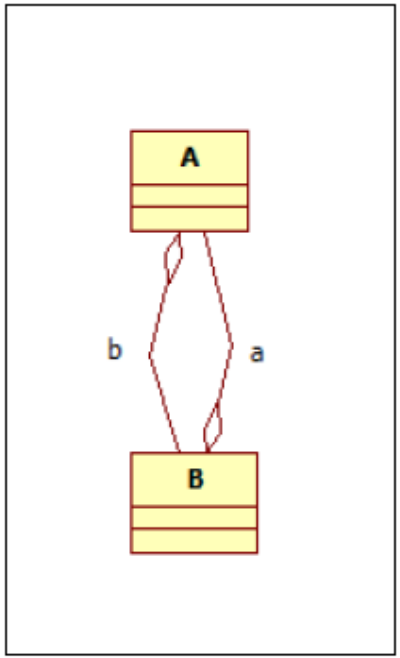
```
#include <iostream>
#include "A.h"
using std::cout;
using std::endl;

int main() {

    A* a = new A();

    cout << a->b << end;

    return 0;

}
```

Compiler output

```
B.h:3:7: error: redefinition of
'class B'
B.h:3:8: error: previous
definition of 'class B'
In file included from B.h:1:0,
                 from A.h:1,
                 from B.h:1,
...
A.h:3:7: error: redefinition of
'class A'
A.h:3:8: error: previous
definition of 'class A'
In file included from A.h:1:0,
                 from B.h:1,
...
```

ISEN
ALL IS DIGITAL!
OUEST

Exemple



A.h

```
#ifndef _CLS_A_
#define _CLS_A_

#include "B.h"
class A{
      public:
      B* b = nullptr;
};

#endif
```

B.h

```
#ifndef _CLS_B_
#define _CLS_B_

#include "A.h"
class B{
      public:
      A* a = nullptr;
};

#endif
```

ISEN
ALL IS DIGITAL!
OUEST

Exemple

ABTest.cpp

```
#include <iostream>
#include "A.h"
using std::cout;
using std::endl;

int main() {

    A* a = new A();

➔   cout << a->b << end;

    return 0;

}
```

Standard output

ISEN
ALL IS DIGITAL!
OUEST

Exemple

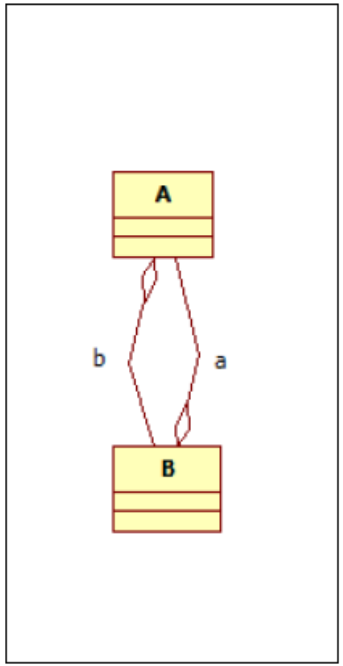
ABTest.cpp

```
#include <iostream>
#include "A.h"
using std::cout;
using std::endl;

int main() {

    A* a = new A();

    cout << a->b << end;

    return 0;

}
```

Compiler output

```
In file included from A.h:4:0,
                 from
ABTest.cpp:3:
B.h:8:2: error: 'A' does not
name a type
```

Exemple



A.h

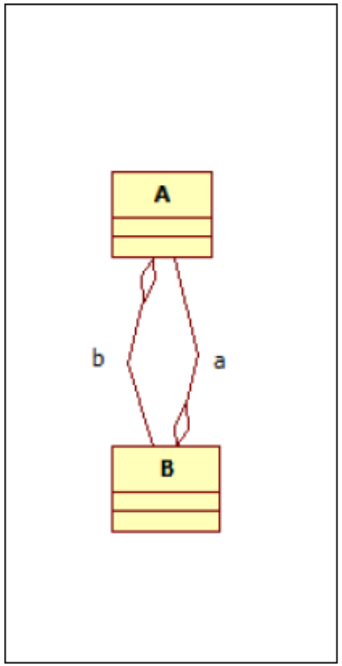
```
#ifndef _CLS_A_
#define _CLS_A_

#include "B.h"

class B;

class A{
    public:
    B* b;
};

#endif
```

B.h

```
#ifndef _CLS_B_
#define _CLS_B_

#include "A.h"

class A;

class B{
    public:
    A* a;
};

#endif
```

Exemple

On parle de déclaration avancée (forward declaration) !




```
#ifndef _CLS_A_
#define _CLS_A_

#include "B.h"

class B;

class A{
    public:
    B* b;
};

#endif
```

```
#ifndef _CLS_B_
#define _CLS_B_

#include "A.h"

class A;

class B{
    public:
    A* a;
};

#endif
```

ISEN
ALL IS DIGITAL!
OUEST

Exemple

ABTest.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "A.h"

int main() {

    A* a = new A();

    cout << a->b << end;

    return 0;

}
```

Standard output

```
0
```

FIN