C++ smart pointers

Dr. Yihsiang Liow (May 1, 2024)

Contents

Chapter 1

C++ smart pointers

Look at my CISS245 notes and review the `IntPointer` class example and for CISS350, look at a02 and look for the `ptr` class template. Note that `IntPointer` is just a wrapper class for `int *` with the ability to manage memory automatically (allocation and deallocation) for the user of the class. Note that if `p` and `q` are `IntPointer` objects, the values they point to (through their actual pointers) are at different addresses. This means that

```
IntPointer q = p; // copy constructor
```

has a different intention from

```
int * q = p;
```

C++ STL provides pointer class templates:

- `std::auto_ptr` – deprecated
- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

`std::auto_ptr` was the first C++ pointer class template and has been deprecated. An object created from one of these STL pointer classes is sometimes called a **smart pointer** while a regular C/C++ pointer is sometimes called **raw pointer** . I will call an object from the class `std::unique_ptr` a unique pointer. A shared pointer is an object from the `std::shared_ptr` class and a weak pointer is a object from `std::weak_ptr`.

smart pointer
raw pointer

To use the above three STL pointer classes, the header file to include is:

```
#include <memory>
```

For initialization (i.e. constructor) you can do

```
std::unique_ptr< int > p = std::make_unique< int >(42);
std::shared_ptr< int > q = std::make_shared< int >(43);
```

(there's no `std::make_weak`). You can do the following for initial memory allocation

```
std::unique_ptr< int > p(new int);
std::shared_ptr< int > q(new int);
```

but the above `make_*` functions are better.

For the above `p` (or `q`), you can get the address of the pointer in `p` or the value that the pointer in `p` is pointing to:

```
std::cout << p.get() << '\n';   // p.get() is address of p's ptr
std::cout << *p << '\n';        // value that p's ptr is pointing to
```

Again you can't do the above for `std::weak_ptr`.

Obviously just like the `IntPointer` class from CISS245, `p` will automatically deallocate if necessary when `p` goes out of scope. I say "if necessary" because there are cases when this has to be delayed. See later.

Recall from the `IntPointer` example from CISS245, the point of that class is to automatically deallocate the value that an `IntPointer` object is pointing to. This is of course done in the destructor of `IntPointer`.

C++ will keep track on how many smart pointers are pointing to a `T`. These numbers are called reference counts. I'll call such a number a **refcount**. If the refcount for a `T` value (that is in the memory heap) is `0`, C++ will then know that no smart pointer is pointing to this value and so this `T` value can be deallocated and can be given back to the memory heap.

refcount

In the case of C++ STL smart pointers, the three smart pointer classes have the following behaviors. Briefly `std::unique_ptr< T >`s do not point to the same value while `std::shared_ptr< T >`s can. There are also `std::weak_ptr< T >` objects, but they play a slightly different role – I'll talk about them later.

The above means that the value that a `std::unique_ptr< T >` object is pointing to will always have a refcount of 1. When this unique pointer goes out of scope or points to another value, the refcount of the original value becomes 0 and that value can be safely deallocated. This is similar to our `IntPointer` class from CISS245.

For the case of the value that a `std::shared_ptr< T >` object is pointing to, if this pointer goes out of scope or points to another value, the refcount of the value drops by 1 and can only be safely deallocated when the refcount reaches

0. You can get the refcount of this value:

```
std::shared_ptr< int > p = std::make_shared< int >(42);
std::cout << p.use_count() << '\n'; // refcount is 1
```

Clearly the refcount for `std::unique_ptr< T >` is not useful. So there's no `use_count()` for a `std::unique_ptr< T >` object. By the way `std::weak_ptr< T >` objects do not take part in reference counting so they don't have `use_count()`.

1.1 std::unique_ptr

A std::unique_ptr< T > object is basically an object the acts like a pointer that points to a T value in such a way that other std::unique_ptr< T > objects cannot point to this same T value. So std::unique_ptr< T > objects don't share values.

Run and study the following example code:

```
#include <iostream>
#include <memory> // for std::unique_ptr and std::make_unique

int main()
{
    std::cout << "\n1 ...\n";
    std::unique_ptr< int > p;          // p's ptr set to NULL
    std::cout << p.get() << '\n';      // p.get() returns value of p's ptr
    p = std::make_unique< int >();     // allocate memory for p's ptr
    *p = 42;                           // *(p's ptr) = 42
    std::cout << p.get() << ' ' << *p << '\n';

    std::cout << "\n2 ...\n";
    std::unique_ptr< int > q(new int); // allocate memory for q's ptr
    *q = 43;
    std::cout << q.get() << ' ' << *q << '\n';

    std::cout << "\n3 ...\n";
    std::unique_ptr< int > r = std::make_unique< int >(44); // allocate
                                       // memory for r's ptr and
                                       // *(r's ptr) = 42
    std::cout << r.get() << ' ' << *r << '\n';

    std::cout << "\n4 ...\n";
    std::unique_ptr< int > s = std::move(r); // s's ptr = r's ptr,
                                       // set r's ptr to NULL, i.e., s
                                       // takes over r's ownership of value
    std::cout << r.get() << '\n';
    std::cout << s.get() << ' ' << *s << '\n';

    std::cout << "\n5 ...\n";
    auto t = std::make_unique< int >(44);
    t = NULL;                          // deallocate t's ptr
    std::cout << t.get() << '\n';
    t = std::make_unique< int >(44);
    t.reset();                         // another way to deallocate t's ptr

    std::cout << "\n6 ...\n";
    auto u = std::make_unique< int >(45);
    auto v = std::make_unique< int >(46);
```

```
    std::cout << u.get() << ' ' << v.get() << '\n';
    // v = u;                           // ERROR
    v = std::move(u);                   // deallocate v's ptr,
                                        // set v's ptr to u's ptr,
                                        // set u's ptr to NULL, i.e.,
                                        // v takes over r's ownership of
                                        // value
    std::cout << u.get() << ' ' << v.get() << '\n';
    u.reset(new int);                   // another way to reallocate
    std::cout << u.get() << '\n';

    return 0;
                                        // auto deallocate all heap memory
                                        // usage
}
```

```
[student@localhost cpp-smart-pointers] g++ main.cpp
[student@localhost cpp-smart-pointers] ./a.out
1 ...
0
0x1706ec0 42
2 ...
0x1706ee0 43
3 ...
0x1706f00 44
4 ...
0
0x1706f00 44
5 ...
0
6 ...
0x1706f20 0x1706f40
0 0x1706f20
0x1706f40
```

Examples of constructor calls are

```
std::unique_ptr< int > p; // no memory allocate
std::unique_ptr< int > q = std::make_unique< int >();
std::unique_ptr< int > r = std::make_unique< int >(42);
std::unique_ptr< int > s(new int);
```

or

```
auto q = std::make_unique< int >();
auto r = std::make_unique< int >(42);
```

The following are ways to allocate memory after declaration:

```
p = std::make_unique< int >();
p = std::make_unique< int >(42);
p.reset(new int);
```

To deallocate memory:

```
p = NULL;
p.reset();
```

Note that for assignment, if `p` and `q` are `std::unique_ptr` objects, you cannot do

```
p = q; // ERROR!!!
```

But for `std::unique_ptr` objects, you can pass the value that `q` points to over to `p`, which will result in `q` *losing* its original value. In other words, ownership of the value is passed from `q` to `p`.

```
p = std::move(q);
```

`p`'s pointer will be set to `q`'s pointer and then `q`'s pointer will be set to `NULL`. Furthermore, the origin value that `p` points too will be deallocated so that there's no memory leak.

Note that you can do this:

```
int * p = new int;
std::unique_ptr q(p);
*p = 42;
std::cout << p << ' ' << q.get() << '\n'; // same address printed
std::cout << *p << ' ' << *q << '\n';     // 42 printed twice
```

So in this case `p` and `q` do share the same value (i.e., the `42`). So if you mix `std::unique_ptr` and raw pointers, you can break the usage intention of `std::unique_ptr`. `std::unique_ptr` cannot prevent this from happening.

1.2 std::shared_ptr

`std::shared_ptr` is similar to except that a bunch of them can point to the same value.

Examples of constructor calls:

```
std::shared_ptr< int > p;                       // no memory allocation
std::shared_ptr< int > q  = std::make_shared< int >();
std::shared_ptr< int > r  = std::make_shared< int >(42);
std::shared_ptr< int > s = r;                   // q's ptr = p's ptr
```

or

```
auto q = std::make_shared< int >();
auto r = std::make_shared< int >(42);
auto s = r;
```

Deallocation is the same as unique pointers:

```
p.reset();
p = NULL;
```

You can perform assignment for shared pointers or change ownership

```
p = q;
p = std::move(q);
```

Also, you can find out how many `std::shared_ptr< T >` objects are pointing to a value:

```
std::shared_ptr< int > p = std::make_shared< int >(0);
std::shared_ptr< int > q = p;
std::shared_ptr< int > r = p;
std::cout << p.use_count() << '\n';     // 3: p,q,r point to the same 0
```

Run and study the following very carefully:

```
#include <iostream>
#include <memory> // for std::unique_ptr and std::make_unique

int main()
{
    std::cout << "\n0 ...\n";
    std::shared_ptr< int > p;         // p's ptr set to NULL
    std::cout << p.get() << '\n';     // p.get() returns value of p's ptr

    std::cout << "\n1 ...\n";
```

```
    p = std::shared_ptr< int >(new int); // p's ptr set to new int
    *p = 42;                             // *(p's ptr) = 42
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';

    std::cout << "\n2 ...\n";
    p = std::make_shared< int >();       // deallocate p's ptr,
                                         // p's ptr = new int
    std::cout << p.get() << p.use_count() << '\n';

    std::cout << "\n3 ...\n";
    p = std::make_shared< int >(43);     // deallocate p's ptr,
                                         // p's ptr = new int,
                                         // and *(p's ptr) = 0
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';

    std::cout << "\n4 ...\n";
    p = NULL;                            // deallocate p's ptr and set it to
                                         // NULL
    std::cout << p.get() << '\n';

    std::cout << "\n5 ...\n";
    std::shared_ptr< int > q = std::make_shared< int >(0);
                                         // p's ptr = new int,
                                         // and *(p's ptr) = 0

    std::cout << "\n6 ...\n";
    *q = 44;
    std::cout << q.get() << ' ' << *q << '\n';

    std::cout << "\n7 ...\n";
    std::shared_ptr< int > r = q;        // = works for shared_ptr
    std::cout << r.get() << ' ' << *r << ' ' << r.use_count() << '\n';
    std::cout << q.get() << ' ' << *q << ' ' << q.use_count() << '\n';
    *r = 99;
    std::cout << r.get() << ' ' << *r << ' ' << r.use_count() << '\n';
    std::cout << q.get() << ' ' << *q << ' ' << q.use_count() << '\n';

    std::cout << "\n8 ...\n";
    r = std::move(q);                    // change ownership
                                         // same as for std::unique_ptr
    std::cout << r.get() << ' ' << *r << ' ' << r.use_count() << '\n';
    std::cout << q.get() << ' ' << q.use_count() << '\n';

    return 0;
}
```

```
[student@localhost cpp-smart-pointers] g++ main.cpp
[student@localhost cpp-smart-pointers] ./a.out
0 ...
0
```

```
1 ...
0x10d9ec0 42 1
2 ...
0x10d9f101
3 ...
0x10d9ef0 43 1
4 ...
0
5 ...
6 ...
0x10d9ef0 44
7 ...
0x10d9ef0 44 2
0x10d9ef0 44 2
0x10d9ef0 99 2
0x10d9ef0 99 2
8 ...
0x10d9ef0 99 1
0 0
```

Exercise 1.2.1. Can you assign a unique pointer to a shared pointer (of the same type)? Can you assign a shared pointer to a unique pointer (of the same type)?

1.3 std::weak_ptr

A weak pointer `w` can watch a value `v` that was initially tracked by a shared pointer. If all shared pointers leaves this `v` (i.e., no shared pointers hold the address of `v`), then `w` can tell you that `v` is not safely accessible by checking `w.expired()`.

If `w.expired()` is false (i.e., the address that `w` is watching is still in use by some shared pointer), you can create a shared pointer to point to the address that `w` is watching by doing this:

```
if (!w.expired())
    p = w.lock();
```

Run and study this carefully:

```
#include <iostream>
#include <memory> // for std::unique_ptr and std::make_unique

int main()
{
    std::cout << "\n0 ...\n";
    auto p = std::make_shared< int >(42);
    std::weak_ptr< int > w = p;
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n1 ...\n";
    std::shared_ptr< int > q = p;
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';
    std::cout << q.get() << ' ' << *q << ' ' << q.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n2 ...\n";
    std::shared_ptr< int > r = p;
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';
    std::cout << q.get() << ' ' << *q << ' ' << q.use_count() << '\n';
    std::cout << r.get() << ' ' << *r << ' ' << r.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n3 ...\n";
    p.reset();
    std::cout << p.get() << ' ' << p.use_count() << '\n';
    std::cout << q.get() << ' ' << *q << ' ' << q.use_count() << '\n';
    std::cout << r.get() << ' ' << *r << ' ' << r.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n4 ...\n";
```

```
    q.reset();
    r.reset();
    std::cout << p.get() << ' ' << p.use_count() << '\n';
    std::cout << q.get() << ' ' << q.use_count() << '\n';
    std::cout << r.get() << ' ' << r.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n5 ...\n";
    p = std::make_shared< int >(43);
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n6 ...\n";
    p = std::make_shared< int >(43);
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n7 ...\n";
    w = p;
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    std::cout << "\n7 ...\n";
    p = std::make_shared< int >(44);
    w = p;
    q = w.lock();
    std::cout << p.get() << ' ' << *p << ' ' << p.use_count() << '\n';
    std::cout << q.get() << ' ' << *q << ' ' << q.use_count() << '\n';
    std::cout << w.use_count() << ' ' << w.expired() << '\n';

    return 0;
}
```

```
[student@localhost cpp-smart-pointers] g++ main.cpp
[student@localhost cpp-smart-pointers] ./a.out
0 ...
0x1d4ced0 42 1
1 0
1 ...
0x1d4ced0 42 2
0x1d4ced0 42 2
2 0
2 ...
0x1d4ced0 42 3
0x1d4ced0 42 3
0x1d4ced0 42 3
3 0
3 ...
0 0
0x1d4ced0 42 2
```

```
0x1d4ced0 42 2
2 0
4 ...
0 0
0 0
0 0
0 1
5 ...
0x1d4cef0 43 1
0 1
6 ...
0x1d4cf10 43 1
0 1
7 ...
0x1d4cf10 43 1
1 0
7 ...
0x1d4ced0 44 2
0x1d4ced0 44 2
2 0
```

Suppose you have a lookup table of values v_0, ..., v_{999}. You can store these in a lookup table `x[0..999]`. But suppose they are expensive to compute and store. Then when you need v_{42}, you compute it (in the heap) and store the address of v_{42} at `x[42]`. When you are done, you can release v_{42} and set `x[42]` to `NULL`. Of course while a function f is using v_{42}, that function might call another function g which might also want to use v_{42}. It would be bad if g deallocate this v_{42} if f still needs it!

Another way to do the above is to let `x[0..999]` be an array of weak pointers. If a function need v_{42} it will check if `x[42]` has expired. If not, it will use `x[42]` (a weak pointer) and create a shared pointer to point to v_{42}. If it is, then this function will need to create v_{42}, store that as a shared pointer, and set `x[42]` to this shared pointer. Get it? Using shared pointers and weak pointers to build a cache is very common.

(Python also has the concept of weak references. But in the case of python, when an object goes out of scope, its memory is not reclaimed immediately. This means that when a refcount of an object goes to zero, the object might still be around since an object's memory is reclaimed by the garbage collector and not directly by the object's destructor. This means that the weak reference might still have the ability to create a pointer to that object, resetting the refcount to 1.)

Here's an example you should run and study very carefully. The example involves two objects (in the heap) pointing to each other (i.e., there's a cycle).

Here's the point. In the first example, the objects in the heap do not call their destructors. In the second example, they do. In the first example the objects store shared pointers while in the second example the objects store weak pointers.

EXAMPLE 1:

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

class X
{
public:
    X()
    {
        std::cout << "X::X()\n";
    }
    ~X()
    {
        std::cout << "X::~X()\n";
    }
    std::shared_ptr< X > p;
};

int main()
{
    auto x = std::make_shared< X >();
    auto y = std::make_shared< X >();
    x->p = y;
    y->p = x;
    return 0;
}
```

```
[student@localhost cpp-smart-pointers] g++ main.cpp
[student@localhost cpp-smart-pointers] ./a.out
X::X()
X::X()
```

EXAMPLE 2:

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

class X
{
public:
    X()
    {
        std::cout << "X::X()\n";
```

```
    }
    ~X()
    {
        std::cout << "X::~X()\n";
    }
    std::weak_ptr< X > p;
};

int main()
{
    auto x = std::make_shared< X >();
    auto y = std::make_shared< X >();
    x->p = y;
    y->p = x;
    return 0;
}
```

```
[student@localhost cpp-smart-pointers] g++ main.cpp
[student@localhost cpp-smart-pointers] ./a.out
X::X()
X::X()
X::~X()
X::~X()
```

The above is a serious warning to writing caches or garbage collectors where objects have complex cyclical relationships.

Index