Bonjour tout le monde

Site de tutos fonctionnels multi OS :
https://www.server-world.info/en/

Gestion des erreurs

→ fonction → "renvoie" une erreur

AVANT

→ Code d'erreur

fct
Si (erreur)
alors ....
Sinon
....

~~Complètement obsolète~~

Depuis > go

Utilisation d'exceptions

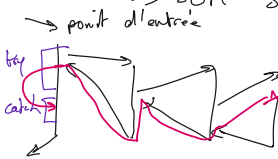
- "essayer de faire mes opérations"
  → peut "planter" à tout moment

catch ("attraper les exceptions")
→ réagir en fonction de l'exception

dans tous les cas (finally)
code à exécuter dans tous les cas

mots clefs
- Try / catch / finally
- throw (déclencher une exception)

⇒ SEH Structured Exception Handling



⇒ A APPLIQUER !

https://www.cisecurity.org/cis-benchmarks/

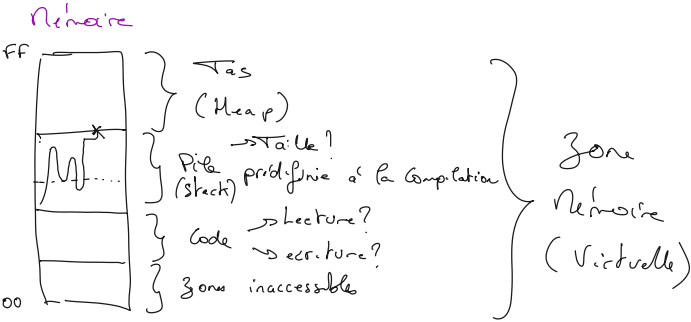
https://nvd.nist.gov/vuln/search

https://cve.mitre.org/

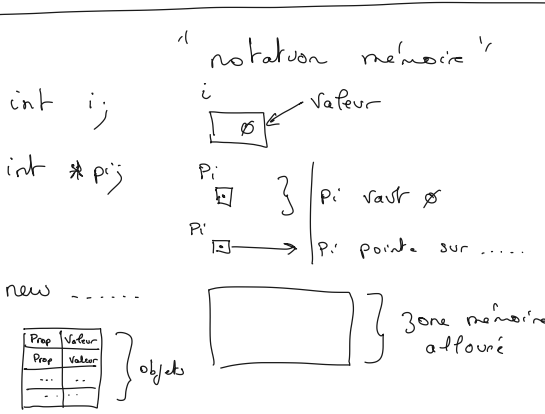
https://cwe.mitre.org/data/slices/1200.html

Les faiblesses C++ en CWE : https://cwe.mitre.org/data/definitions/659.html

---



Restriction de la conso mémoire par l'OS linux : https://www.server-world.info/en/note?os=CentOS_7&p=cgroups&f=1

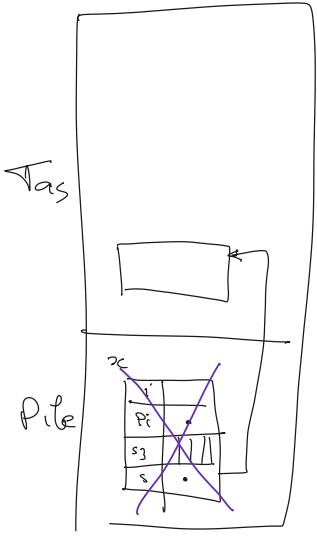


ex :

```
class X {
    int i;
    int *pi;
    char s3[10];
    char *s = new char[10];
}
```

→ alloué là où sera allouer l'objet

=> Dans le tas



{

X xc().

}

?

langage natif
C / C++

- "Vous faites votre ménage"

→ Cantons à l'intérieur de soi-même

langage géré

- Vous faites faire le ménage

→ Le ménage qui sera fait de manière non déterministe

Evaluation des tendances trends.google.com

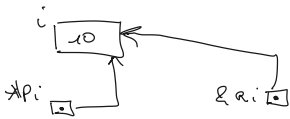
Les règles SonarLint : https://rules.sonarsource.com/cpp/RSPEC-1874

Lancement d'un serveur sonarQube community en conteneur linux :
sudo docker container run -d --name sonarqube -e SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true -p 9000:9000 sonarqube:9.4-community
login / mot de passe par défaut : admin/admin

Pointeurs / Références

Pointeur → élément de base (reste valable)

Inconvénients : alourdissement de la syntaxe —



*pi = 14 pi est une adresse

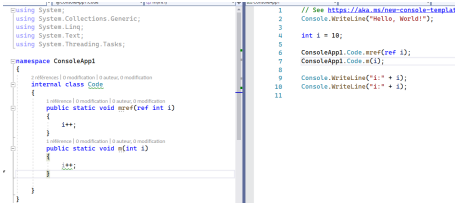
ri = 12 ri est une référence

C# → c'est tout l'inverse.

Tout est référence pour les objets.
Il faut être "unsafe" pour avoir des pointeurs

inconvénient : On ne sait pas explicitement si une variable est, ou pas, une référence —

C# renforce en obligeant à informer la connaissance d'une variable modifiable par l'appel



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    0 références | 0 modification | 0 auteur, 0 modification
    internal class Code
    {
        0 référence | 0 modification | 0 auteur, 0 modification
        public static void mref(ref int i)
        {
            i++;
        }
        0 référence | 0 modification | 0 auteur, 0 modification
        public static void m(int i)
        {
            i++;
        }
    }
}
```

```
// See https://aka.ms/new-console-templat
Console.WriteLine("Hello, World!");

int i = 10;


ConsoleApp1.Code.mref(ref i);
ConsoleApp1.Code.m(i);

Console.WriteLine("i:" + i);
Console.WriteLine("i:" + i);
```

C++ (y compris moderne) ne renforce pas l'expression de la conscience du risque de modification avec les références.

```
#include <iostream>

void mref(int& i)
{
    i++;
}

void m(int i)
{
    i++;
}

int main()
{
    int i = 10;
    int* pi = 0;
    pi = &i;    // adresse de i
    // modifier le contenu du i :
    *pi = 11;   // alourdissement de la syntaxe à
    int& ri=i;  // je déclare une référence sur un
    ri = 12;

    mref(i);    // i sera modifiée
    m(i);       // i n'est pas modifiée

    std::cout << "Hello World!\n";
}
```

4G → Prog. fonctionnelle → Passer des fonctions à des fonctions

3G → Prog "orientée objet" → Delphi, C++ (Java, C#, Python)

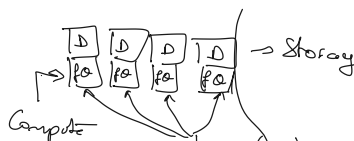
2G → Prog "structurée" → Pascal, C, Fortran

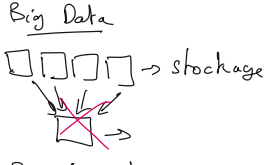
Générations :

1G → Prog "naturelle" → Scripts, basic, Cobol de base

Prog fonctionnelle | Gestion de Projets

SCRUM → Agile → DevOps





Hadoop →

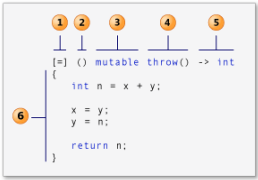
Plateforme opensource → Solution Big

"Concept de calcul distribué" → Map Reduce

```
long somme = entiers.stream()
                    .filter(v -> v <10).
                    .mapToInt(i -> i)
                    .sum();
System.out.println(somme);
```

(v -> { return v<10;})

équivaut à




```
// Utilisation de la syntaxe C++ moderne pour créer une collection de 5 int,
// l'allouer de façon moderne en respect du pattern RAII ( make_unique )
// pNums devenant une collection de 5 int
auto pNums = std::make_unique<std::vector<int>>(5);
// déclarer une variable de type expression lambda est "compliqué"
// du coup on s'autorise à utiliser auto pour obtenir une variable a qui est une expression lambda
// ( le code d'une fonction qui ici n'est pas appellée, mais qui le sera plus tard )
auto a = [ptr = std::move(pNums)]()
{
    // use ptr
};
```

Types génériques :

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
template <class T>
T GetMax(T a, T b) {
    T result;
    result = (a > b) ? a : b;
    return (result);
}

int GetMaxInt(int a, int b) {
    int result;
    result = (a > b) ? a : b;
    return (result);
}
long GetMaxLong(long a, long b) {
    long result;
    result = (a > b) ? a : b;
    return (result);
}

int main() {
    int i = 5, j = 6, k;
    long l = 10, m = 5, n;
    // Code paramétrique :
    k = GetMax<int>(i, j);
    n = GetMax<long>(l, m);
    // Equivaut à ( en C legacy ) :
    int k2 = GetMaxInt(i, j);
    long n2 = GetMaxLong(l, m);

    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

RAII

Implémentation du pattern RAII en C++ legacy conforme :

```C++
class widget
{
private:
    int* data;
public:
    widget(const int size) { data = new int[size]; } // acquire
    ~widget() { delete[] data; } // release
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);   // lifetime automatically tied to enclosing scope
                         // constructs w, including the w.data member
    w.do_something();

}// automatic destruction and deallocation for w and w.data
```

Pour le pattern, le contenu lourd sera dans le tas (3)

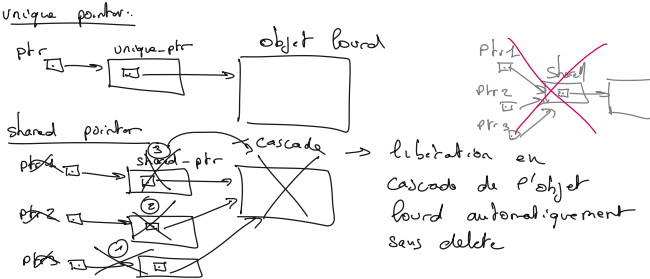
(4) en C++ legacy, nous utilisons new et delete

(1) objet "léger" créé sur la pile

(2) w sera forcément libéré (destructeur) automatiquement, y compris sur exception non gérée.

Implémentation du pattern en utilisant le C++ moderne (unique et shared pointers) :

l'objet sera référencé 1 fois → unique_ptr
n fois → shared_ptr



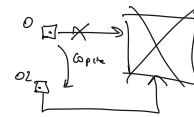
Compteur d'instances

recommandé

le Problème :

```
object *p; = 0;
object *o = new object();
object *o2 = o;
delete o;   o = 0;
o2->m();
```

→ ok ô la Compile, possible que ala ne plante pas —



---

le Compteur d'instance(s)

```
int nbi = 0;
object *o = new object(); i++;
object *o2 = o;  i++;

free(o, i); o = 0;
free(o2, i); o2 = 0;

void free(object *o, int &nb)
{ if(nb > 0)
    nb--;
  if(nb == 0)
  { delete o; }
}
```

```
{ object *o = new object();
(1) make something(o);
(2) push(o)
}

void make something(object *o2)
{
  ..... delete o2;
}
```

Java, c#, python, etc... (managed)

```
{ object o = new object();
  Push(o);
}
```

FReachables

→ a: appel destructeurs
b: récupérer la RAM
c: Compacter la RAM

Obtention d'un objet Cégur

unique
shared - ptr :

(1) automatiquement

```
std::unique_ptr<Node> head;

head = std::unique_ptr<Node>(new Node{data, std::move(head)});
```

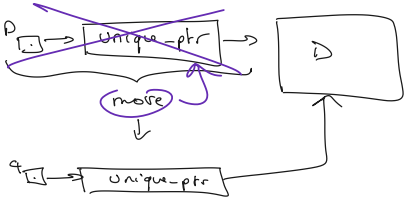
(2) via une fonction d'allocation

```
{
    // Create a (uniquely owned) resource
    std::unique_ptr<D> p = std::make_unique<D>();

    // Transfer ownership to `pass_through`,
    // which in turn transfers ownership back through the return value
    std::unique_ptr<D> q = pass_through(std::move(p));

    // `p` is now in a moved-from 'empty' state, equal to `nullptr`
    assert(!p);
}
```

je demande d'allouer le type D sans paranthèses de son .ctor()

est un objet local automatique (son destructeur sera appelé dans tous les cas en sortie du scope)

Le Constructeur par déplacement




```
    // `p` is now in a moved-from 'empty' state, equal to `nullptr`
    assert(!p); // p en effet est non null, mais le compilateur emet un WARNING
}
```

```
    {
        cout << ".dtor()\n";
    }

};

void TestSmartPointers()
{
    unique_ptr<Person> p1 = make_unique<Person>();  // Alloue un Person
    unique_ptr<Person> p2 = unique_ptr<Person>();   // N'alloue pas un Person

    // Utilisation de l'objet derrière le smart pointer.

    Person* pp = p1.get();
    pp->Age = 10;

    Person* pp2 = p1.get();
    cout << "Age:" << pp2->Age << "\n";

    unique_ptr<Person> p3 = move(p1);
    Person* pp3 = p3.get();
    cout << "Age:" << pp3->Age << "\n";

    Person* pp4 = p1.get(); // PAS BON !!
    cout << "Age:" <<
        inline Person *std::unique_ptr<Person>::get() const
        Rechercher en ligne
        C26800: Utilisation d'un objet déplacé : "p1" (lifetime.1).
}
```

Weak Pointers

stocke une référence non intrusive (ne manipule pas le compteur d'instance)

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

std::weak_ptr<int> gw;

void observe()
{
    std::cout << "gw.use_count() == " << gw.use_count() << "; ";
    // we have to make a copy of shared pointer before usage:
    if (std::shared_ptr<int> spt = gw.lock()) {
        std::cout << "*spt == " << *spt << '\n';
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;

        observe();
    }

    observe();
}
```

tentative de récupération du pointer si pas obsolète (doit passer par un shared_ptr le temps de l'utilisation)

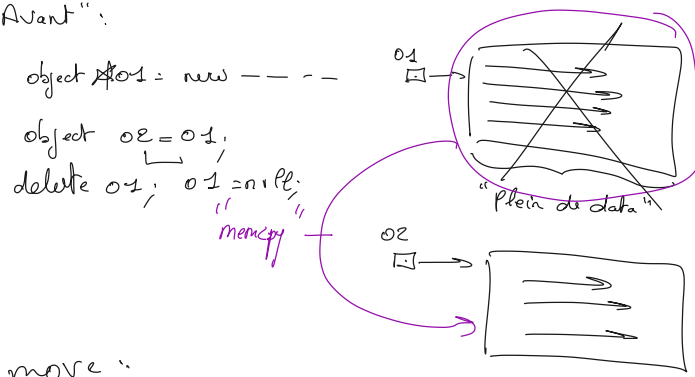
sp se libère car sortie du scope

gw est devenu obsolète

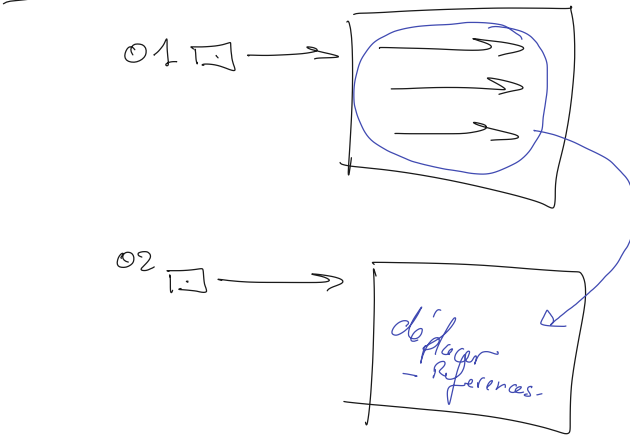
Move .ctor()
https://docs.microsoft.com/en-us/cpp/cpp/move-constructors-and-move-assignment-operators-cpp?view=msvc-170

Pourquoi le move "maintenant" ?

"Avant" :



move :



```
// ConsoleApplication1.cpp : Ce fichier contient la fonction 'main'. L'exécution du
programme commence et se termine à cet endroit.
//

#include <iostream>
#include <algorithm>
#include <vector>


#include "smartpointers.h"

void mref(int& i)
{
    i++;
}

inline void m(int i)    // Le code est directement injecté au niveau de l'appelant sans passer
par la pile
{
    int p;
    i++;
}

void TestSePresenter();

bool mysortcriteria(float a, float b)
{
    return (std::abs(a) < std::abs(b));
}

void abssort(float* x, unsigned n) {
    // Méthode traditionnelle : implique l'écriture d'une fonction externe utilisée que pour ce
cas de figure ( ce qui est dommage )
    std::sort(x, x + n, &mysortcriteria);

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

```
/*
int main()
{
   int i = 10;
   int* pi = 0;
   pi = &i;    // adresse de i
   // modifier le contenu du i :
   *pi = 11;   // alourdissement de la syntaxe à cause de la notation de pointeurs ( "contenu
de" )
   int& ri=i;  // je déclare une référence sur une variable ( forcément ) ne peut être vide,
nulle, ou constante
   ri = 12;

   mref(i);    // i sera modifiée
   m(i);       // i n'est pas modifiée
   m(i);       // i n'est pas modifiée
   m(i);       // i n'est pas modifiée

   std::cout << "Hello World!\n";
   for (;;)
   {
      char* p;
      try {
         p = new char[102400];
         // probabilité de planter :
            throw 0;
         delete[] p; // Fuite mémoire
      }
      catch (int code) {
         delete[] p; // Fuite mémoire
         std::cout << "Erreur gérée";
      }
   }

   auto v = 10;


   // Utilisation de la syntaxe C++ moderne pour créer une collection de 5 int,
   // l'allouer de façon moderne en respect du pattern RAII ( make_unique )
   // pNums devenant une collection de 5 int
   auto pNums = std::make_unique<std::vector<int>>(5);
   // déclarer une variable de type expression lambda est "compliqué"
   // du coup on s'autorise à utiliser auto pour obtenir une variable a qui est une expression
lambda
   // ( le code d'une fonction qui ici n'est pas appellée, mais qui le sera plus tard )
   auto a = [ptr = std::move(pNums)]()
   {
      // use ptr
   };

   auto const indice = { 5, 10 };
   std::string const phrase{ "Bonjour tout le monde !" };

//    phrase.
}
*/
```

```
// function template
#include <iostream>
#include <cassert>
using namespace std;

template <class T>
T GetMax(T a, T b) {
   T result;
   result = (a > b) ? a : b;
   return (result);
}

int GetMaxInt(int a, int b) {
   int result;
   result = (a > b) ? a : b;
   return (result);
}
long GetMaxLong(long a, long b) {
   long result;
   result = (a > b) ? a : b;
   return (result);
}

class widget
{
private:
   int* data;
public:
   explicit widget(const int size) { data = new int[size]; } // acquire
   ~widget() {
      std::cout << "Releasing";
      delete[] data;
   } // release
   void  do_something() const {}
};

void functionUsingWidget() {
   widget w(1000000);   // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data member

   throw 0;

   w.do_something();

} // automatic destruction and deallocation for w and w.data

struct B
{
   virtual ~B() = default;

   virtual void bar() { std::cout << "B::bar\n"; }
};

struct D : B
{
   D() { std::cout << "D::D\n"; }
   ~D() { std::cout << "D::~D\n"; }

   void bar() override { std::cout << "D::bar\n"; }
};
```

```
void TestUniquePtr()
{
    std::unique_ptr<D> p = std::make_unique<D>();

    // Transfer ownership to `pass_through`,
    // which in turn transfers ownership back through the return value
    std::unique_ptr<D> q = std::move(p);

    // `p` is now in a moved-from 'empty' state, equal to `nullptr`
    assert(!p); // p en effet est non null, mais le compilateur emet un WARNING
}

int main() {
    /*
    try
    {
        functionUsingWidget();
    }
    catch( int i )
    {
        std::cout << "Erreur gérée";
    }

    */
    /*
    int i = 5, j = 6, k;
    long l = 10, m = 5, n;
    // Code paramétrique :
    k = GetMax<int>(i, j);
    n = GetMax<long>(l, m);
    // Equivaut à ( en C legacy ) :
    int k2 = GetMaxInt(i, j);
    long n2 = GetMaxLong(l, m);

    cout << k << endl;
    cout << n << endl;
    return 0;
    */
    /*
    int *pi = new int;

    // 1 :
    delete pi;  // dangereux.( je suis sur que pi sera forcément alloué et que ce code ne
s'executera qu'une seule fois )

    // 2 : Meilleure pratique : je libère ce qui est libérable ( lourd en code )
    if (pi != 0)
    {
        delete pi;
        pi = 0;
    }
    */
    TestUniquePtr();

    TestSmartPointers();    // dans le fichier à côté
}
```

```
#include <iostream>
#include <cstring>


// Ce code est compilable avec un compilateur C ( non c++ )

// Déclaration de mes membres :

struct Personne
{
 char nom[100], prenom[100];
};

// Des "méthodes" associées à cette "classe" ( structure )

void sePresenter(Personne* self, char* NomPrenom )
{
 //sprintf(NomPrenom, "%s %s", self->nom, self->prenom);
}

void f()
{}

void f2()
{}

int fonction(int a, int phrase)  //Une jolie fonction
{
 //blablabla
 return 0;
}

// Test Unitaire :
void TestSePresenter()
{
 char nomcomplet[250];

 Personne moi;
 //strcpy(moi.nom, "Pecqueur");
 //strcpy(moi.prenom, "Philippe");

 sePresenter(&moi, nomcomplet);
 std::cout << nomcomplet;

 void (*pf)();
 pf = f;
 pf = f2;
 (*pf)();
}
```

```
using namespace std;

#include <memory> // utilisation des smart pointers

#include "smartpointers.h"
#include <iostream>
#include <thread>
#include <mutex>

class Person
{
public:
 int Age;

 Person()
 {
  cout << ".ctor()\n";
 }

 ~Person()
 {
  cout << ".dtor()\n";
 }

};

struct Base
{
 Base() { std::cout << "  Base::Base()\n"; }
 // Note: non-virtual destructor is OK here
 ~Base() { std::cout << "  Base::~Base()\n"; }
};

struct Derived : public Base
{
 Derived() { std::cout << "  Derived::Derived()\n"; }
 ~Derived() { std::cout << "  Derived::~Derived()\n"; }
};

void thr(std::shared_ptr<Base> p)
{
 std::this_thread::sleep_for(std::chrono::seconds(1));
 std::shared_ptr<Base> lp = p; // thread-safe, even though the
         // shared use_count is incremented
 {
  static std::mutex io_mutex;
  std::lock_guard<std::mutex> lk(io_mutex);
  std::cout << "local pointer in a thread:\n"
   << "  lp.get() = " << lp.get()
   << ", lp.use_count() = " << lp.use_count() << '\n';
 }
}

void TestSmartPointers()
{
 // UNIQUE POINTERS :
 unique_ptr<Person> p1 = make_unique<Person>(); // Alloue un Person
 unique_ptr<Person> p2 = unique_ptr<Person>(); // N'alloue pas un Person

 // Utilisation de l'objet derrière le smart pointer.

 Person* pp = p1.get();
 pp->Age = 10;

 Person* pp2 = p1.get();
 cout << "Age:" << pp2->Age << "\n";

 unique_ptr<Person> p3 = move(p1);
 Person* pp3 = p3.get();
 cout << "Age:" << pp3->Age << "\n";
 /*
 Person* pp4 = p1.get(); // PAS BON !!
 cout << "Age:" << pp4->Age << "\n";
 */

 // SHARED POINTERS :
 std::shared_ptr<Base> p = std::make_shared<Derived>();

 std::cout << "Created a shared Derived (as a pointer to Base)\n"
  << "  p.get() = " << p.get()
  << ", p.use_count() = " << p.use_count() << '\n';
 std::thread t1(thr, p), t2(thr, p), t3(thr, p);
 p.reset(); // release ownership from main
 std::this_thread::sleep_for(std::chrono::milliseconds(1500));
 std::cout << "Shared ownership between 3 threads and released\n"
  << "ownership from main:\n"
  << "  p.get() = " << p.get()
  << ", p.use_count() = " << p.use_count() << '\n';
 t1.join(); t2.join(); t3.join();
 std::cout << "All threads completed, the last one deleted Derived\n";
}
```