

Advanced C++ programming

♦ Introduction to C++

- C++: from C and beyond
- Classes, objects and lifetime (vs. JAVA)
- Oriented-Object Programming (inheritance, polymorphism)

♦ Memory management & object manipulation

- References, operators, « copy » object construction
- « move » object construction, lambda functions

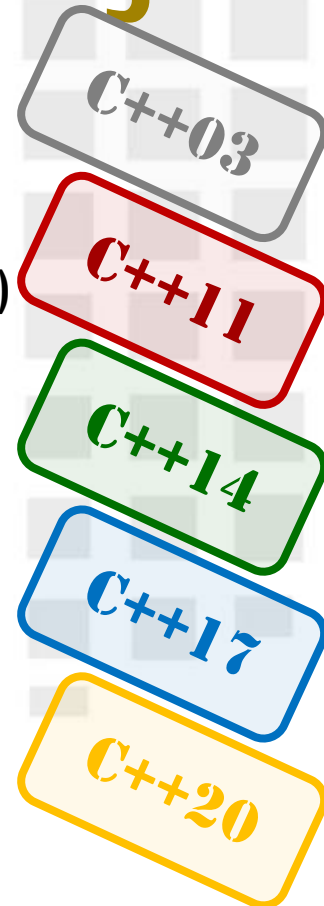
♦ Template vs OO programming

- Template functions and classes

♦ The Standard Template Library

- Containers, iterators and algorithms
- Using sequence & associative containers ...

♦ Smart pointers (STL & Boost)



Slot 9

Problématique

- Comment gérer la durée de vie des objets créés dynamiquement ?
 - Création : **explicite** ► allocation de mémoire sur le tas (via **new**)
 - Destruction : **doit aussi être explicite** (lorsque la référence sur l'objet sort de la portée) ► libération explicite de la mémoire (via **delete**)
 - Que se passe-t-il si on oublie la libération ?
 - Le destructeur de l'objet pointé n'est jamais appelé :
 - handles non libérés
 - fuites de mémoire
 - ...

```
int main () {  
    Point* pt = new Point (10 , 20) ;  
  
    // Call the function 'fn'  
    fn (pt) ;  
  
    // Free the heap memory for  
    // point 'pt'  
    // if not => memory leaks  
    delete pt ;  
  
    return 0 ;  
}
```

```
int main () {  
    Point* pt1 = getPoint() ;  
    Point* pt2 = makePoint() ;  
    Point* pt3 = computePoint() ;  
  
    ...  
  
    // Which ones ???  
    delete pt1 ;  
    delete pt2 ;  
    delete pt3 ;  
  
    return 0 ;  
}
```



Faire appel à
des « **smart pointers** »



Smart pointers ?

C++ 11 / C++ 14

`tr1::` (si C++0x)

Smart pointers
(`shared_ptr`)

Générateurs aléatoires
(`poisson`, `gamma`, ...)

Mathématiques
(`bessel`, `laguerre`, ...)

Expressions
régulières

Gestion des entités
(`binders`, `wrapper`, ...)

Nouveaux types
(`tuple`, `hash table`, ...)

Meta-programming
(`type_traits`, ...)

Boost

C++ 98

Internationalisation
(`wchar_t`, `wchar_ts`)

C++
(Template)

Standard Template
Library (STL)

C++
(Objet)

C89

Mathématiques
(`complex`, `valarray`)

Gestion des flots
(`iostream`)

Hiérarchisation
des exceptions

Graph Library

Meta Programming Library (MPL)





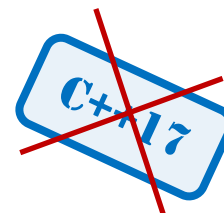
Définition (1/2)

- Un « smart pointer » ~ un pointeur ++
 - Il s'agit juste d'une classe encapsulant un pointeur !
 - Même comportement qu'un pointeur
 - Déréférencement (*), indirection (->)
 - Proposer des solutions à certains problèmes induits par la notion de pointeur
 - Ne pas permettre d'avoir un pointeur sur un objet qui a été détruit
 - Toujours libérer un objet auquel on ne peut plus accéder
- Implémentation de différents « smart pointers »
 - adaptés à différentes problématiques
 - avec des comportements différents ➡

auto_ptr, shared_ptr,
unique_ptr



Définition (2/2)




• Exemple de « smart pointer » : `auto_ptr`

- Le seul « smart pointer » standard proposé jusqu'à C++11
- Une classe encapsulant un pointeur !
- Le côté « smart » :
son destructeur se charge
de libérer l'objet pointé !

```
template <class T> class auto_ptr {  
    T* ptr ;  
  
public:  
    explicit auto_ptr(T* p = 0) : ptr(p) {}  
    ~auto_ptr() {delete ptr ; }  
    T& operator*() {return *ptr ; }  
    T* operator->() {return ptr ; }  
    // ...  
} ;
```

• Utilisation de `auto_ptr`



```
void foo() {  
    MyClass* p(new MyClass) ;  
    p->DoSomething() ;  
    delete p ;  
}
```



```
void foo() {  
    auto_ptr<MyClass> p(new MyClass) ;  
    p->DoSomething() ;  
}
```



Manipulation (1/3)

- **Comportements fondamentaux d'un smart pointer**
 - Que se passe-t-il ?
 - lors de sa construction & destruction \Rightarrow éviter les fuites de mémoire
 - lors de sa copie & affectation \Rightarrow notion de **propriété**
 - Que se passe-t-il avec l'objet pointé après une copie ?
 \Rightarrow copie en surface ? copie en profondeur ? duplication des ressources ? ...
 - Que se passe-t-il avec l'objet pointé après une affectation ?
 \Rightarrow duplication ? pointeurs multiples sur ce même objet (compteur de référence, ...) ?
 - lors de son déréférencement \Rightarrow simple accès, accès avec copie retardée, ...
- **Point clé : la notion de propriété**
 - Le smart pointer est-il **propriétaire** ou non l'objet pointé ?
 - Si oui : à sa destruction, le smart pointer est responsable de la destruction de l'objet pointé



Manipulation (2/3)

• Comportement de `auto_ptr`

• Règles de propriété :

- Propriété **exclusive** : pas plus d'un `auto_ptr` pointant sur le même objet O
 - Éviter la destruction multiple d'un objet référencé plusieurs fois !
- Implémentation : **transfert de la propriété** lors de l'affectation / copie
 - Autre choix possible : la duplication \Rightarrow problème : nécessité de constructeurs virtuels (`auto_ptr<T>` doit supporter des objets d'un type dérivé de **T**)

```
// Ne jamais passer d'auto_ptr par valeur :  
void print (ostream& s , auto_ptr<Point> P) {  
    s << *p ; }  
  
int main () {  
    auto_ptr<Point> I (new Point(10,20)) ;  
  
    // I est passé par valeur => copie  
    print (cout , I) ;  
    // La propriété de Point(10,20) a été donnée  
    // à la variable locale « P » de print :  
    // ici, I ne référence donc plus un objet  
    // valide !  
}
```



```
// Affectation d'auto_ptr :  
auto_ptr<Point> P1 (new Point(10,20)) ;  
auto_ptr<Point> P2 (new Point(30,40)) ;  
  
P1 = P2 ; // le Point(10,20) est détruit  
          // avant de prendre la propriété  
          // du Point(30,40)  
          // P1 -> Point(30,40)  
          // P2 -> NULL
```

Passage par référence : OK

```
void print (ostream& s , auto_ptr<Point>& P)
```



Manipulation (3/3)

• Remarques générales sur les « smart pointers »

- Multiples implémentations, comportements spécifiques
- Toujours savoir quelle est la notion de propriété
- Attention au polymorphisme :



- `smart_pointer<Circle>` n'hérite pas de `smart_pointer<Shape>`
- même si `Circle` hérite de `Shape`

• Comportement spécifique à `auto_ptr`

- Transfert de propriété



⇒ on ne peut pas stocker pas d'objets référencés par un smart pointer `auto_ptr` dans un container STL !

`auto_ptr` est remplacé par `unique_ptr` à partir de C++11



std::unique_ptr<T> (1/3)



cppreference.com

Create account

Search

Page

Discussion

View

Edit

History

C++ Utilities library Dynamic memory management **std::unique_ptr**

std::unique_ptr

Defined in header <memory>

```
template<
    class T,
    class Deleter = std::default_delete<T> (1) (since C++11)
> class unique_ptr;

template <
    class T,
    class Deleter (2) (since C++11)
> class unique_ptr<T[], Deleter>;
```



std::unique_ptr is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope.

The object is disposed of, using the associated deleter when either of the following happens:

- the managing unique_ptr object is destroyed
- the managing unique_ptr object is assigned another pointer via `operator=` or `reset()`.

The object is disposed of, using a potentially user-supplied deleter by calling `get_deleter()(ptr)`. The default deleter uses the `delete` operator, which destroys the object and deallocates the memory.

A unique_ptr may alternatively own no object, in which case it is called *empty*.

There are two versions of std::unique_ptr:

1. Manages a single object (e.g. allocated with `new`)
2. Manages a dynamically-allocated array of objects (e.g. allocated with `new[]`)

The class satisfies the requirements of *MoveConstructible* and *MoveAssignable*, but of neither *CopyConstructible* nor *CopyAssignable*.



std::unique_ptr<T> (2/3)

C++11

TP
16.2

• Transfert de propriété explicite



- auto_ptr : via « copy » mais de façon cachée
- unique_ptr : uniquement via « move » et **explicitement**

```
// Création d'un unique_ptr pour un nouvel objet  
std::unique_ptr<Shape> up1 (new Shape) ;
```

```
// Tentative de copie de l'objet :  
// ERREUR de compilation  
std::unique_ptr<Shape> up3 (up1) ;  
  
// Transfert de propriété à up2 : OK  
std::unique_ptr<Shape> up2 (std::move(up1)) ;
```

```
// Création d'un autre unique_ptr  
std::unique_ptr<Shape> up2 ;
```

```
// Tentative d'affectation :  
// ERREUR de compilation  
up2 = up1
```

```
// Transfert de propriété à up2 : OK  
up2 = std::move(up1) ;
```

```
// up1 ne référence plus aucun objet (erreur à l'exécution si tentative d'utilisation)  
// L'objet Shape sous-jacent ne sera détruit qu'une seule fois (à la disparition de up2)
```

C++14

```
// Création d'un unique_ptr pour un nouvel objet  
std::unique_ptr<Shape> up1 = std::make_unique<Shape>() ;
```



std::unique_ptr<T> (3/3)

• Using "std::unique_ptr" with STL containers

```
int main() {

    std::unique_ptr<A> pa1 = std::make_unique<A> (10) ;
    cout << "pa1 : " << pa1 << " = " << pa1->k << endl ;

    std::unique_ptr<A> pa2 = std::move(pa1) ;
    cout << "Moving pa1 to new unique_ptr pa2 : " << endl ;
    cout << "pa1 : " << pa1 << endl ;
    cout << "pa2 : " << pa2 << " = " << pa2->k << endl ;

    std::vector<std::unique_ptr<A>> vp ;
    vp.push_back(std::move(pa2)) ;
    cout << "Moving pa2 into an empty vector of unique_ptr<A> : " << endl ;
    cout << "pa2 : " << pa2 << endl ;

    auto it = vp.begin() ;
    cout << "Retrieving object A in the vector : " << endl ;
    cout << "A in vector : " << *it << " = "
        << (*it)->k << endl ;

    cout << "return : " << endl ;
    return 0 ;
}
```



```
struct A {

    int k = 0 ;

    A(int i) ;
    A(const A& a) ;
    A(A&& a) ;
    A& operator= (const A& a) ;
    A& operator= (A&& a) ;
    ~A() ;
};
```



```
(10)   ctor : 0x1406067d0
pa1 : 0x1406067d0 = 10
```



```
Moving pa1 to new unique_ptr pa2 :
pa1 : 0x0
pa2 : 0x1406067d0 = 10
```

```
Moving pa2 into an empty vector of unique_ptr<A> :
pa2 : 0x0
```

```
Retrieving object A in the vector :
A in vector : 0x1406067d0 = 10
```

```
return :
~      dtor [10] 0x1406067d0
```



`std::shared_ptr<T>` (1/4)



- Un autre smart pointer : `shared_ptr`

- ▶ Règle de propriété : propriété **partagée**.

- Plusieurs pointeurs pointent sur un même objet O et chaque pointeur en partage la propriété
- Implémentation :
 - **Compteur de références** pour savoir combien de pointeurs courants sur l'objet
 - Le dernier pointeur vivant sur l'objet est responsable de la destruction de l'objet

- Un dernier smart pointer : `weak_ptr`

- ▶ Règle de propriété : aucune

- Pointe sur un objet référencé par un `shared_ptr` sans toucher au compteur de références
 - éviter les cycles de dépendances où deux `shared_ptr` se « pointent » mutuellement
 - éviter le coût d'incrémenter le compteur de références si la durée de vie du `weak_ptr` est obligatoirement inférieure à celle du `shared_ptr` sous-jacent



`std::shared_ptr<T>` (2/4)

- Autre propriété de `shared_ptr`

- Rappel : `shared_ptr<Circle>` n'hérite pas de `shared_ptr<Shape>`
- Mais conversion implicite offerte par `shared_ptr` :
 - de `shared_ptr<Circle>` vers `shared_ptr<Shape>`
 - si `Circle*` peut être implicitement converti en `Shape*`
(OK si `Circle` hérite de `Shape`)
- Aucun souci de stockage dans les conteneurs de la STL

- Exemples de déclaration

```
#include <memory>
```

```
// Déclaration (2 manières)
std::shared_ptr<Point> P1_ptr (new Point(10,20)) ;
std::shared_ptr<Point> P2_ptr = std::make_shared<Point>(10,20) ;

// Réutilisation du point « P2 »
P2_ptr.reset (new Point(30,40)) ;
```

► Le compteur de référence est initialisé à 1



std::shared_ptr<T> (3/4)

- Passage shared_ptr → simple pointeur

```
boost::shared_ptr<Point> P (new Point(20,40)) ;  
  
// Récupération d'un pointeur sur l'objet pointé  
Point* ptr = P.get() ;
```

- Passage simple pointeur → shared_ptr (interdit)

- Connaissance & maîtrise de la durée de vie de l'objet pointé par le « smart pointer » impossibles

► Pour éviter le comportement suivant :



```
void some_function (Point *f) {  
    // fP : compteur de références = 1  
    std::shared_ptr<Point> fP(f) ;  
    ...  
  
    // fP : compteur de références = 0  
    // => Objet pointé par fP détruit  
}
```

```
// Corps du programme  
Point *ptP = new Point(10,20) ;  
  
some_function (ptP) ;  
// Problème si l'objet pointé par ptP a été détruit  
// => erreur lors de l'appel suivant  
  
ptP->something () ;
```



std::shared_ptr<T> (4/4)

```
#include <algorithm>
#include <memory>
```

```
#include <vector>
#include <set>
#include <iostream>
```

```
struct Foo {
    int x ;

    Foo (int _x) : x(_x) {}
    ~Foo() {
        std::cout << "~Foo on a Foo with x=" << x << "\n" ;
    }
};
```

```
using FooPtr = std::shared_ptr<Foo> ;
```

```
struct FooPtrOps {
    bool operator()(const FooPtr& a, const FooPtr& b) {
        return a->x > b->x ;
    }
    void operator()(const FooPtr& a) {
        std::cout << a->x << " " ;
    }
};
```

```
std::for_each (foo_vector.begin(), foo_vector.end(), FooPtrOps()) ;
std::for_each (foo_set1.begin(), foo_set1.end(), FooPtrOps()) ;
std::for_each (foo_set2.begin(), foo_set2.end(),
    [](const FooPtr& p) { std::cout << p->x << " " ; }
) ;

return 0 ;
}
```

```
int main() {
    std::vector<FooPtr>      foo_vector ;
    std::set<FooPtr, FooPtrOps> foo_set1 ;
    std::set<FooPtr>        foo_set2 ;

    FooPtr foo_ptr21 = std::make_shared<Foo>(2) ;
    foo_vector.push_back (foo_ptr21) ;
    foo_set1.insert (foo_ptr21) ;
    foo_set2.insert (foo_ptr21) ;

    FooPtr foo_ptr1 = std::make_shared<Foo>(1) ;
    foo_vector.push_back (foo_ptr1) ;
    foo_set1.insert (foo_ptr1) ;
    foo_set2.insert (foo_ptr1) ;

    FooPtr foo_ptr3 (new Foo(3)) ;
    foo_vector.push_back (foo_ptr3) ;
    foo_set1.insert (foo_ptr3) ;
    foo_set2.insert (foo_ptr3) ;

    FooPtr foo_ptr22 (new Foo(2)) ;
    foo_vector.push_back (foo_ptr22) ;
    foo_set1.insert (foo_ptr22) ;
    foo_set2.insert (foo_ptr22) ;
}
```

```
foo_vector: 2 1 3 2
foo_set1:   3 2 1
foo_set2:   2 3 1 2
~Foo on a Foo with x=2
~Foo on a Foo with x=1
~Foo on a Foo with x=3
~Foo on a Foo with x=2
```



Histoires de C++

Death and Memory (C++ Stories)

Scope will
kill me!

```
int x;
```

My master
will kill me!

```
std::unique_ptr<int> y;
```

I will die when
nobody knows
who I am

```
std::shared_ptr<int> t;
```

I hope someone
remembers
to kill me

```
int* z;
```

Kill me?
You don't even
know what I am!

```
void* q;
```

**I AM HARDWARE
I WILL OUTLIVE YOU
I WILL OUTLIVE
YOUR SOFTWARE
THERE IS NO DEATH**

```
uint8_t* r =  
(uint8_t*)0x0020;
```

2017 Ólafur Waage (@olafurw)
with thanks to Frank A. Krueger (@praeclearum)



Boost ... qu'est-ce donc ?



- Bibliothèques de classes, de fonctions, ...
 - Nombreux outils, algorithmes, opérations disponibles
 - Expressions régulières, gestion unifiée des threads, quaternions
 - Intégration avec Python, timers, quelques “smart pointers”
 - Bonne qualité de fonctionnement
 - Utilisation par de nombreux programmeurs
 - Plusieurs années de test
- Candidates à la standardisation
 - Plusieurs bibliothèques de Boost ont globalement servi de base aux nouveautés de C++11 dont
 - Gestion des threads (multi-plateforme)
 - Quelques “smart pointers” dont **shared_ptr**

```
#include <memory>
```



Perspectives



- **C++11/C++14/C++17/ ... ► « Modern C++ »**
 - « Smart pointers », lambda functions
 - « Thread » : accès au threading unifié
 - multi-plateforme : interface commune sur des bibliothèques dédiées
 - très intéressant sur des machines multi-cœur (↗ performances)
 - « Regex » : expressions régulières
 - « Numerics » : complexes, générateurs aléatoires, ...
- **Boost : ensemble de bibliothèques, répondant à des problèmes très variés**
 - « Serialization » : texte, binaire, XML, ...
 - « Math » : diverses bibliothèques mathématiques
 - PGCD, trigonométrie complexe, statistiques, loi de probabilités, ...
 - « uBLAS » : vecteurs, matrices (sparse, ...), algèbre linéaire, ...

A vous de jouer !

