Smart Pointers C++

Thursday, October 10, 2019 8:48 PM

1. Smart Pointers in C++:

   a. Problem with Normal Pointers:
      i. A pointer may or may not point to an object
      ii. A pointer does not indicate who owns the object
      iii. Smart Pointers are used to express ownership

   b. 3 Types of Smart Pointers:

Unique_ptr	Represents exclusive ownership
Shared_ptr	Represents shared ownership
Weak_Ptr	To break loops in circular shared data structures

   c. We can make pointers to work in a way that many normal pointers can't do

      i. Automatic Pointer Destruction
      ii. Reference Counting

   d. Idea:
      i. Make another class with
         1) Pointer
         2) Destructor
         3)

Overloaded Operators	*	->

2. Smart Pointer Implementation:

a. There are 8 major things to implement in a Smart Pointer Implementation:

i.

1. Empty Constructor
2. Explicit Constructor
3. Destructor
4. Null Pointer Semantics
5. Move Semantics: 1. Move Constructor 2. Move Assignment
6. Copy Semantics: 1. Copy Constructor 2. Copy Assignment
7. Operator Overloading: 1. For Dereferencing operator

2. For Arrow operator

8. Others:
    1. Get
    2. Swap

```cpp
template<class T>
class UniquePtr {

    T*   data;
    public:

        // 1. Empty Constructor
        UniquePtr()
            : data(nullptr)
        {}

        // 2. Explicit constructor
        explicit UniquePtr(T* data)
            : data(data)
        {}

        // 3. Destructor
        ~UniquePtr() {
            delete data;
        }
```

```
30          // 4. NULLPTR Semantics
31          // Constructor/Assignment that binds to NULLPTR
32          // Null Pointer Constructor
33          UniquePtr(std::nullptr_t)
34              : data(nullptr)
35          {}
36
37          // Null Pointer Assignment
38          UniquePtr& operator=(std::nullptr_t) {
39              return *this;
40          }
41
42
43          // 5. MOVE Semantics
44          // Constructor/Assignment that allows MOVE semantics
45          // Move Constructor
46          UniquePtr( UniquePtr&& moving ) {
47
48              moving.swap(*this);
49          }
50
51          // Move Assignment
52          UniquePtr& operator = (UniquePtr&& moving) {
53              std::swap(data, *this);
54              return *this;
55          }
```

```
58          // 6. Copy Semantics
59          // Remove compiler generated COPY semantics.
60          // Copy Constructor Prohibited
61          UniquePtr ( UniquePtr const& ) = delete;
62
63          // Copy Assignment Prohibited
64          UniquePtr& operator = ( UniquePtr const& ) = delete;
65
66
67          // 7. Operator Overloading
68          // Operator Overloading for Arrow operator
69          T* operator->() {
70
71              return data;
72          }
73
74          // Operator Overloading for De-referncing operator
75          T& operator*() {
76
77              return *data;
78          }
79
```

```
        // 8. Others - Get and Swap
        // Returns the actual underlying raw pointer
        // Constant function
        T* get() {

            return data;
        }

        void swap(UniquePtr& src)
        {
            std::swap(data, src.data);
        }
};
```

```
typedef vector<int> Vector;

int main() {

    UniquePtr<Vector> up1 = UniquePtr<Vector>(new Vector());
    up1->push_back(4);
    up1->push_back(5);
    up1->push_back(6);

    cout << "Size of Vector = " << up1->size() << endl;

    return 0;
}
```

```
$g++ -std=c++11 -o main *.cpp
$main
Size of Vector = 3
```

3. Shared Pointer:

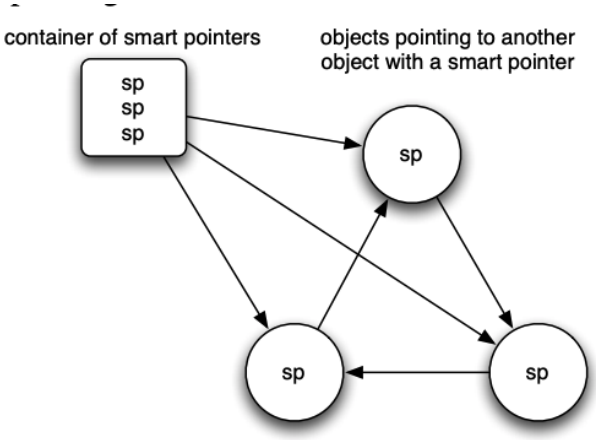
a. Referenced-counted Smart Pointer

b. A count is kept as to how many shared_ptrs are pointing to the managed object
c. When the last shared_ptr is destroyed, and count goes to zero, the managed object is automatically deleted
d. Called a shared_ptr because ownership of the object is shared among shared_ptrs
e. Any one shared_ptr can keep the managed object alive. Object deleted only when last shared_ptr goes out of scope

4. Problems with Shared pointer:

a. With referenced-counted shared_ptrs, if there is a ring or cycle, then shared_ptrs can keep each other alive
b. Objects will be alive even if no other shared_ptrs are not pointing to the managed object from the outside universe

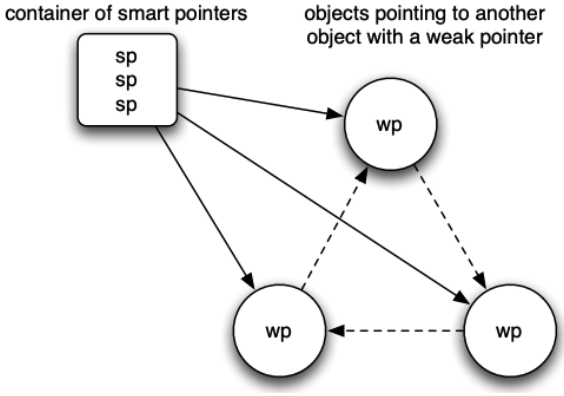
c.



d. Solution to the Ring Problem:

i. We can use Weak Pointers over here.
ii. They only observe an object but do not influence the lifetime and existence of the object
iii. If ring of objects point to each other with weak pointers, then when last shared_pointer from outside universe goes out of scope, the managed object will get deleted

iv.



5. Weak Pointers:

a. Unlike, raw pointers, weak pointers have an advantage that they know whether the managed object is still in existence or not
b. Weak pointer can look at the manager object and can tell whether the managed object is still in existence or has died

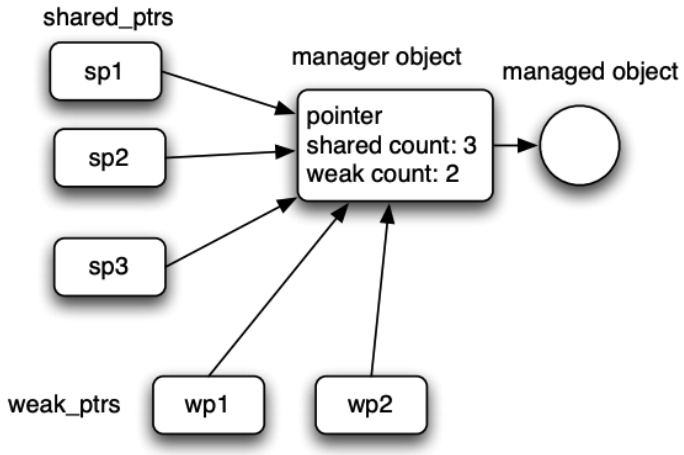
6. **Flow of Shared and Weak Pointers:**

a. Firstly:
   i. The process starts when the first shared pointer(sp1) is created to point to a managed object
   ii. Sp1 creates a manager object pointing to this managed object
   iii. Manager object also contains the reference counts for other shared pointers and weak pointers

b. Secondly:
   i. If another shared pointer(sp2) is created by copy or assignment from sp1, then it also points to the same manager object

ii. At this moment, reference count is incremented for the shared count

c. Likewise:
   i. If another weak pointer is created by copy or assignment from another shared pointer or weak pointer, it also points to the same manager object
   ii. Again, reference count is incremented, but this time for the weak count



d. Any of the Shared or Weak pointer is destroyed:
   i. Whenever a shared_ptr is destroyed, or reassigned to point to a different object, the shared_ptr destructor or assignment operator decrements the shared count in the manager object
   ii. If the shared count reaches 0, the shared_ptr destructor deletes the managed object, and sets the pointer to 0. However, if the weak count is greater than 0 at this point, manager object will still be in existence, even if managed object has been deleted

a)

Managed Object lifetime	As long as the shared count is greater than zero
Manager object lifetime	As long as any of the shared count and weak count - both are greater than zero

7. **Fundamental difference between shared and weak pointers:**
   a. Shared_ptr can be used syntactically almost identically to a built-in pointer
   b. You can basically do only 2 things with the weak pointer:
      i. Check whether the managed object is still in existence or not
      ii. If the managed object is still in existence, you may create a shared_ptr from the weak pointer

8. **Restrictions in using shared_ptr and weak_ptr:**
   a. It should always be ensured that there is only one manager object for the managed object
   b. Meaning, only the first shared pointer creates the manager object
   c. And then, all other shared and weak pointer afterwards are created from the first shared pointer only
   d. To ensure this, the shared_ptr should always be initialized using the "make_shared" function

9. Using the shared_ptr:

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <memory>

using namespace std;

class Movies {

  public:

    vector<string> movies;

    Movies(string movie)
        : movies{movie}
    {
    }

    void addMovie(string movie) {

        movies.push_back(movie);
    }

    void removeLastMovie() {

        if(movies.size()) {
            movies.pop_back();
        }
    }

    void printMovies() {

        for(auto & x: movies) cout << x << " --- ";
        cout << endl;
    }

};
```

```
46      // Method 1 of creating the shared pointer via Move Constructor
47      shared_ptr<Movies> sp1 = make_shared<Movies>("Sp1 Object");
48      sp1->addMovie("Batman");
49      sp1->printMovies();
50      cout << "Count of Sp1 = " << sp1.use_count() << endl << endl;
51
52
53      // Method 2 of creating the shared pointer via Move Constructor
54      shared_ptr<Movies> sp2(new Movies("Sp2 Object"));
55      sp2->printMovies();
56      cout << "Count of Sp2 = " << sp2.use_count() << endl << endl;
57
58
59      // Method 3a of creating the shared pointer
60      shared_ptr<Movies> sp3(nullptr);
61
62      // Method 3b of creating the shared pointer
63      // This method id equivalent to 3a
64      shared_ptr<Movies> sp4;
65
66      // sp4 was created before. Here, Move Assignment occurs
67      sp4 = make_shared<Movies>("Sp4 Object");
68      sp4->printMovies();
```

```
$g++ -std=c++11 -o main *.cpp
$main
Sp1 Object --- Batman ---
Count of Sp1 = 1

Sp2 Object ---
Count of Sp2 = 1

Sp4 Object ---
```

```
72      // sp5 is created and initialized via the Copy Assignment
73      auto sp5 = sp1;
74      cout << "Count of Sp5 = " << sp5.use_count() << endl << endl;
75
76
77      // sp6 is created via the Copy Constructor
78      auto sp6(sp1);
79      sp6->printMovies();
80      cout << "Count of Sp6 = " << sp6.use_count() << endl << endl;
81
82      // Comparison with NullPtr
83      cout << endl << "Is Sp3 a Nullptr = " << (sp3 == nullptr) << endl << endl;
84
```

```
Count of Sp5 = 2

Sp1 Object --- Batman ---
Count of Sp6 = 3


Is Sp3 a Nullptr = 1
```

```
86      // Use of Reset function deletes the shared pointer
87      // and makes it point to another object
88      cout << "Sp 5 count = " << sp5.use_count() << endl << endl;
89      sp1->printMovies();
90      sp1.reset(new Movies("Reset Object"));
91      sp1->printMovies();
92      cout << "Sp 5 count = " << sp5.use_count() << endl << endl;
93
```

```
Sp1 Object --- Batman ---
Reset Object ---
Sp 5 count = 2
```

```
// Get functionality on the Shared Pointer
cout << "Actual address of Sp5 = " << sp5.get() << endl << endl;


// Swap functionality on the Shared Pointer
// Now, Sp5 will print movies of Sp4
sp5.swap(sp4);
sp4->printMovies();
sp5->printMovies();
```

Sp_

```
Actual address of Sp5 = 0xe7bc30

Sp1 Object --- Batman ---
Sp4 Object ---
```

10. **Using the weak_ptr:**

a.

```
// Method 1 of creating the shared pointer via Move Constructor
shared_ptr<Movies> sp1 = make_shared<Movies>("Sp1 Object");
sp1->addMovie("Batman");
sp1->printMovies();
cout << "Count of Sp1 = " << sp1.use_count() << endl << endl;


// Weak Pointer created from SP via Copy Constructor
weak_ptr<Movies> wp1(sp1);
cout << "Shared Count = " << wp1.use_count() << endl << endl;
```

b.

```
$g++ -std=c++11 -o main *.cpp
$main
Sp1 Object --- Batman ---
Count of Sp1 = 1

Shared Count = 1
```

c.

```
58      // Weak Pointer created from shared pointer via Copy Assignment
59      weak_ptr<Movies> wp2 = sp1;
60
61
62      // Move Constructor and Assignment may be defined for Weak Pointer
63      // But they are not used
64
65
66      // Checking for expiry of the managed object
67      cout << "Object Expired = " << wp1.expired() << endl;
68
69      // Reset on the Shared Pointer
70      sp1.reset();
71      cout << "Shared Count = " << wp1.use_count() << endl << endl;
72      cout << "Object Expired = " << wp1.expired() << endl << endl;
```

```
Object Expired = 0
Shared Count = 0

Object Expired = 1
```

d.

```
    shared_ptr<Movies> sp2 = make_shared<Movies>("Sp2 Object");
    sp2->addMovie("Spiderman");
    sp2->printMovies();
    cout << "Count of Sp2 = " << sp2.use_count() << endl << endl;


    // Getting Shared Pointer from the Weak Pointer
    // Use of Lock on Weak Pointer
    // Lock function will give an exception if the managed object has already expired
    cout << "Count of Sp2 = " << sp2.use_count() << endl << endl;
    weak_ptr<Movies> wp3(sp2);
    shared_ptr<Movies> sp3 = wp3.lock();
    cout << "Count of Sp2 = " << sp2.use_count() << endl << endl;
```

```
Sp2 Object --- Spiderman ---
Count of Sp2 = 1

Count of Sp2 = 1

Count of Sp2 = 2
```

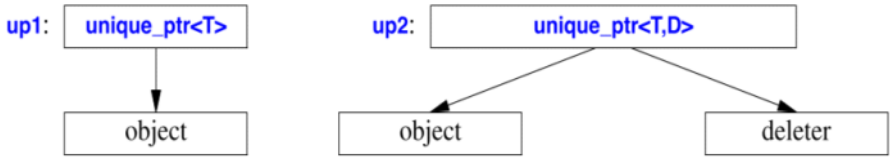
11. **Unique Pointer:**

i. Owns the object to which it points to. Has the obligation to destroy the pointed object after the scope

ii. Unique Ownership - Unique Pointer cannot be copied:
   1) Does not have a copy constructor
   2) Does not have a Copy Assignment
   3) Has Move Constructor and Move Assignment
   4) In case of Move Assignment, the original pointer is lost and no longer owns the original object

iii. It stores a pointer and deletes the object pointed to using the associated deleter

iv. Uses:
   1) Returning dynamically allocated memory from a function



12. **Benefits of Unique Pointer over shared pointer:**
   i. Unique pointer has no overhead - it only carries the underlying object pointer
   ii. Shared Pointer

1) It carries the overhead of the manager object
2) Overhead of increasing and decreasing the reference count

13. Usage Of Unique Pointers:

1. Move Constructor:

```
49
50      // Method 1 of creating the unique pointer via Move Constructor
51      unique_ptr<Movies> up1 = make_unique<Movies>("Spiderman");
52      up1->addMovie("Batman");
53      up1->printMovies();
54
55
56      // Trying Copy Assignment on Unique Pointer
57      // unique_ptr<Movies> up2 = up1;
58
59      // Trying Copy Constructor on Unique Pointer
60      //unique_ptr<Movies> up3(up1);
61
62
63      unique_ptr<Movies> up4 = move(up1);
64      up4->printMovies();
65      //up1->printMovies();
66
```

```
Spiderman --- Batman ---
Spiderman --- Batman ---
Destructor called for Movies
```

2. Copy Assignment:

```
49
50      // Method 1 of creating the unique pointer via Move Constructor
51      unique_ptr<Movies> up1 = make_unique<Movies>("Spiderman");
52      up1->addMovie("Batman");
53      up1->printMovies();
54
55
56      // Trying Copy Assignment on Unique Pointer
57      unique_ptr<Movies> up2 = up1;
58
```

```
uniquePtr.cpp:57:22: error: call to implicitly-deleted copy constructor of 'unique_ptr<Movies>'
                unique_ptr<Movies> up2 = up1;
                                   ^     ~~~
/Library/Developer/CommandLineTools/usr/include/c++/v1/memory:2490:3: note: copy constructor is implicitly deleted because
      user-declared move constructor
  unique_ptr(unique_ptr&& __u) noexcept
  ^
1 error generated.
```

1. Copy Constructor:

```
49
50      // Method 1 of creating the unique pointer via Move Constructor
51      unique_ptr<Movies> up1 = make_unique<Movies>("Spiderman");
52      up1->addMovie("Batman");
53      up1->printMovies();
54
55
56      // Trying Copy Assignment on Unique Pointer
57      // unique_ptr<Movies> up2 = up1;
58
59      // Trying Copy Constructor on Unique Pointer
60      unique_ptr<Movies> up3(up1);
```

```
uniquePtr.cpp:60:22: error: call to implicitly-deleted copy constructor of 'unique_ptr<Movies>'
            unique_ptr<Movies> up3(up1);
                               ^   ~~~
/Library/Developer/CommandLineTools/usr/include/c++/v1/memory:2490:3: note: copy constructor is implicitly deleted because 'u
      user-declared move constructor
  unique_ptr(unique_ptr&& __u) noexcept
  ^
1 error generated.
```

4. Move Assignment:

```
50      // Method 1 of creating the unique pointer via Move Constructor
51      unique_ptr<Movies> up1 = make_unique<Movies>("Spiderman");
52      up1->addMovie("Batman");
53      up1->printMovies();
54
55
56      // Trying Copy Assignment on Unique Pointer
57      // unique_ptr<Movies> up2 = up1;
58
59      // Trying Copy Constructor on Unique Pointer
60      // unique_ptr<Movies> up3(up1);
61
62
63      // Move Assignment on Unique Pointer
64      unique_ptr<Movies> up4 = move(up1);
65      up4->printMovies();
66      up1->printMovies();
```

```
Spiderman --- Batman ---
Spiderman --- Batman ---
Segmentation fault: 11
```