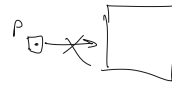
GC

int i = 10;

new objet();

objet* p;

objet* p = new objet() =



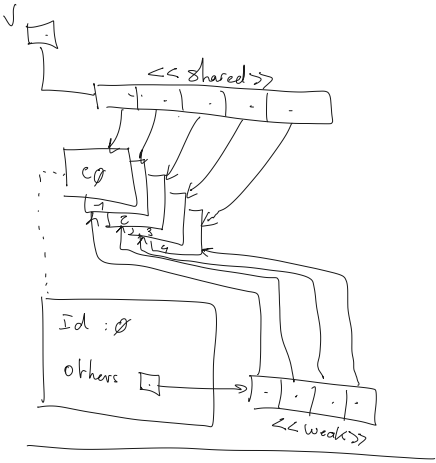
p = null

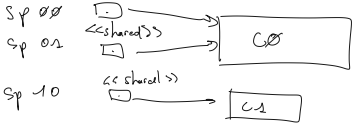
```
//   std::unique_ptr<LargeObject> pLarge(new LargeObject());
   std::unique_ptr<LargeObject> pLarge(make_unique<LargeObject>());
```

Pattern RAII : https://fr.wikipedia.org/wiki/Resource_acquisition_is_initialization

Implémentation avec les smart pointers :

```
#include <memory>
class widget
{
private:
   std::unique_ptr<int[]> data;
public:
   widget(const int size) { data = std::make_unique<int[]>(size); }
   void do_something() {}
};

void functionUsingWidget() {
   widget w(1000000);  // lifetime automatically tied to enclosing scope
                       // constructs w, including the w.data gadget member
   // ...
   w.do_something();
   // ...
} // automatic destruction and deallocation for w and w.data
```



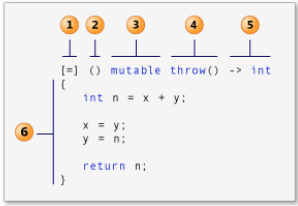


Consultez les membres de Weak et Shared Pointers :
Weak :
https://docs.microsoft.com/fr-fr/cpp/standard-library/weak-ptr-class?view=msvc-170#expired
Shared :
https://docs.microsoft.com/fr-fr/cpp/standard-library/shared-ptr-class?view=msvc-170

Lambdas :

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
        // Lambda expression begins
        [](float a, float b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}
```

Cette illustration montre les éléments d'une expression lam



Capture : ne capture rien

[=] ou []

Capture toutes les variables locales (et params)

[i, j] = ne capture que i et j

Ty

```cpp
#include "Lambdas.h"

using namespace std;

void abssort(float* x, unsigned n) {
    std::sort(_First:x, _Last:x + n,
        // Lambda expression begins
        _Pred:[](float a, float b) {
            return (std::abs(_Xx:a) < std::abs(_Xx:b));
        } // end of lambda expression
    );
}

void Lambdas::main()
{
    float floats[] = { 1, 5, 3, 4, 2 };
    wcout << "Avant le tri :" << endl;
    for (float f : floats)  wcout << "" << f << endl;
    abssort(x:floats, n:5);
    wcout << "Après le tri :" << endl;
    for (float f : floats)  wcout << "" << f << endl;

    wstring message = L"variable locale";

    std::sort(_First:floats, _Last:floats+3,
        // Lambda expression begins
        //   ici, "message" n'est pas capturé, donc inutilisable dans le code :
        //   [](float a, float b) {
        // [message](float a, float b)
        // Toutes les locales sont ici capturées, transmises au code de la Lambda :
        _Pred:[=](float a, float b)
        {
            wcout << "Dans la boucle :" << message << endl;
            float f = floats[0];
            return (std::abs(_Xx:a) < std::abs(_Xx:b));
        } // end of lambda expression
    );

    auto lambda [](int i)->int x1 = [](int i) { return i; };
    // Exécution de ma lambda ( qui est un pointeur vers une fonction )
    int res = x1(10);   // affiche 10


}
```

```cpp
#include "WeakPointers.h"

using namespace std;


class Controller
{
public:
    int Num;
    wstring Status;
    vector<weak_ptr<Controller>> others;
    explicit Controller(int i) : Num(i), Status(L"On")
    {
        wcout << L"Creating Controller" << Num << endl;
    }

    ~Controller()
    {
        wcout << L"Destroying Controller" << Num << endl;
    }

    // Demonstrates how to test whether the
    // pointed-to memory still exists or not.
    void CheckStatuses() const
    {
        for_each(_First:others.begin(), _Last:others.end(), _Func:[](weak_ptr<Controller> wp) {
            auto std::shared_ptr<Controller> p = wp.lock();
            if (p)
            {
                wcout << L"Status of " << p->Num << " = " << p->Status << endl;
            }
            else
            {
                wcout << L"Null object" << endl;
            }
            });
    }
};
```

```cpp
void RunTest()
{
    vector<shared_ptr<Controller>> v{
        make_shared<Controller>(_Args:0),
        make_shared<Controller>(_Args:1),
        make_shared<Controller>(_Args:2),
        make_shared<Controller>(_Args:3),
        make_shared<Controller>(_Args:4),
    };

    // Each controller depends on all others not being deleted.
    // Give each controller a pointer to all the others.
    for (int i = 0; i < v.size(); ++i)
    {
        for_each(_First:v.begin(), _Last:v.end(), _Func:[&v, i](shared_ptr<Controller> p) {
            if (p->Num != i)
            {
                v[i]->others.push_back(_Val:weak_ptr<Controller>(p));
                wcout << L"push_back to v[" << i << "]: " << p->Num << endl;
            }
            });
    }

    // Les compteurs de référence restent à 1 car on passe par une référence vers le sharepointer
    for_each(_First:v.begin(), _Last:v.end(), _Func:[](shared_ptr<Controller>& p) {
        wcout << L"use_count = " << p.use_count() << endl;
        p->CheckStatuses();
        });
}

void WeakPointers::main()
{
    RunTest();
    wcout << L"Press any key" << endl;
    char ch;
    cin.getline(_Str:&ch, _Count:1);
}
```

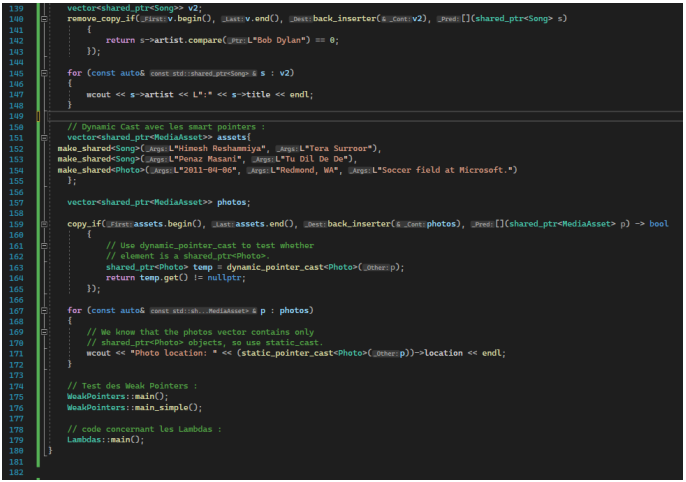
```cpp
void WeakPointers::main_simple()
{
    auto std::shared_ptr<Controller> sp00 = make_shared<Controller>(_Args:0);
    wcout << L"use_count début sp00 = " << sp00.use_count() << endl;
    shared_ptr<Controller> sp01 = sp00;
    auto std::shared_ptr<Controller> sp10 = make_shared<Controller>(_Args:1);

    wcout << L"use_count ensuite sp00 = " << sp00.use_count() << endl;
    wcout << L"use_count sp01 = " << sp01.use_count() << endl;
    wcout << L"use_count sp10 = " << sp10.use_count() << endl;

    // Une référence vers un Shared pointer n'incrémente pas son compteur d'instances :
    // Mais il faudra respecter son cycle de vie "a l'ancienne ! "
    shared_ptr<Controller>& spref = sp00;
    wcout << L"use_count sp00 = " << sp00.use_count() << endl;
    wcout << L"use_count spref = " << spref.use_count() << endl;

    weak_ptr<Controller> wp00 = sp00;
    weak_ptr<Controller> wp01(sp00);
    wcout << L"use_count sp00 avec deux weak_pointors qui le surveillent = " << sp00.use_count() << endl;
    wcout << L"use_count wp00 = " << wp00.use_count() << endl;

    // Affichage du détail :
    wcout << L"Id de sp00  " << sp00->Num << endl;
    {
        auto std::shared_ptr<Controller> lo = wp00.lock();
        wcout << L"Id de sp00 via wp00 " << lo->Num << endl;
        // mauvaise pratique :
        // wcout << L"Id de sp00 via wp01  " << wp01.lock()->Num << endl;
    }

    wcout << L"use_count sp00 Before Reset = " << sp00.use_count() << endl;
    sp00.reset();
    //sp00=nullptr;
    wp00 = sp00;
    wcout << L"use_count sp00 After Reset = " << sp00.use_count() << endl;

    if(!wp00.expired())
    {
        auto std::shared_ptr<Controller> lo = wp00.lock();
        wcout << L"Id de sp00 via wp00 " << lo->Num << endl;
    }
}
```

Aucun problème détecté

```
16
17  struct MediaAsset
18  {
19      virtual ~MediaAsset() = default; // make it polymorphic
20  };
21
22  struct Song : public MediaAsset
23  {
24      std::wstring artist;
25      std::wstring title;
26      Song(const std::wstring& artist_, const std::wstring& title_) :
27          artist{ artist_ }, title{ title_ } {}
28  };
29
30  struct Photo : public MediaAsset
31  {
32      std::wstring date;
33      std::wstring location;
34      std::wstring subject;
35      Photo(
36          const std::wstring& date_,
37          const std::wstring& location_,
38          const std::wstring& subject_) :
39          date{ date_ }, location{ location_ }, subject{ subject_ } {}
40  };
41
42
43  void UseRawPointer()
44  {
45      // Using a raw pointer -- not recommended.
46      //Song* pSong = new Song();
47
48      // Use pSong...
49
50      // Don't forget to delete!
51      //delete pSong;
52  }
53
54
55  void UseSmartPointer()
56  {
57      // Declare a smart pointer on stack and pass it the raw pointer.
58      //unique_ptr<Song> song2(new Song());
59
```

```
66  class LargeObject
67  {
68  public:
69      void DoSomething() {}
70  };
71
72  void ProcessLargeObject(const LargeObject& lo) {}
73  void SmartPointerDemo()
74  {
75      // Create the object and pass it to a smart pointer
76  //  std::unique_ptr<LargeObject> pLarge(new LargeObject());
77      std::unique_ptr<LargeObject> pLarge(make_unique<LargeObject>());
78
79      //Call a method on the object
80      pLarge->DoSomething();
81
82      // Pass a reference to a method.
83      ProcessLargeObject(lo: *pLarge);
84      LargeObject* p0 = pLarge.get();
85      pLarge.reset();
86      LargeObject* p1 = pLarge.get();
87      pLarge.reset();
88      LargeObject* p2 = pLarge.get();
89  }
90
91
92  int main()
93  {
94      std::cout << "Hello World!\n";
95      UseSmartPointer();
96      UseRawPointer();
97
98      SmartPointerDemo();
99
100     // Use make_shared function when possible.
101     auto std::shared_ptr<Song> sp1 = make_shared<Song>(_Args: L"The Beatles", _Args: L"Im Happy Just to Dance With You");
102
103     // Ok, but slightly less efficient.
104     // Note: Using new expression as constructor argument
105     // creates no named variable for other code to access.
106     shared_ptr<Song> sp2(new Song(L"Lady Gaga", L"Just Dance"));
107
```

```cpp
    // Ok, but slightly less efficient.
    // Note: Using new expression as constructor argument
    // creates no named variable for other code to access.
    shared_ptr<Song> sp2(new Song(L"Lady Gaga", L"Just Dance"));

    // When initialization must be separate from declaration, e.g. class members,
    // initialize with nullptr to make your programming intent explicit.
    //shared_ptr<Song> sp5(nullptr);
    //shared_ptr<Song> sp5;
    shared_ptr<Song> sp5(0);

    //Initialize with copy constructor. Increments ref count.
    auto std::shared_ptr<Song> sp3(sp2);

    //Initialize via assignment. Increments ref count.
    auto std::shared_ptr<Song> sp4 = sp2;

    //Initialize with nullptr. sp7 is empty.
    shared_ptr<Song> sp7(nullptr);

    // Initialize with another shared_ptr. sp1 and sp2
    // swap pointers as well as ref counts.
    sp1.swap(_Other: sp2);

    //Equivalent to: shared_ptr<Song> sp5;
    //...
    sp5 = make_shared<Song>(_Args:L"Elton John", _Args:L"I'm Still Standing");


    // Utilisation des collections avec les smart pointers :
    vector<shared_ptr<Song>> v{
          make_shared<Song>(_Args:L"Bob Dylan", _Args:L"The Times They Are A Changing"),
          make_shared<Song>(_Args:L"Aretha Franklin", _Args:L"Bridge Over Troubled Water"),
          make_shared<Song>(_Args:L"Thalía", _Args:L"Entre El Mar y Una Estrella")
    };

    vector<shared_ptr<Song>> v2;
    remove_copy_if(_First:v.begin(), _Last:v.end(), _Dest:back_inserter(& _Cont:v2), _Pred:[](shared_ptr<Song> s)
        {
            return s->artist.compare(_Ptr:L"Bob Dylan") == 0;
        });

    for (const auto& const std::shared_ptr<Song>& s : v2)
    {
```



```
    vector<shared_ptr<Song>> v2;
    remove_copy_if(_First: v.begin(), _Last: v.end(), _Dest: back_inserter(s _Cont: v2), _Pred: [](shared_ptr<Song> s)
        {
            return s->artist.compare(_Ptr: L"Bob Dylan") == 0;
        });

    for (const auto& const std::shared_ptr<Song> & s : v2)
    {
        wcout << s->artist << L":* " << s->title << endl;
    }

    // Dynamic Cast avec les smart pointers :
    vector<shared_ptr<MediaAsset>> assets{
make_shared<Song>(_Args: L"Hinosh Reshammiya", _Args: L"Tera Surroor"),
make_shared<Song>(_Args: L"Penaz Masani", _Args: L"Tu Dil De De"),
make_shared<Photo>(_Args: L"2011-04-06", _Args: L"Redmond, WA", _Args: L"Soccer field at Microsoft.")
    };

    vector<shared_ptr<MediaAsset>> photos;

    copy_if(_First: assets.begin(), _Last: assets.end(), _Dest: back_inserter(s _Cont: photos), _Pred: [](shared_ptr<MediaAsset> p) -> bool
        {
            // Use dynamic_pointer_cast to test whether
            // element is a shared_ptr<Photo>.
            shared_ptr<Photo> temp = dynamic_pointer_cast<Photo>(_Other: p);
            return temp.get() != nullptr;
        });

    for (const auto& const std::sh...MediaAsset> & p : photos)
    {
        // We know that the photos vector contains only
        // shared_ptr<Photo> objects, so use static_cast
        wcout << "Photo location: " << (static_pointer_cast<Photo>(_Other: p))->location << endl;
    }

    // Test des Weak Pointers :
    WeakPointers::main();
    WeakPointers::main_simple();

    // code concernant les Lambdas :
    Lambdas::main();
}
```

Règles Sonar :
https://rules.sonarsource.com/

Nouveautés en fonction des OS

- Windows 11 : https://docs.microsoft.com/fr-fr/windows/apps/whats-new/windows-11-build-22000
-