

LE LANGAGE C++  
MASTER I  
LES POINTEURS MALINS

Jean-Baptiste.Yunes@univ-paris-diderot.fr

U.F.R. d'Informatique

Université Paris Diderot - Paris 7

**2019–2020**

- La gestion mémoire du C++ est pour le moins désagréable...
- allocation dynamique de bas-niveau (raw pointers) :
  - pointeurs fous (dangling pointers),
  - fuites mémoire (memory leaks)
- co-existence avec l'allocation statique/automatique

- STL : auto\_ptr
- BOOST : scoped\_ptr, shared\_ptr weak\_ptr, intrusive\_ptr
- C++11 : unique\_ptr, shared\_ptr, weak\_ptr

- Différents genres de pointeurs
  - pour différent genres de propriété
- Dans l'allocation dynamique, la difficulté principale réside dans la gestion de la propriété
  - qui «possède» la mémoire

- On a deux grands genres :
  - un seul propriétaire :
    - la mémoire disparaît avec son propriétaire
  - une propriété partagée :
    - la mémoire disparaît avec son dernier propriétaire

- Container de pointeurs nus :

```
class K {}  
vector<K *> v;  
v.push_back(new K);  
...  
delete v.back();  
v.pop_back();
```


ne pas oublier!!!!

- `unique_ptr` (`#include <memory>`)

```
#include <memory>
std::unique_ptr<int> p(new int(4));
std::unique_ptr<int> t(new int[100]);
```

- un pointeur = une ressource

```
#include <memory>
std::unique_ptr<int> p(new int(4));
std::unique_ptr<int> p2 = p;
```



- un pointeur = une ressource

```
#include <memory>
```

```
void f() {
```

```
    std::unique_ptr<A> p(new A);
```

```
}
```

```
int main() {
```

```
    f();
```

```
}
```

```
$ ./A
```

```
A()
```

```
~A()
```

```
$
```

RAII

```
class A {
```

```
public:
```

```
    A() {
```

```
        std::cout << "A()" << std::endl;
```

```
    }
```

```
    ~A() {
```

```
        std::cout << "~A()" << std::endl;
```

```
    }
```

```
};
```



- shared\_ptr

```
#include <memory>
```

```
std::shared_ptr<A> f() {  
    std::shared_ptr<A> p(new A);  
    std::cout << "in f" << std::endl;  
    return p;  
}  
int main() {  
    std::shared_ptr p = f();  
    std::cout << "in main" << std::endl;  
}
```

```
$ ./A  
A()  
in g  
in main  
~A()  
$
```

- Problème : les références circulaires...

```
#include <memory>
#include <vector>
#include <iostream>

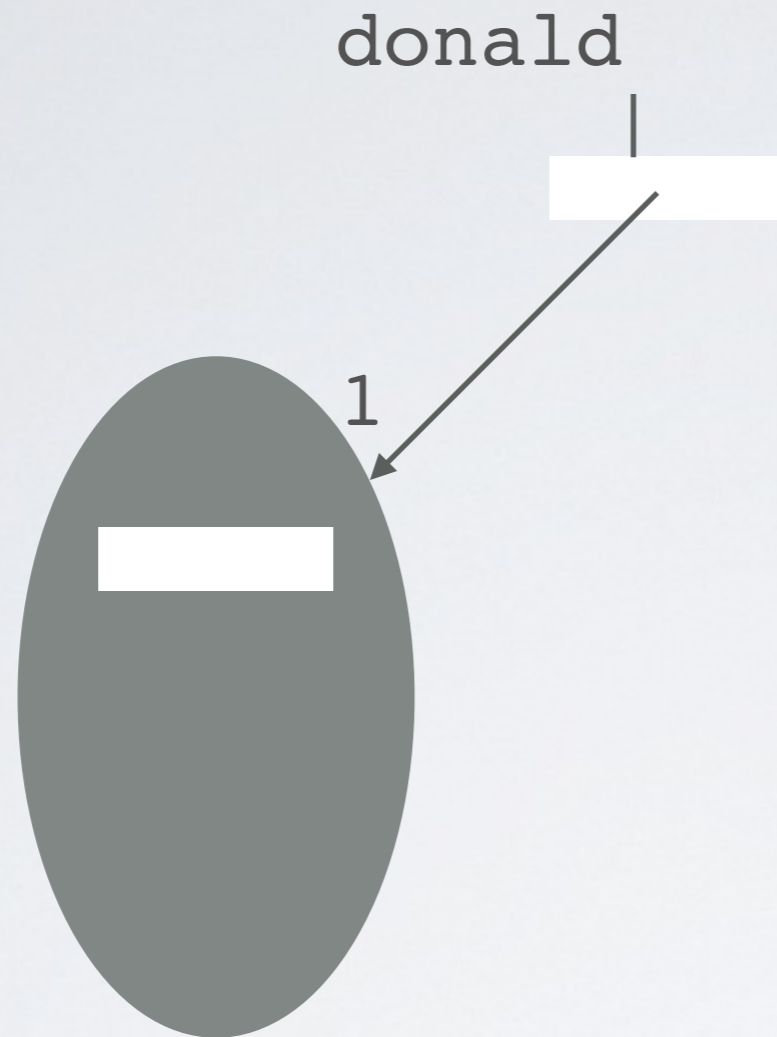
class F {
private:
    std::shared_ptr<F> myFriend;
public:
    void setFriend(std::shared_ptr<F> aFriend) { myFriend = aFriend; }
    F() { std::cout << "F()" << std::endl; }
    ~F() { std::cout << "~F()" << std::endl; }
};

int main() {
    std::shared_ptr<F> donald(new F);
    std::shared_ptr<F> kim(new F);
    donald->setFriend(kim);
    kim->setFriend(donald);
}
```

```
$ ./F
F()
F()
$
```

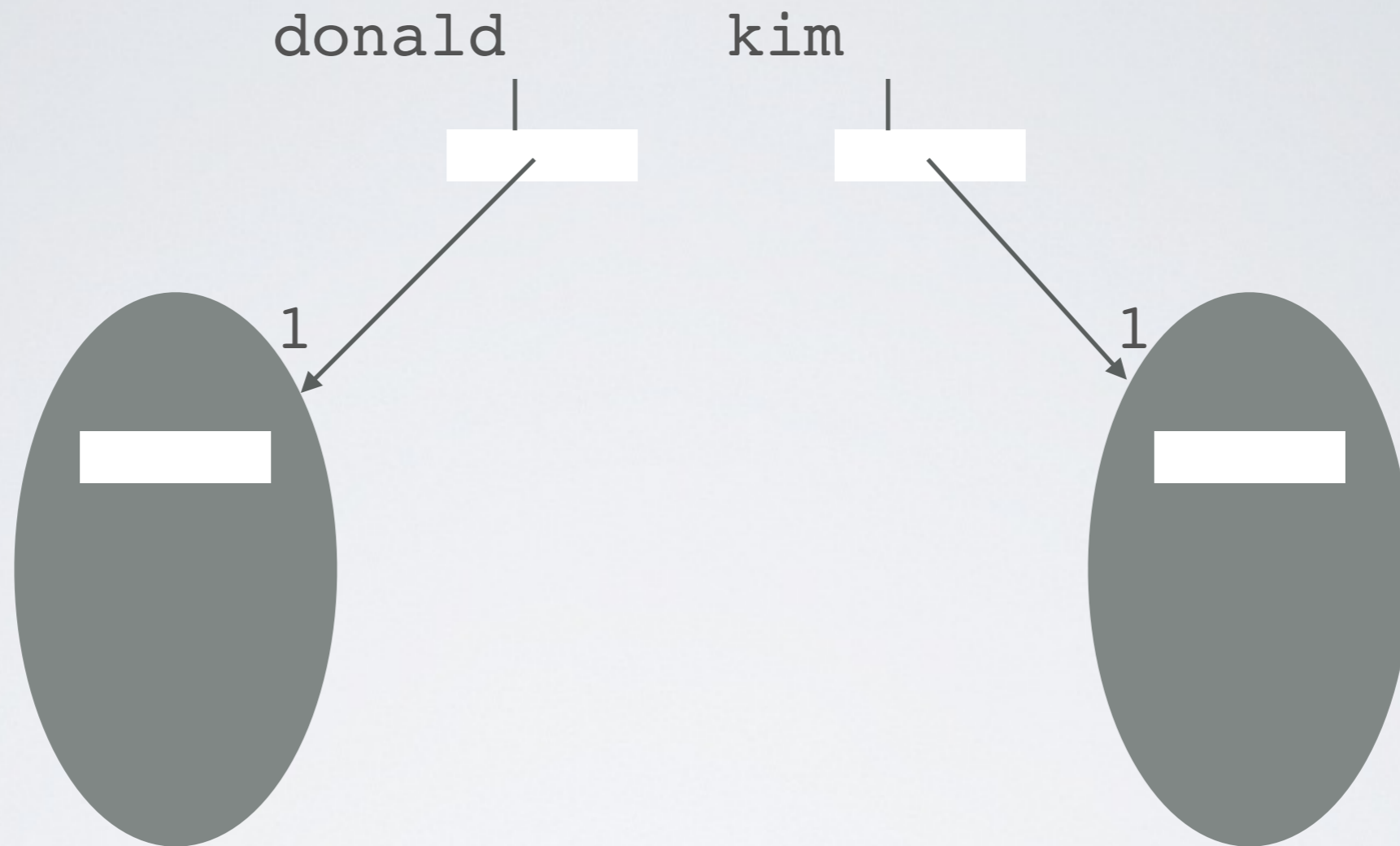
ouch!

- Problème : les références circulaires...



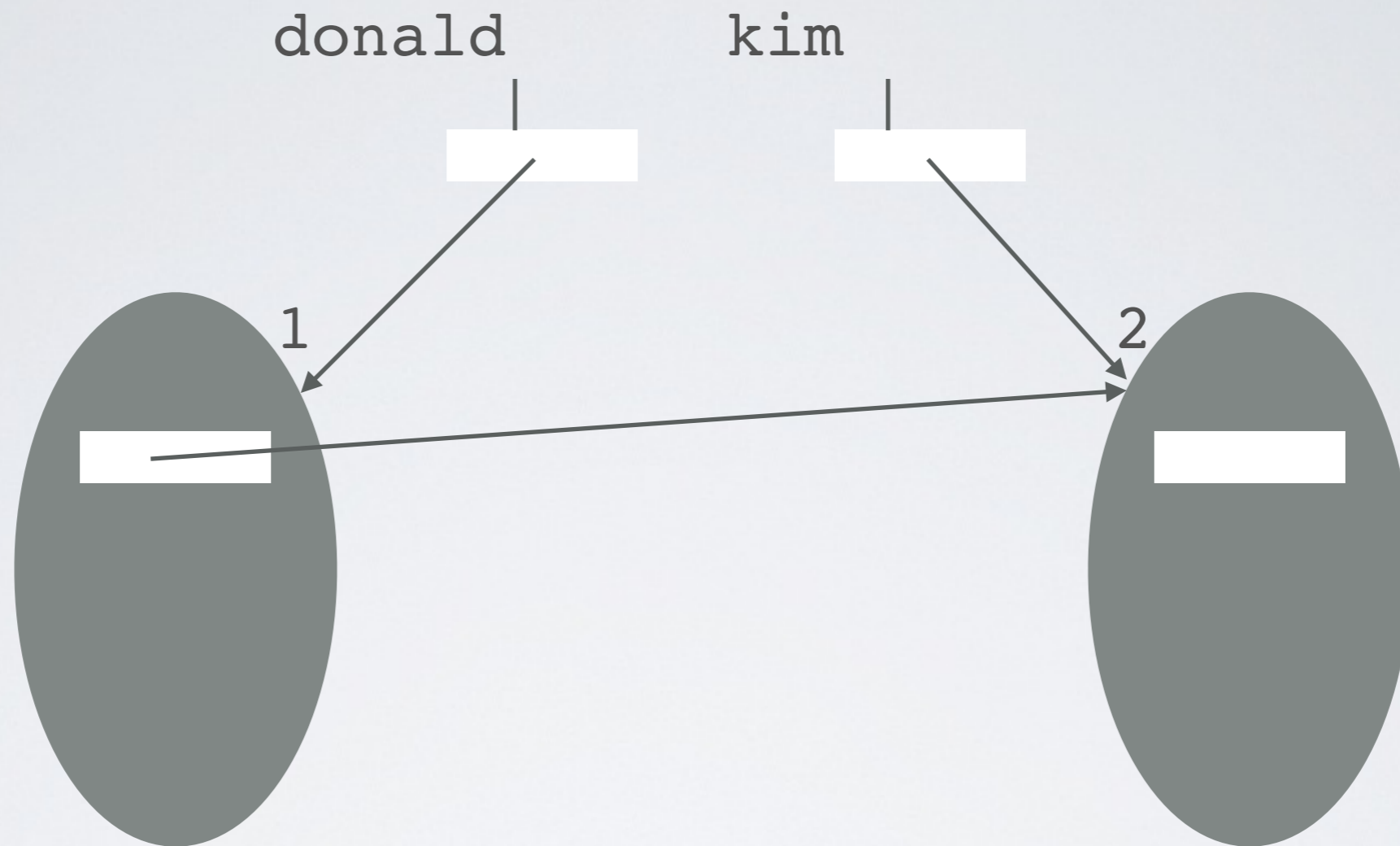
```
std::shared_ptr<F> donald(new F);
```

- Problème : les références circulaires...



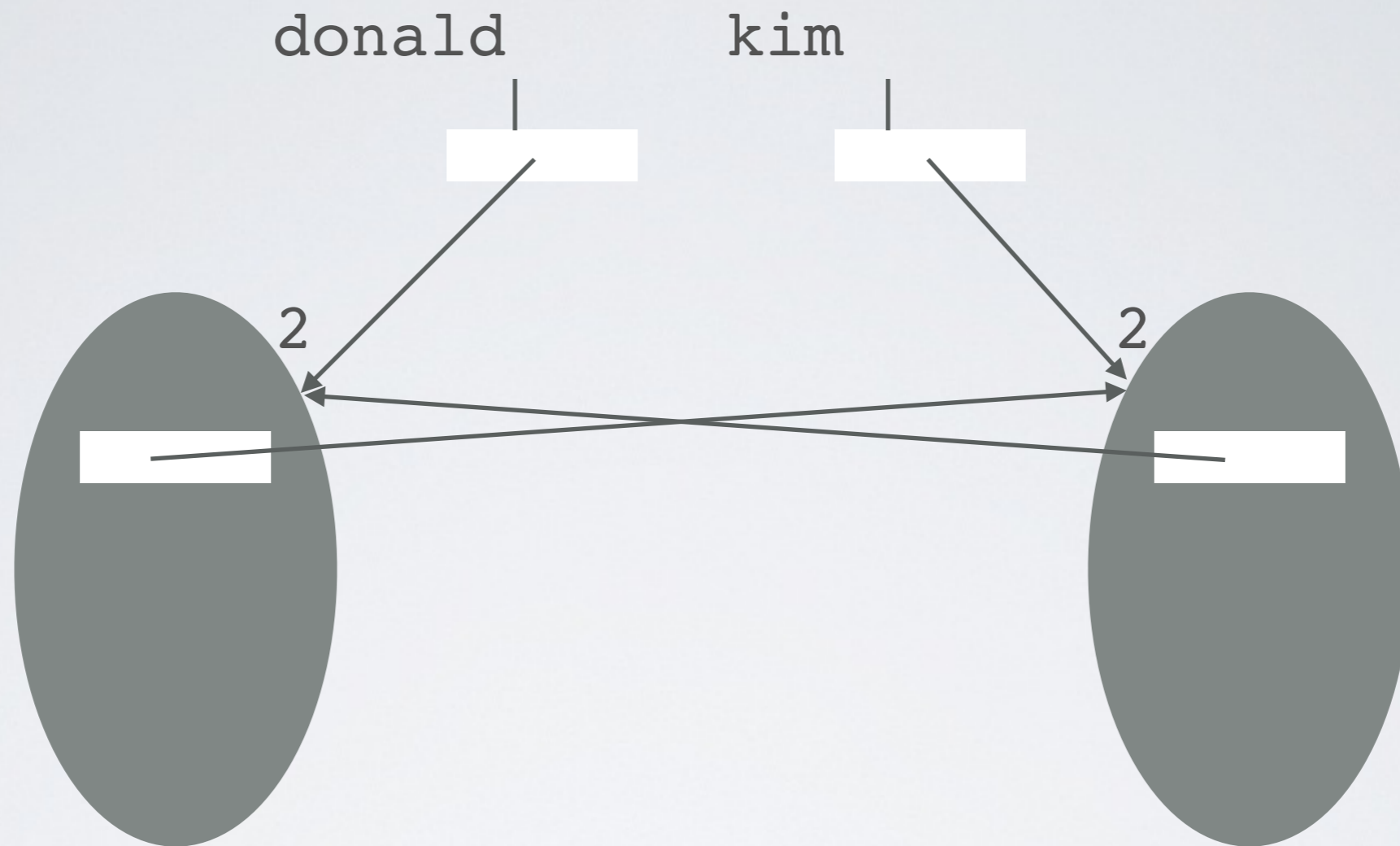
```
std::shared_ptr<F> donald(new F);  
std::shared_ptr<F> kim(new F);
```

- Problème : les références circulaires...



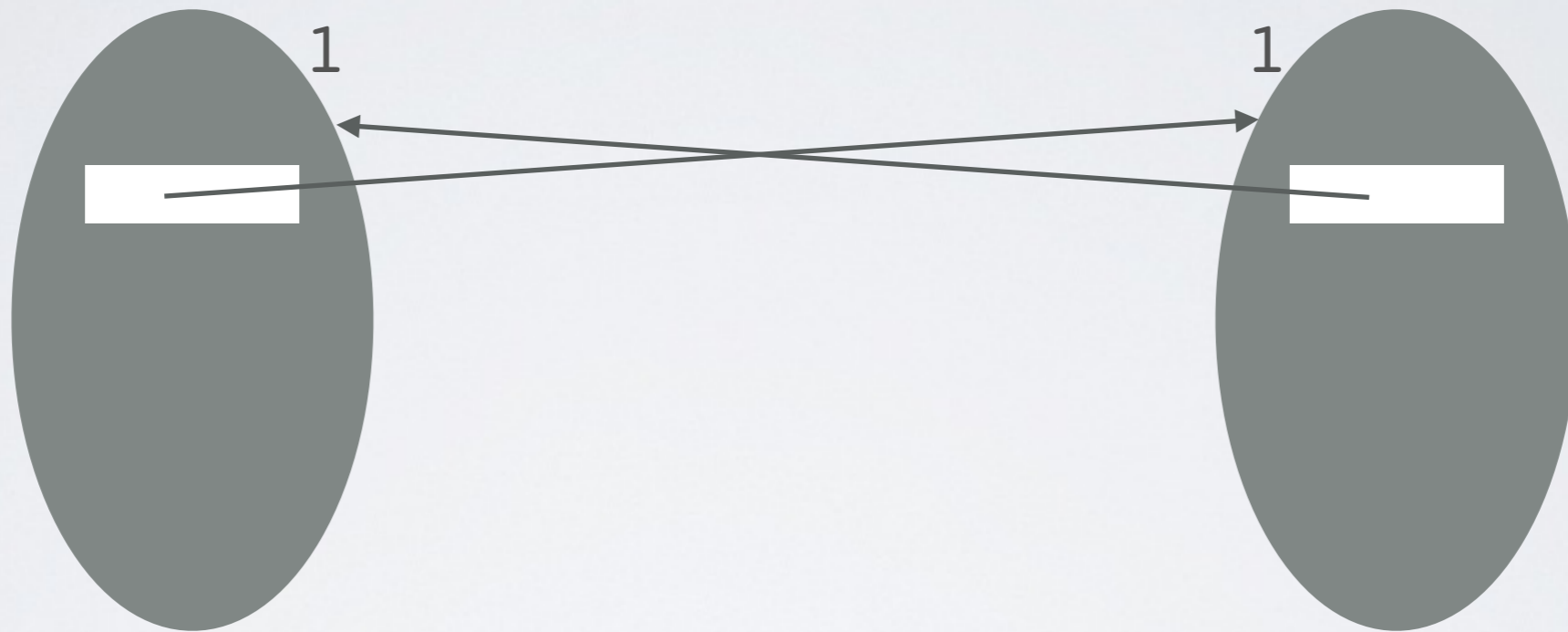
```
std::shared_ptr<F> donald(new F);  
std::shared_ptr<F> kim(new F);  
donald->setFriend(kim);
```

- Problème : les références circulaires...



```
std::shared_ptr<F> donald(new F);  
std::shared_ptr<F> kim(new F);  
donald->setFriend(kim);  
kim->setFriend(donald);
```

- Problème : les références circulaires...



```
std::shared_ptr<F> donald(new F);  
std::shared_ptr<F> kim(new F);  
donald->setFriend(kim);  
kim->setFriend(donald);  
}
```

- weak\_ptr : un shared\_ptr qui ne «compte» pas...

```
#include <memory>
#include <vector>
#include <iostream>
```

weak!!!!

```
class F {
private:
    std::weak_ptr<F> myFriend;
public:
    void setFriend(std::shared_ptr<F> aFriend) { myFriend = aFriend; }
    F() { std::cout << "F()" << std::endl; }
    ~F() { std::cout << "~F()" << std::endl; }
};
```

```
int main() {
    std::shared_ptr<F> donald(new F);
    std::shared_ptr<F> kim(new F);
    donald->setFriend(kim);
    kim->setFriend(donald);
}
```

```
$ ./F
F()
F()
~F()
~F()
$
```



- On peut dériver un `weak_ptr` depuis un `shared_ptr`
  - un pointeur faible
    - est un observateur, un simple lien
    - ne représente pas une propriété

- Exemple ok, mais comment faire si on ajoute un getter :

```
class F {
private:
    std::weak_ptr<F> myFriend;
public:
    void setFriend(std::shared_ptr<F> aFriend) { myFriend = aFriend; }
    F() { std::cout << "F()" << std::endl; }
    ~F() { std::cout << "~F()" << std::endl; }
    std::weak_ptr<F> getFriend() { return myFriend; }
};

std::weak_ptr<F> f() {
    std::shared_ptr<F> donald(new F);
    std::shared_ptr<F> kim(new F);
    donald->setFriend(kim);
    kim->setFriend(donald);
    return kim->getFriend();
}

int main() {
    std::weak_ptr<F> kimBestFriend;
    std::cout << "Before f()" << std::endl;
    kimBestFriend = f();
    std::cout << "After f()" << std::endl;
    std::weak_ptr<F> kimBestFriendBestFriend = kimBestFriend->getFriend();
}
```

ne compile pas

- même si on enlève l'accès :

```
class F {
private:
    std::weak_ptr<F> myFriend;
public:
    void setFriend(std::shared_ptr<F> aFriend) { myFriend = aFriend; }
    F() { std::cout << "F()" << std::endl; }
    ~F() { std::cout << "~F()" << std::endl; }
    std::weak_ptr<F> getFriend() { return myFriend; }
};

std::weak_ptr<F> f() {
    std::shared_ptr<F> donald(new F);
    std::shared_ptr<F> kim(new F);
    donald->setFriend(kim);
    kim->setFriend(donald);
    return kim->getFriend();
}

int main() {
    std::weak_ptr<F> kimBestFriend;
    std::cout << "Before f()" << std::endl;
    kimBestFriend = f();
    std::cout << "After f()" << std::endl;
}
```

```
$ ./F
Before f()
F()
F()
~F()
~F()
After f()
$
```

«dangling»

```

class F {
private:
    std::weak_ptr<F> myFriend;
    std::string name;
public:
    void setFriend(std::shared_ptr<F> aFriend) { myFriend = aFriend; }
    F(std::string n) { name=n; std::cout << "F(" << n << ")" << std::endl; }
    ~F() { std::cout << "~F()" << std::endl; }
    std::shared_ptr<F> getFriend() { return myFriend.lock(); }
    std::string getName() { return name; }
};

std::shared_ptr<F> f() {
    std::shared_ptr<F> donald(new F("donald"));
    std::shared_ptr<F> kim(new F("kim"));
    donald->setFriend(kim);
    kim->setFriend(donald);
    return kim->getFriend();
}

int main() {
    std::shared_ptr<F> kimBestFriend;
    std::cout << "Before f()" << std::endl;
    kimBestFriend = f();
    std::cout << "After f()" << std::endl;
    std::cout << kimBestFriend->getName() << std::endl;
}

```

```

$ ./F
Before f()
F()
F()
~F()
After f()
donald
~F()

```

- on ne fait pas grand-chose d'un `weak_ptr`, c'est un lien
  - mais on peut le transformer en `shared_ptr` *via* `lock()`
- un `share_ptr` peut se dériver simplement en `weak_ptr`