

ISEN

ALL IS DIGITAL!

LILLE



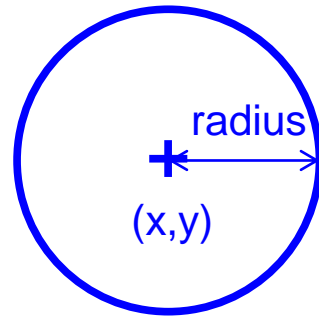
yncréa



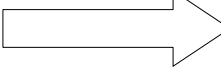
Le Langage C++

La composition et l'agrégation

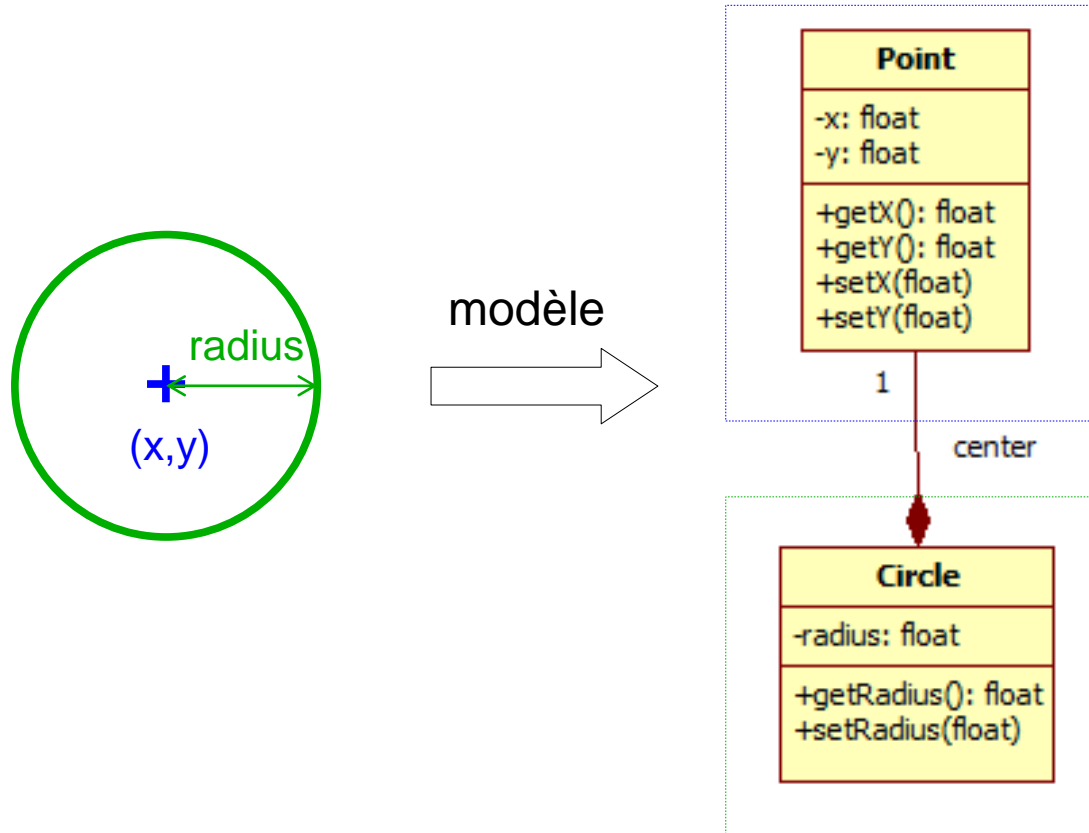
La composition



modèle



Circle
-x: float -y: float -radius: float
+getX(): float +getY(): float +getRadius(): float +setX(float) +setY(float) +setRadius(float)



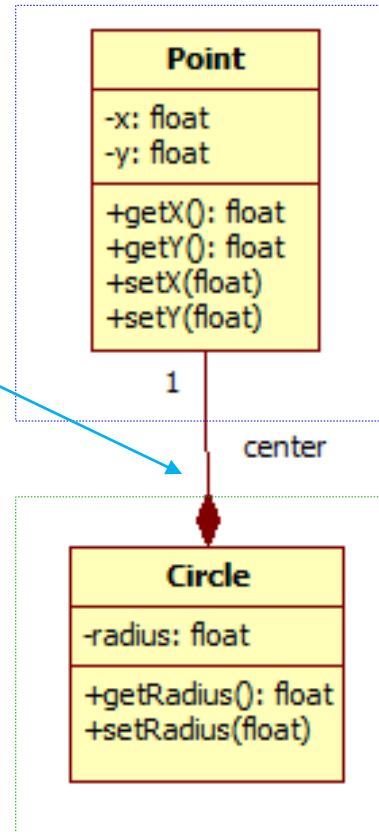
✓ Classe réutilisable

✓ Eviter la redondance dans le code

Le lien de composition permet au Cercle d'encapsuler un point qui devient son centre.

- Un lien fort
- Les deux classe auront le même cycle de vie

Toute instance de **Cercle** contiendra une instance de **Point**



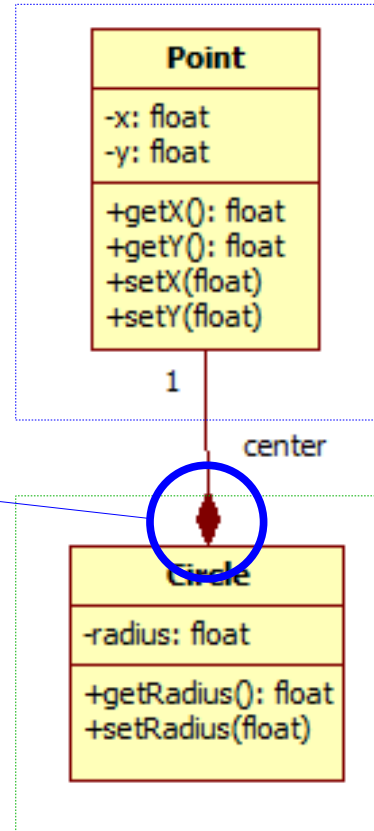
⚠ Ne pas oublier cette inclusion

Circle.h

```
#include "Point.h"
```

```
class Circle{
private:
    Point center;
    float radius;

public:
    Circle();
    Circle(float);
    float getRadius();
    void setRadius(float);
};
```



Point.h

```
class Point{
private:
    float x;
    float y;

public:
    Point();
    Point(float, float);
    float getX();
    float getY();
    void setX(float);
    void setY(float);
};
```

Point.cpp

```
#include "Point.h"
#include <iostream>
using namespace std;

Point::Point() {
    cout << "Point()" << endl;
    setX(0);
    setY(0);
}

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

...
```

Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

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

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

...
```


CircleTest.cpp

```
#include "Circle.h"

int main() {

    Circle c();

    return 0;

}
```

Standard output

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

CircleTest.cpp

```
#include "Circle.h"

int main() {

Circle c(1.0);

return 0;

}
```

Standard output

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

- Lorsqu'une instance de **Circle** est créée:
 1. Le constructeur par défaut de **Point** est appelé implicitement
 2. Le constructeur approprié de **Circle** est appelé explicitement

- Supposons que A (objet composé) encapsule les instances de B_1, B_2, \dots, B_n (les composants) :
- Lorsqu'une instance de A est créée :
 1. Les constructeurs par défaut de B_1, B_2, \dots, B_n sont appelés implicitement
 2. Le constructeur approprié de A est appelé explicitement

Problèmes

- Redondances dans le code et dans l'exécution
- Il est possible qu'un des composants n'ait pas de constructeur par défaut

```
class A Objet composé
{
    ...
private:
    ...
    B1 b1;
    B2 b2;
    ...
    Bn bn; Composants
    ...
public
    A(...)
    ...
};
```

A.h

Les constructeurs personnalisés des composants peuvent être appelés avant le constructeur de l'objet composé en utilisant la syntaxe suivante :

```
A::A(...) : b1(...), b2(...), ..., bn(...)\n{\n    ...\n}
```

A.cpp

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

Objet composé

Composants

A.h

Une partie des arguments du constructeur peuvent être passés (transférés) aux constructeurs des composants :

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

A.cpp

Circle.h

```
class Circle Objet composé
{
private:

Point center; Composant
float radius;

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

Circle.cpp

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

Certains arguments sont transférés au constructeur Point().
Les autres sont destinés aux attributs spécifiques de la classe cercle.

Circle.h

```
class Circle Objet composé
{
private:
Point center; Composant
float radius;

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

Circle.cpp

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

La syntaxe du transfert d'arguments et aussi valide pour les attributs.

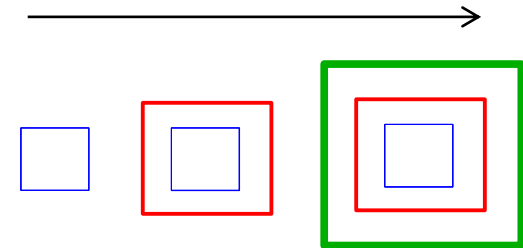
Aucun contrôle ne sera fait sur la valeur de l'argument (ex. $r \geq 0$)

- Supposons que A (objet composé) encapsule les instances de B_1, B_2, \dots, B_n (les composants) :
- Lorsqu'une instance de A est détruite :
 1. Le destructeur de A est appelé
 2. Les destructeurs de B_1, B_2, \dots, B_n sont appelés dans l'ordre inverse de leur déclaration

 Ne pas confondre les règles de création et de destruction

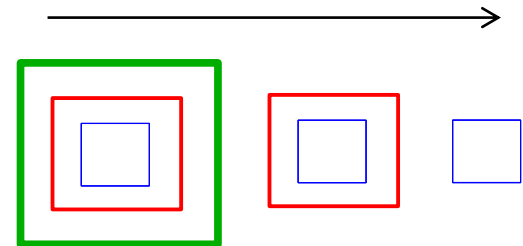
Lorsque un objet composé est **créé** :

Les constructeurs sont appelés de l'objet composant vers l'objet composé.
(de l'intérieur vers l'extérieur)



Lorsque un objet composé est **détruit** :

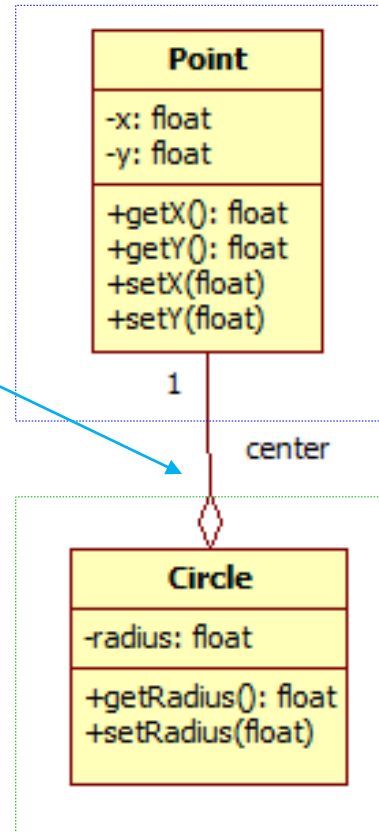
l'ordre d'appel des destructeur va de l'objet composé à ses différents composants
(de l'extérieur vers l'intérieur)



L'agrégation

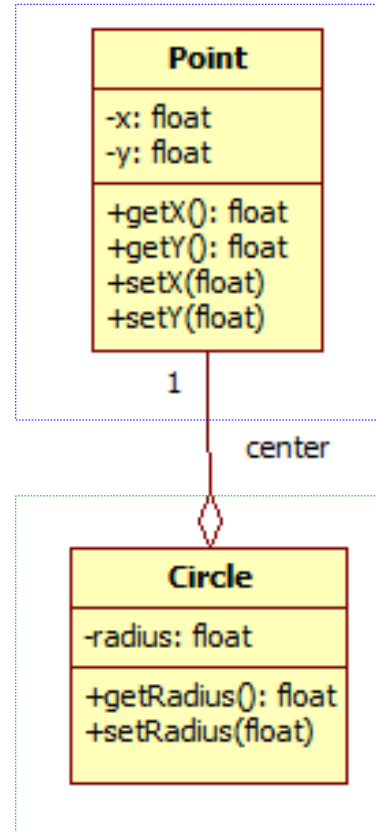
L'agrégation permet au Cercle d'avoir un lien avec un point qui devient son centre, et qui peut être partagé avec d'autres points.

- Un lien faible
- Les deux classes auront des cycles de vie différents



Circle.h

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



Circle.cpp

```
#include "Circle.h"
#include <iostream>
using namespace std;

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

...

Circle::~~Circle() {
    cout << "~Circle() " << endl;
}

...
```

CircleTest.cpp

```
#include "Circle.h"

int main() {
    Circle c;
    return 0;
}
```

Sortie standard

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

- Lorsque l'instance c est **créée**, le *pointeur* center est alloué, mais il n'est pas initialisé.
- Lorsque l'instance « c » est **détruite**, le *pointeur* center est désalloué, mais pas l'instance qu'il pointe. On parle de fuite mémoire.



Aucun appel au constructeur de Point
Aucun appel au destructeur de Point

Circle.cpp

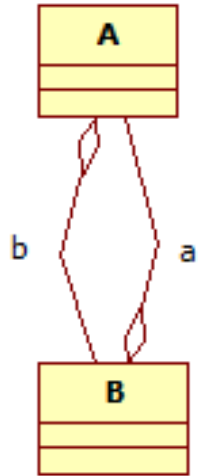
```
#include "Circle.h"
#include <iostream>
using namespace std;

Circle::Circle(float x, float y, float r){
    cout << "Circle()" << endl;
    center = new Point(x,y);    // Appel explicite du constructeur
    setRadius(r);
}
...

Circle::~~Circle(){
    cout << "~Circle()" << endl;
    delete center;              // Appel explicite du destructeur
}
...
```


La dépendance cyclique

Exemple de dépendance cyclique



A.h

```
#include "B.h"

class A{
public:
    B* b;
};
```

B.h

```
#include "A.h"

class B{
public:
    A* a;
};
```

ABTest.cpp

```
#include <iostream>
#include "A.h"
using namespace std;

int main() {

A* a = new A();

cout << a->b << end;

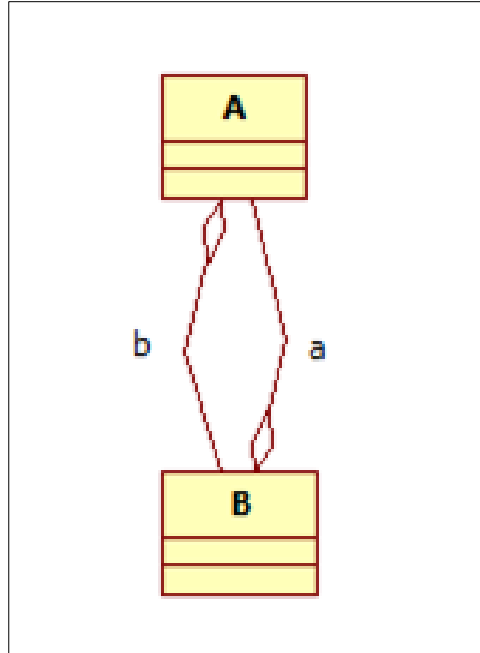
return 0;

}
```

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,
...
```

Définition conditionnelle (Directives de compilation)



A.h

```

#ifndef _CLS_A_
#define _CLS_A_

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

#endif
  
```

B.h

```

#ifndef _CLS_B_
#define _CLS_B_

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

#endif
  
```

Le préprocesseur ignore la partie entre `#define` et `#endif` si elle a été déjà définie

ABTest.cpp

```
#include <iostream>
#include "A.h"
using namespace std;

int main() {

A* a = new A();

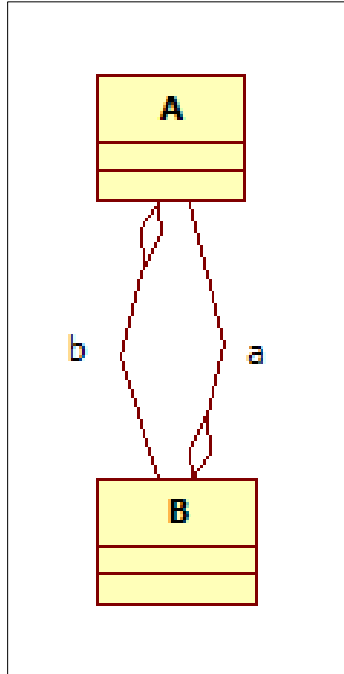
cout << a->b << end;

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
```



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
  
```

Il est possible de déclarer une classe de manière avancée, comme pour les fonctions

ABTest.cpp

```
#include <iostream>
#include "A.h"

using namespace std;

int main() {

A* a = new A();

cout << a->b << end;

return 0;

}
```

Standard output

0