Programming in Modern C++

Module M57: C++11 and beyond: Resource Management by Smart Pointers: Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Revisited Raw Pointers and discussed how to deal with the objects through raw pointer
- Introduced Smart pointers with typical interface and use
- Introduced some of the policies for smart pointer:
  - Different storage policies
  - Ownership Policies

Module Objectives

- To continue Discussions on various policies of smart pointers
  - Ownership Policies
  - Implicit Conversion policy
  - Null test policy
- To familiarize with Resource Management using Smart Pointers from Standard Library
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`
  - `auto_ptr`

Module Outline

Smart Pointers

Sources:

- Chapter 4. Smart Pointers: Effective Modern C++, Scott Meyers
  - Item 18: Use `std::unique_ptr` for exclusive-ownership resource management
  - Item 19: Use `std::shared_ptr` for shared-ownership resource management
  - Item 20: Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle
  - Item 21: Prefer `std::make_unique` and `std::make_shared` to direct use of `new`
- The Rule of The Big Three (and a half) – Resource Management in C++, 2014
- Smart pointer, wikipedia
- How to use C++ raw pointers properly?, Sorush Khajepor, 2020
- What is a C++ unique pointer and how is it used? smart pointers part I, Sorush Khajepor, 2021
- What is a C++ shared pointer and how is it used? smart pointers part II, Sorush Khajepor, 2021
- What is a C++ weak pointer and where is it used? smart pointers part III, Sorush Khajepor, 2021
- Lambdas: Smart Pointers, Jim Fix, Reed College

Smart Pointers

Smart Pointers: Recap

What is a Smart Pointer? (Recap Module 56)

- A Smart pointer is a C++ object
- Stores pointers to dynamically allocated (heap / free store) objects
- Improves raw pointers by implementing
  - Construction & Destruction
  - Copying & Assignment
  - Dereferencing:
    - `operator->`
    - unary `operator*`
- Grossly mimics raw pointer syntax and semantics

Typical Tasks of a Smart Pointer (Recap Module 56)

- Selectively *disallows unwanted* operations, that is, Address Arithmetic
- *Lifetime Management*
  - Automatically deletes dynamically created objects at appropriate time
  - On face of exceptions – ensures proper destruction of dynamically created objects
  - Keeps track of dynamically allocated objects shared by multiple owners
- *Concurrency Control*
- Supports **Idioms**: **RAII**: Resource Acquisition is Initialization Idiom and **RRID**: Resource Release Is Destruction
  - The idiom makes use of the fact that every time an object is created a constructor is called; and when that object goes out of scope a destructor is called
  - The constructor/destructor pair can be used to create an object that automatically allocates and initialises another object (known as the *managed object*) and cleans up the managed object when it (the *manager*) goes out of scope
  - This mechanism is generically referred to as **resource management**

Typical Smart Pointer Interface (Recap Module 56)


```
template<typename T> // Pointee type T
class SmartPtr {
public:
    // Constructible
    // No implicit conversion from Raw ptr
    explicit SmartPtr(T* pointee):
        pointee_(pointee) { }
    // Copy Constructible
    SmartPtr(const SmartPtr& other);
    // Assignable
    SmartPtr& operator=(const SmartPtr& other);
    // Destroys the pointee
    ~SmartPtr();
    // Dereferencing
    T& operator*() const { ... return *pointee_; }
    // Indirection
    T* operator->() const { ... return pointee_; }
private:
    T* pointee_; // Holding the pointee
};
```

The Smartness Charter (Recap Module 56)

- It always points either to a valid allocated object or is NULL
- It deletes the object once there are no more references to it
- Fast: Preferably zero de-referencing and minimal manipulation overhead
- Raw pointers to be only explicitly converted into smart pointers. Easy search using grep is needed (it is unsafe)
- It can be used with existing code
- Programs that do not do low-level stuff can be written exclusively using this pointer. No Raw pointers needed
- Thread-safe
- Exception safe
- It should not have problems with circular references
- *Programmers would not keep raw pointers and smart pointers to the same object*

Policies (Recap Module 56)

- The charter is managed through a set of policies that bring in flexibility and leads to different flavors of smart pointers
- Major policies include:
  - Storage Policy
  - Ownership Policy
  - Conversion Policy
  - Null-test Policy

Ownership Policy: Exclusive and Shared (Recap Module 56)

Exclusive Ownership

- **Exclusive Ownership Policy**
- Transfer ownership on copy
- On Copy: Source is set to NULL
- On Delete: Destroy the pointee Object

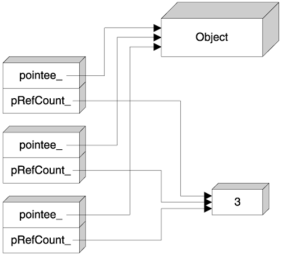
- `std::auto_ptr` (C++03), `std::unique_ptr` (C++11)
- Coded in: C-Ctor, `operator=`

Shared Ownership

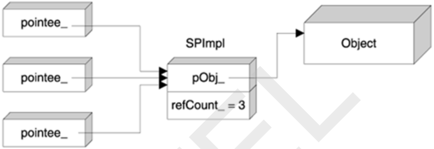
- **Shared Ownership Policy**
- Multiple Smart pointers to same pointee
- On Copy: Reference Count (`RC`) incremented
- On Delete: `RC` decremented, if `RC > 0`. Pointee object destroyed for `RC = 0`
- `std::shared_ptr`, `std::weak_ptr` (C++11)

- Coded in: Ctor, C-Ctor, `operator=`, Dtor

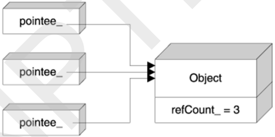
Ownership Policy: Exclusive and Shared (Recap Module 56)



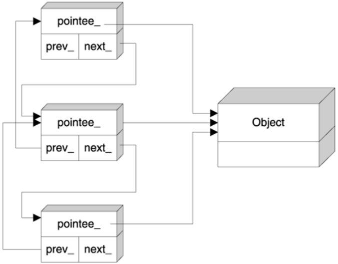
Extra pointer & non-intrusive counter



Extra indirection & non-intrusive counter



Intrusive counter



Reference linking

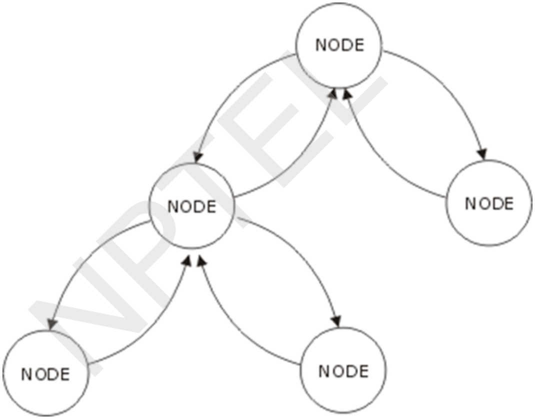
- **Non-Intrusive Counter**
  - Addl. count ptr per smart ptr
  - Count in Free Store
  - Allocation of Count may be slow as it is too small (may be improved by global pool)
- **Non-Intrusive Counter**
  - Addl. count ptr removed
  - But addl. access level means slower speed
- **Intrusive Counter**
  - Most optimized RC smart ptr
  - Cannot work for an already existing design
  - Used in Component Object Model (COM)
- **Reference Linking**
  - Overhead of two addl. ptrs
  - Doubly-linked list for constant time:
    - ▷ For Append, Remove & Empty detection

Smart Pointers: Ownership Policy

Ownership Policy: Reference Management: Shortcoming

- Circular / Cyclic Reference
  - Object A holds a smart pointer to an object B. Object B holds a smart pointer to A. Forms a cyclic reference
    - Typical for a Tree: Child & Parent pointers
  - Cyclic references go undetected
    - Both the two objects remain allocated forever
    - Resource Leak occurs
  - The cycles can span multiple objects

Ownership Policy: Cyclic Reference: Hack



- Use *Smart pointer* (`std::shared_ptr`) from Parent to Child: *Data Structure* Pointers
- Use *Weak pointer* (`std::weak_ptr`) from Child to Parent: *Algorithm* Pointers

Ownership Policy: Cyclic Reference: Solution

- Maintain two flavors of RC Smart Pointers
  - *Strong* pointers that really link up the data structure (Child / Sibling Links). They behave like regular RC. `std::shared_ptr`
    - These are **Ownership** pointers
  - *Weak* pointer for cross / back references in the data structure (Parent / Reverse Sibling Links). `std::weak_ptr`
    - These are **Observer** pointers
- Keep two reference counts:
  - One for strong pointers, and
  - One for weak pointers
- While dereferencing a weak pointer, check the strong reference count:
  - If it is zero, return NULL. As if, the object is gone

Smart Pointers: Conversion Policy

Implicit Conversion Policy

- Consider

```
void Fun(Something* p); ... // For maximum compatibility this should work
SmartPtr<Something> sp(new Something);
Fun(sp); // OK or error?
```

- User-Defined Conversion (cast)

```
template<typename T>class SmartPtr { public:
    operator T*() { return pointee_; } // user-defined conversion to T*
};
```

- **Pitfall**: This following compiles okay and defeats the purpose of the smart pointer

```
SmartPtr<Something> sp; ... // Undetected semantic error at compile time
delete sp; // Compiler passes this by casting to raw pointer
```

- **No conversion allowed in library**. No `operator T*() const noexcept;` is even provided. Use `get()` to obtain the raw pointer from `unique_ptr` or `shared_ptr`

Smart Pointers: Null-test Policy

Null Test Policy

- How to check if the smart pointer is null? Expect the following to work?

```
SmartPtr<Something> sp1, sp2;
Something* p; ...

if (sp1)       // Test 1: direct test for non-null pointer ...
if (!sp1)      // Test 2: direct test for null pointer ...
if (sp1 == 0) // Test 3: explicit test for null pointer ...
```

- Without implicit conversion to to raw pointers, these cannot work
- Overloading `bool operator!() { return pointee_ == 0; }` would pass `Test 2`, would need `Test 1` to be written as `if(!!sp)`, and fail `Test 3`
- The library provides `explicit operator bool() const noexcept;` for the purpose in `unique_ptr` and `shared_ptr`
- `Test 1`, `Test 2` and `Test 3` work

Resource Management

Sources:

- Chapter 4. Smart Pointers: Effective Modern C++, Scott Meyers
  - Item 18: Use `std::unique_ptr` for exclusive-ownership resource management
  - Item 19: Use `std::shared_ptr` for shared-ownership resource management
  - Item 20: Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle
  - Item 21: Prefer `std::make_unique` and `std::make_shared` to direct use of `new`
- Smart Pointer in C++ Standard Library
  - `std::unique_ptr`, cppreference
  - `std::shared_ptr`, cppreference
  - `std::weak_ptr`, cppreference
  - `std::auto_ptr`, cppreference
- The Rule of The Big Three (and a half) – Resource Management in C++, 2014

Resource Management

Resource Management

- Smart pointers enable automatic, exception-safe, object lifetime management
- The various Pointers are:
  - `unique_ptr`: smart pointer with unique object ownership semantics
  - `shared_ptr`: smart pointer with shared object ownership semantics
  - `weak_ptr`: weak reference to an object managed by `std::shared_ptr`
  - `auto_ptr`: smart pointer with strict object ownership semantics
- All these are Defined in header `<memory>`
- First three pointers are included in C++11 where as last one is as in C++03

Resource Management: `std::unique_ptr`

Sources:

- `std::unique_ptr`, cppreference

std::unique_ptr


```
// Managing a single object. Deleter may be use supplied or default delete
template<typename T, typename Deleter = std::default_delete <T> >
class unique_ptr;

// Managing an array of object
template<typename T, typename Deleter>
class unique_ptr<T[], Deleter>;
```

- It retains sole ownership of an object through a pointer and destroys that object when the `unique_ptr` goes out of scope
- No two `unique_ptr` instances can manage the same object
- The raw pointer to the managed object can be obtained by `get()`
- The object is destroyed and its memory deallocated when:
  - The managing `unique_ptr` object is destroyed, or
  - The managing `unique_ptr` object is assigned another pointer via `operator=` or `reset()`
- The ownership can also be relinquished by `release()` which returns the raw pointer of the managed object

std::unique_ptr

- The object is destroyed using a potentially user-supplied deleter by calling `Deleter(ptr)`
- A `unique_ptr` may alternatively own no object (managed object pointer is `nullptr`), in which case it is called *empty*
- There are two versions of `std::unique_ptr`:
  - Manages the lifetime of a single object (for example, allocated with `new`)
  - Manages the lifetime of a dynamically-allocated array of objects (for example, allocated with `new[]`)
- Typical uses of `std::unique_ptr` include:
  - exception safety to classes and functions that handle objects with dynamic lifetime, by guaranteeing deletion
  - ownership of uniquely-owned objects with dynamic lifetime into functions
  - ownership of uniquely-owned objects with dynamic lifetime from functions
  - element type in move-aware containers, such as std::vector.

std::unique_ptr: Example


```cpp
#include <iostream>
#include <memory>
 struct Foo {
    Foo()      { std::cout << "Foo::Foo\n";  }
    ~Foo()     { std::cout << "Foo::~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n";  }
};
void f(const Foo &) { std::cout << "f(const Foo&)\n"; }
int main() {
    std::unique_ptr<Foo> p1 = std::make_unique<Foo>(); // (C++14) p1 owns Foo. // Foo::Foo
//  std::unique_ptr<Foo> p1(new Foo);                  // (C++11) p1 owns Foo. // Foo::Foo
    if (p1) p1->bar(); // Foo::bar
    {
        std::unique_ptr<Foo> p2(std::move(p1)); // now p2 owns Foo
        f(*p2); // f(const Foo&)
        p1 = std::move(p2); // ownership returns to p1
        std::cout << "destroying p2...\n"; // destroying p2...
    }
    if (p1) p1->bar(); // Foo instance is destroyed when p1 goes out of scope. // Foo::bar
} // Foo::~Foo
```

Resource Management: std::shared_ptr

Sources:

- std::shared_ptr, cppreference

std::shared_ptr

`template<typename T> class shared_ptr;`

- It retains shared ownership of an object through a pointer. Several `shared_ptr` objects may own the same object
- Object is destroyed and its memory deallocated when either of the following happens:
  - the last remaining `shared_ptr` owning the object is destroyed
  - the last remaining `shared_ptr` owning the object is assigned another pointer via `operator=` or `reset()`
- The object is destroyed using delete-expression or a custom deleter that is supplied to `shared_ptr` during construction
- The raw pointer to the managed object can be obtained by `get()`
- We can get the number of managed objects by invoking `use_count()`
- A `shared_ptr` can share ownership of an object while storing a pointer to another object
- It may also own no objects, in which case it is called *empty*
- All specializations of `shared_ptr` meet the requirements of `CopyConstructible`, `CopyAssignable`, and `LessThanComparable` and are contextually convertible to `bool`

std::shared_ptr: Example


```
#include <iostream>
#include <memory> // We need to include this for shared_ptr
using namespace std;
int main() {
    shared_ptr<int> p1 = make_shared<int>(); // Creating through make_shared
    *p1 = 78; // Set a value for the managed object
    cout << "p1 = " << *p1 << endl; // Access the value from managed object: p1 = 78
    cout << "p1 RC = " << p1.use_count() << endl; // Show RC: p1 RC = 1
    shared_ptr<int> p2(p1); // Second shared_ptr points to same pointer. RC = 2
    cout << "p2 RC = " << p2.use_count() << endl; // Show RC: p2 RC = 2
    cout << "p1 RC = " << p1.use_count() << endl; // Show RC: p1 RC = 2
    if (p1 == p2) { cout << "Same objects\n"; } // Compare ptrs: Same object
    cout<< "Reset p1 " << endl; // : Reset p1
    p1.reset(); // Reset the shared_ptr - it will not point to any object
    cout << "p1 RC = " << p1.use_count() << endl; // RC = 0: p1 RC = 0
    p1.reset(new int(11)); // Reset the shared_ptr with a new Pointer
    cout << "p1  RC = " << p1.use_count() << endl; // RC = 1: p1  RC = 1
    p1 = nullptr; // Assign nullptr to de-attach managed object
    cout << "p1  RC = " << p1.use_count() << endl; // RC = 0: p1  RC = 0
    if (!p1) { cout << "p1 is NULL" << endl; } // Test pointer: p1 is NULL
}
```

Resource Management: std::weak_ptr

Sources:

- std::weak_ptr, cppreference

std::weak_ptr


```
template<typename T> class weak_ptr;
```

- It holds a non-owning (*weak*) reference to an object that is managed by `std::shared_ptr`
- It must be converted to `std::shared_ptr` in order to access the referenced object
- `std::weak_ptr` models temporary ownership: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else
- It is used to track the object, and it is converted to `std::shared_ptr` to assume temporary ownership
- We can get the number of managed objects by invoking `use_count()`
- To check if the managed object is already deleted we can call `expired()`
- Also, `lock()` can be used to creates a `shared_ptr` from a `weak_ptr` to manage the referenced object
- If the original `std::shared_ptr` is destroyed at this time, the object's lifetime is extended until the temporary `std::shared_ptr` is destroyed as well
- **Note:** It is used to break circular references of `std::shared_ptr`. It cannot be used to access the managed object

std::weak_ptr: Example


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

std::weak_ptr<int> gw;

void f() {
    if (auto spt = gw.lock()) { // Has to be copied into a shared_ptr before usage
        std::cout << *spt << "\n";
    }
    else { std::cout << "gw is expired\n"; }
}
int main() {
    {
        auto sp = std::make_shared<int>(42); //
        gw = sp;
        f(); // 42
    }
    f(); // gw is expired
}
```

Resource Management: `std::auto_ptr`

Sources:

- `std::auto_ptr`, cppreference

std::auto_ptr


```
// Managing a single object in C++03. Deprecated in C++11, removed in C++17
template<typename T> class auto_ptr;

template<> class auto_ptr<void>;
```

- It retains sole ownership of an object through a pointer and destroys that object when the `auto_ptr` goes out of scope
- No two `auto_ptr` instances can manage the same object
- The raw pointer to the managed object can be obtained by `get()`
- The object is destroyed and its memory deallocated when:
  - The managing `auto_ptr` object is destroyed, or
  - The managing `auto_ptr` object is assigned another pointer via `operator=` or `reset()`
- The ownership can also be relinquished by `release()` which returns the raw pointer of the managed object
- **Never use `auto_ptr` in C++11 and beyond**

Resource Management: Summary of Smart Pointer Operations

Summary of Smart Pointer Operations

Member	unique_ptr	shared_ptr	weak_ptr	auto_ptr	Remarks
operator=	Y	Y	Y	Y	assigns the ptr[1]
release	Y	N	N	Y	returns a ptr to the managed object and releases the ownership
reset	Y	Y	Y	Y	replaces the managed object
swap	Y	Y	Y	N	swaps the managed objects
get	Y	Y	N	Y	returns a ptr to the managed obj
operator bool	Y	Y	N	N	checks if the stored ptr is not null
owner_before	N	Y	Y	N	owner-based ordering of smart pointers
operator*	Y	Y	N	Y	accesses the managed object
operator->	Y	Y	N	Y	accesses the managed object
operator[]	Y	Y (C++17)	N	N	indexed access to the managed array
use_count	N	Y	Y	N	returns the number of shared_ptr objects that manage the object
make_unique (C++14)	unique_ptr				creates a unique ptr that manages a new object
make_shared	shared_ptr				creates a shared pointer that manages a new object
static_pointer_cast	shared_ptr				applies static_cast to the stored ptr
dynamic_pointer_cast	shared_ptr				applies dynamic_cast to the stored ptr
const_pointer_cast	shared_ptr				applies const_cast to the stored ptr
reinterpret_pointer_cast	shared_ptr				applies reinterpret_cast to the stored ptr (C++17)
expired	weak_ptr				checks whether the referenced obj was already deleted
lock	weak_ptr				creates a shared_ptr that manages the referenced object

[1] transfers ownership from another auto_ptr

Resource Management: Binary Tree

Sources:

- `std::shared_ptr`, cppreference
- `std::weak_ptr`, cppreference

Binary Tree

- We show an example of a binary tree where every node keep a back pointer to its parent
- This leads to circularity and using `std::shared_ptr` we cannot clean up the tree
- So we use `std::shared_ptr` for the two children and `std::weak_ptr`
- Similar strategy may be employed in every case of circular data structure design
- Note that using `std::shared_ptr` for a binary tree may be an overkill as every node is held by its unique parent. So using `std::unique_ptr` for child and raw pointer for parent may be more optimal

Binary Tree using std::shared_ptr and std::weak_ptr


```
#include <iostream>
#include <memory>
using namespace std;
struct Node {
    shared_ptr<Node> lc;    // owns left child
    shared_ptr<Node> rc;    // owns right child
    weak_ptr<Node> parent; // observes parent
    int v;                  // Node value
    Node(int i = 0): v(i)
    { cout << "Node = " << v << endl; }
    ~Node()
    { cout << "~Node = " << v << endl; }
};
```

```
Node = 2
Node = 1
Node = 3
2 1 3
~Node = 2 // Nodes will not be cleaned
~Node = 3 // if parent is a shared_ptr
~Node = 1 // This is due to circularity
```

```
int main() {
    shared_ptr<Node> root = // root: 2
        make_shared<Node>(2);
    root->lc =              // left child: 1
        make_shared<Node>(1);
    root->rc =              // right child: 3
        make_shared<Node>(3);
    root->lc->parent = root; // back link
    root->rc->parent = root;

    shared_ptr<Node> p = root; // visit tree
    weak_ptr<Node> q;          // hold parent
    cout << p->v << ' ';
    p = p->lc;
    cout << p->v << ' ';
    q = p->parent;
    p = q.lock(); // weak to shared
    p = p->rc;
    cout << p->v << ' ';
    cout << endl;
}
```

Recommendations for Smart Pointers

Recommendations for Smart Pointers

- Scott Meyers in the his book Effective Modern C++ (Chapter 4. Smart Pointers) has made the following recommendations for the use of smart pointers for resource management:
  - Item 18: Use `std::unique_ptr` for exclusive-ownership resource management
  - Item 19: Use `std::shared_ptr` for shared-ownership resource management
  - Item 20: Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle
  - Item 21: Prefer `std::make_unique` and `std::make_shared` to direct use of `new`
- We strongly recommend the use of these for modern designs

Module Summary

- Discussed various policies of smart pointer
  - Ownership Policies
  - Implicit Conversion policy
  - Null test policy
- Familiarized with Resource Management using Smart Pointers from Standard Library
  - `unique_ptr`
  - `shared_ptr`
  - `weak_ptr`
  - `auto_ptr`