Programming in Modern C++

Module M56: C++11 and beyond: Resource Management by Smart Pointers: Part 1

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

Weekly Recap

- Learnt how Rvalue Reference works as a Universal Reference under template type deduction
- Understood the problem of forwarding of parameters and its solution using Universal Reference and `std::forward`
- Understood how Move works as an optimization of Copy
- Understood λ expressions (unnamed function objects) in C++ with
  - Closure Objects
  - Parameters
  - Capture
- Learnt different techniques without or with `std::function` to write and use non-recursive and recursive λ expressions in C++11 / C++14
- Introducing several class features in C++11 with examples
- Explained how these features enhance OOP, generic programming, readability, type-safety, and performance in C++11
- Introduced several features in C++11 for non-class types and templates with examples
- Familiarizes with important non-class types like enum class and fixed width integer
- Familiarized with important templates like variadic templates

Module Objectives

- Revisit Raw Pointers for resource management
- Introduce Smart pointers with typical interface and use
- Introduce the policies for smart pointer

Module Outline

Raw Pointers

Sources:

- How to use C++ raw pointers properly? Sorush Khajepor, 2020

Motivation: Resource Management

- Imbibe a culture to write **good** C++ code
  - *Correct*: Achieves the functionality
  - *Bug free*: Free of programming errors
  - *Maintainable*: Easy to develop and support
  - *High performance*: Fast, Low on memory
- Dynamic creation & destruction of objects is a *strength* and a *bugbear* of C / C++
- It needs manual *resource management* by the programmer. She / he has to control:
  - the allocation of memory for the object,
  - handle the object's initialisation and,
  - ensure that object was safely cleaned-up after use and its memory returned to heap
- This leads to C / C++ being an *unsafe*, *memory-leaking* language
- **Resource Management** frees the client from having to worry about the lifetime of the managed object, eliminating memory leaks and other problems in C++ code
- A resource could be any object that required dynamic creation/deletion – *memory*, *files*, *sockets*, *mutexes*, etc.
- Effective *Resource Management* is needed so that *dynamically managed objects can be managed as automatic object*

Raw Pointers: Operations

Raw Pointer Operations

- Dynamic Allocation (result of) or `operator&`
- Deallocation (called on)
- De-referencing `operator*`
- Indirection `operator->`
- Assignment `operator=`
- Null Test `operator!` `(operator== 0)`
- Comparison `operator==, operator!=, ...`
- Cast `operator(int), operator(T*)`
- Address Of `operator&`
- Address Arithmetic `operator+`, `operator-`, `operator++`, `operator--`, `operator+=`, `operator-=`
- Indexing (array) `operator[]`

What is a Raw Pointer?

- Typical use of Pointers
  - Essential – Link ('`next`') in a data structure
  - Inessential – Apparent programming ease
    - Passing Objects in functions: `void MyFunc(MyClass *);`
    - **Smart** expressions: `while (p) cout << *p++;`
- It is not a *First Class Object (FCO)* : An integer value is a FCO
- It Does not have a *Value Semantics* : Cannot COPY or ASSIGN at will
- It is a Weak Semantics for *Ownership* of pointee

Raw Pointers: Ownership Issue of Pointers

Ownership Issue of Pointers

- **Ownership Issue – ASSIGN problem**

```
    MyClass *p = new MyClass; // Create ownership
    p = 0;                    // Lose ownership
```

  Memory Leaks!

- **Ownership Issue – COPY problem**

```
    MyClass *p = new MyClass; // Create ownership
    MyClass *q = p;           // Copy ownership - no Copy Constructor! Performs shallow copy
    delete q;                 // Delete Object & Remove ownership
    delete p;                 // Delete Object - where is the ownership?
```

  Double Deletion Error!

- **Solution Of these: Exception Handling through try-catch**

```
void MyAction() {
    MyClass *p = 0;
    try {
        p = new MyClass;
        p->Function();
    }
    catch (...) {
        delete p; // Repeated code
        throw;
    }
    delete p;
}
```

Pitfall: Handling Ownership Issue of Pointers using `try-catch`

- Exceptional path dominates regular path

```
void MyDoubleAction() {
    MyClass *p = 0, *q = 0;
        try {
            p = new MyClass;
            p->Function(); // May throw
            q = new MyClass;
            q->Function(); // May throw
        }
        catch (...) {
            delete p; // Repeated code
            delete q; // Repeated code
            throw;
        }
    delete p;
    delete q;
}
```

How to deal with an Object?

So how do we deal with the objects to alleviate such problems?

- The object itself – *by value*
  - Performance Issue
  - Redundancy Issue
- As the memory address of the object – *by pointer*
  - Lifetime Management Issue
  - Code Prone to Memory Errors
- With an alias to the object – *by reference*
  - Good when null-ness is not needed
  - `const`-ness is often useful

Raw Pointers: Pointers vs. Reference

Pointers Vs. Reference

- Use Reference to Objects when
  - Null reference is not needed
  - Reference once created does not need to change
- Avoids
  - The security problems implicit with pointers
  - The (pain of) low level memory management (that is, delete)
- Without Pointer – Use Garbage Collection

Smart Pointers

Sources:

- The Rule of The Big Three (and a half) – Resource Management in C++, 2014
- 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

What is a Smart Pointer?

- 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

- 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 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

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

Typical Smart Pointer Use: Mimic a Raw Pointer

```
// A class for use
class MyClass {
public:
    void Function();
    // ...
};

// Create a smart pointer as an object
SmartPtr<MyClass> sp(new MyClass); // RAII: sp takes ownership of the instance

// As if indirecting the raw pointer
sp->Function(); // (sp.operator->())->Function()

// As if dereferencing the raw pointer
(*sp).Function();
```

Smart Pointers: Policies

The Smartness Charter

- 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

- 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

Smart Pointers: Policies: Storage Policy

3-Way Storage Policy

- The Storage Type (`T*`)
  - The type of pointee: Specialized pointer types possible: FAR, NEAR
  - By *default*, it is a raw pointer
  - Other Smart Pointers possible: When layered
- The Pointer Type (`T*`)
  - The type returned by `operator->`
    - Can be different from the storage type if proxy objects are used
- The Reference Type (`T&`)
  - The type returned by `operator*`

Smart Pointers: Policies: Ownership Policy

Ownership Management Policy

- Smart pointers are about *ownership of pointees*
- **Exclusive Ownership**
  - Every smart pointer has an exclusive ownership of the pointee
  - `std::unique_ptr`
  - Use Destructive Copy
- **Shared Ownership**
  - Ownership of the pointee is shared between Smart pointers – more than one smart pointer holds the same pointee
  - `std::shared_ptr`
  - `std::weak_ptr`
  - Track the Smart pointer references for lifetime
    - ▷ Reference Counting
    - ▷ Reference Linking

Ownership Policy: Destructive Copy

- Exclusive Ownership Policy
- Transfer ownership on copy
- Source Smart Pointer in a copy is set to `NULL` / `nullptr`
- Available in C++ Standard Library
  - `std::unique_ptr`
- Implemented in
  - Copy Constructor
  - `operator=`

Ownership Policy: Destructive Copy

```
template <class T>
class SmartPtr { public:
    SmartPtr(SmartPtr& src) { // Src ptr is not const
        pointee_ = src.pointee_; // Copy
        src.pointee_ = 0;        // Remove ownership for src ptr
    }
    SmartPtr& operator=(SmartPtr& src) { // Src ptr is not const
        if (this != &src) {         // Check & skip self-copy
            delete pointee_;        // Release destination object
            pointee_ = src.pointee_; // Assignment
            src.pointee_ = 0;       // Remove ownership for src ptr
        }
        return *this; // Return the assigned Smart Pointer
    } // ...
};
```

- Note that the copy operations (lvalue binding) here actually moves the resource – transfers ownership. Hence, the source object needs a non-const reference
- Though the semantics is similar to move operations, these are different from move operators (rvalue binding) due to lvalue binding

Ownership Policy: Destructive Copy: The Maelstrom Effect

- Consider a call-by-value

```
void Display(SmartPtr<Something> sp); // passed by value - copy performed
SmartPtr<Something> sp(new Something);
Display(sp); // sinks sp
```

- Display acts like a maelstrom of smart pointers:
  - It sinks any smart pointer passed to it
  - After `Display(sp)` is called, `sp` holds the null pointer
- Lesson: **Pass Smart Pointers by Reference**
- Smart pointers with destructive copy cannot usually be stored in containers and in general must be handled with care
- STL Containers need FCO

Ownership Policy: Destructive Copy: Advantages

- Incurs almost no overhead.
- Good at enforcing ownership transfer semantics
  - Use the *maelstrom effect* to ensure that the function takes over the passed-in pointer
- Good as return values from functions
  - The pointee object gets destroyed if the caller does not use the return value
- Excellent as stack variables in functions that have multiple return paths
- Available in Standard Library
  - `std::auto_ptr` [C++03, deprecated in C++11, removed in C++17]
  - `std::unique_ptr` [C++11]

Ownership Policy: Reference Counting / Linking

- Shared Ownership Policy
- Allow multiple Smart pointers to point to the same pointee
- **Reference Counting**: A count of the number of Smart pointers (references) to a pointee
  - *Non-Intrusive Counter*: Multiple / Single Raw Pointers per pointee with count in free store
  - *Intrusive Counter*: Count is a member of the object
  - Destroy the pointee Object when the count equals 0
- **Reference Linking**: All Smart pointers to a pointee are *linked on a chain*
  - The exact count is not maintained – only check if the chain is null
  - Destroy the pointee Object when the chain gets empty
- Available in C++ Standard Library
  - `std::shared_ptr`
  - `std::weak_ptr`
- Implemented in
  - Constructor
  - Copy Constructor
  - `operator=`
  - Destructor

Ownership Policy: Exclusive and Shared

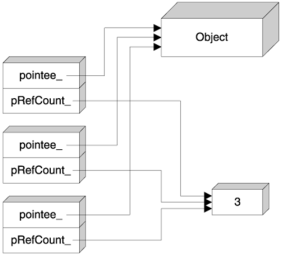
Exclusive Ownership

- **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=`

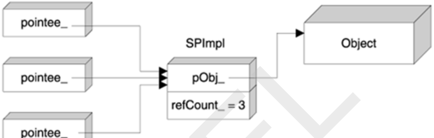
Shared Ownership

- **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

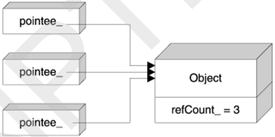
Ownership Policy: Exclusive and Shared



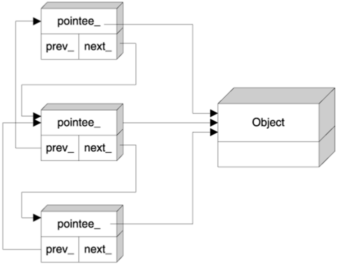
Extra pointer & non-intrusive counter



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

Module Summary

- 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:
  - Storage Policies
  - Ownership Policies