C++11 Smart Pointers

Babu_Abdulsalam

26 Sep 2013 CPOL

Various Smart Pointers in C++11

- **Download Unique Pointer source - 4.54 KB**
- **Download Weak Pointer source - 3.95 KB**
- **Download Shared Pointer source - 4.31 KB**

Introduction

Ooops. Yet another article on smart pointers of C++11. Nowadays I hear a lot of people talking about the new C++ standard which is nothing but C++0x/C++11. I went through some of the language features of C++11 and it's really an amazing work. I'll focus only on the smart pointers section of C++11.

Background

What are the issues with normal/raw/naked pointers?

Let's go one by one.

People refrain to use pointers as they give a lot of issues if not handled properly. That's why the newbie programmers dislike pointers. Many issues are involved with pointers like ensuring the lifetime of objects referred to by pointers, dangling references, and memory leaks.

Dangling reference is caused if a memory block is pointed by more than one pointer variable and if one of the pointers is released without letting know the other pointer. As all of you know, memory leaks occur when a block of memory is fetched from the heap and is not released back.

People say, I write clean and error proof code, why should I use smart pointers? And a programmer asked me, "Hey, here is my code. I fetched the memory from the heap, manipulated it, and after that I released it properly. What is the need of a smart pointer? "

```
void Foo( )
{
    int* iPtr = new int[5];

    //manipulate the memory block
    .
    .
    .

    delete[ ] iPtr;
}
```

The above code works fine and memory is released properly under ideal circumstances. But think of the practical environment of code execution. The instructions between memory allocation and releasing can do nasty things like accessing an invalid memory location, dividing by zero, or say another programmer pitching into your program to fix a bug and adding a premature `return` statement based on some condition.

In all the above cases, you will never reach the point where the memory is released. This is because the first two cases throw an exception whereas the third one is a premature return. So the memory gets leaked while the program is running.

The one stop solution for all of the above issues is Smart Pointers [if they are really smart enough].

What is a smart pointer?

Smart pointer is a RAII modeled class to manage dynamically allocated memory. It provides all the interfaces provided by normal pointers with a few exceptions. During construction, it owns the memory and releases the same when it goes out of scope. In this way, the programmer is free about managing dynamically allocated memory.

C++98 has introduced the first of its kind called `auto_ptr`.

auto_ptr

Let's see the use of `auto_ptr` and how smart it is to resolve the above issues.

```
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
  cout<<"Calling destructor"<<endl;
 }
public:
 int m_a;
};

void main( )
{
 std::auto_ptr<Test> p( new Test(5) );
 cout<<p->m_a<<endl;
}
```

The above code is smart to release the memory associated with it. What we did is, we fetched a memory block to hold an object of type `Test` and associated it with `auto_ptr p`. So when `p` goes out of scope, the associated memory block is also released.

```
//*******************************************************************
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
  cout<<"Calling destructor"<<endl;
 }
public:
 int m_a;
};
//*******************************************************************
void Fun( )
{
 int a = 0, b= 5, c;
 if( a ==0 )
```

```
 {
  throw "Invalid divisor";
 }
 c = b/a;
 return;
}
//*************************************************************
void main( )
{
 try
 {
  std::auto_ptr<Test> p( new Test(5) );
  Fun( );
  cout<<p->m_a<<endl;
 }
 catch(...)
 {
  cout<<"Something has gone wrong"<<endl;
 }
}
```

In the above case, an exception is thrown but still the pointer is released properly. This is because of stack unwinding which happens when an exception is thrown. As all local objects belonging to the `try` block are destroyed, `p` goes out of scope and it releases the associated memory.

Issue 1: So far `auto_ptr` is smart. But it has more fundamental flaws over its smartness. `auto_ptr` transfers the ownership when it is assigned to another `auto_ptr`. This is really an issue while passing the `auto_ptr` between the functions. Say, I have an `auto_ptr` in `Foo( )` and this pointer is passed another function say `Fun( )` from `Foo`. Now once `Fun( )` completes its execution, the ownership is not returned back to `Foo`.

```
//*************************************************************
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
  cout<<"Calling destructor"<<endl;
 }
public:
 int m_a;
};


//*************************************************************
void Fun(auto_ptr<Test> p1 )
{
 cout<<p1->m_a<<endl;
}
//*************************************************************
void main( )
{
 std::auto_ptr<Test> p( new Test(5) );
 Fun(p);
 cout<<p->m_a<<endl;
}
```

The above code causes a program crash because of the weird behavior of `auto_ptr`. What happens is that, `p` owns a memory block and when `Fun` is called, `p` transfers the ownership of its associated memory block to the `auto_ptr p1` which is the copy of `p`. Now `p1` owns the memory block which was previously owned by `p`. So far it is fine. Now `fun` has completed its execution, and `p1` goes out of scope and the memory blocked is released. How about `p`? `p` does not own anything, that is why it causes a crash when the next line is executed which accesses `p` thinking that it owns some resource.

Issue 2: Yet another flaw. `auto_ptr` cannot be used with an array of objects. I mean it cannot be used with the operator `new[]`.

```
//*****************************************************************
void main( )
{
 std::auto_ptr<Test> p(new Test[5]);
}
```

The above code gives a runtime error. This is because when `auto_ptr` goes out of scope, `delete` is called on the associated memory block. This is fine if `auto_ptr` owns only a single object. But in the above code, we have created an array of objects on the heap which should be destroyed using `delete[ ]` and not `delete`.

Issue 3: `auto_ptr` cannot be used with standard containers like vector, list, map, etc.

As `auto_ptr` is more error prone and it will be deprecated, C++ 11 has come with a new set of smart pointers, each has its own purpose.

- `shared_ptr`
- `unique_ptr`
- `weak_ptr`

shared_ptr

OK, get ready to enjoy the real smartness. The first of its kind is `shared_ptr` which has the notion called shared ownership. The goal of `shared_ptr` is very simple: Multiple shared pointers can refer to a single object and when the last shared pointer goes out of scope, memory is released automatically.

Creation:

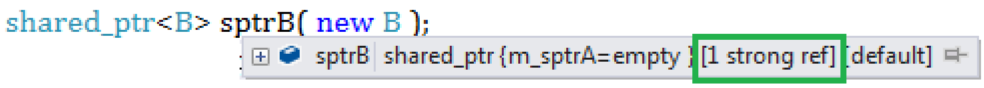
```
void main( )
{
 shared_ptr<int> sptr1( new int );
}
```

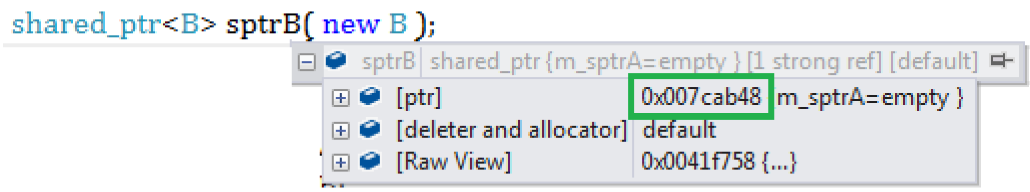
Make use of the `make_shared` macro which expedites the creation process. As `shared_ptr` allocates memory internally, to hold the reference count, `make_shared( )` is implemented in a way to do this job effectively.

```
void main( )
{
 shared_ptr<int> sptr1 = make_shared<int>(100);
}
```

The above code creates a `shared_ptr` which points to a memory block to hold an integer with value 100 and reference count 1. If another shared pointer is created out of `sptr1`, the reference count goes up to 2. This count is known as *strong reference*. Apart from this, the shared pointer has another reference count known as *weak reference*, which will be explained while visiting weak pointers.

You can find out the number of `shared_ptr`s referring to the resource by just getting the reference count by calling `use_count( )`. And while debugging, you can get it by watching the `stong_ref` of the `shared_ptr`.





Destruction:

`shared_ptr` releases the associated resource by calling `delete` by default. If the user needs a different destruction policy, he/she is free to specify the same while constructing the `shared_ptr`. The following code is a source of trouble due to the default destruction policy:

```
class Test
{
public:
 Test(int a = 0 ) : m_a(a)
 {
 }
 ~Test( )
 {
  cout<<"Calling destructor"<<endl;
 }
public:
         int m_a;
};
void main( )
{
 shared_ptr<Test> sptr1( new Test[5] );
}
```

In this scenario shared_ptr owns array of objects and the default destruction policy calls "delete" to release the associated memory when it goes out of scope. Actually, `delete[ ]` should have been called to destroy the array. The user can specify the custom deallocator by a callable object, i.e., a function, lambda expression, function object.

```
void main( )
{
 shared_ptr<Test> sptr1( new Test[5],
        [ ](Test* p) { delete[ ] p; } );
}
```

The above code works fine as we have specified the destruction should happen via `delete[]`.

Interface

`shared_ptr` provides dereferencing operators `*`, `->` like a normal pointer provides. Apart from that it provides some more important interfaces like:

- `get( )` : To get the resource associated with the `shared_ptr`.
- `reset( )` : To yield the ownership of the associated memory block. If this is the last `shared_ptr` owning the resource, then the resource is released automatically.
- `unique`: To know whether the resource is managed by only this `shared_ptr` instance.
- `operator bool`: To check whether the `shared_ptr` owns a memory block or not. Can be used with an `if` condition.

OK, that is all about `shared_ptr`s. But `shared_ptr`s too have a few issues:.

Issues:

```
void main( )
{
 shared_ptr<int> sptr1( new int );
 shared_ptr<int> sptr2 = sptr1;
 shared_ptr<int> sptr3;
 sptr3 = sptr2;
}
```

The below table gives you the reference count values for the above code.

	Reference count
shared_ptr<int> sptr1(new int)	1
shared_ptr<int> sptr2 = sptr1	2
sptr3 = sptr2	3
When main ends -> sptr3 goes out of scope	2
When main ends -> sptr2 goes out of scope	1
When main ends -> sptr1 goes out of scope	0 -> The resource is released as the count goes down to zero.

All `shared_ptr`s share the same reference count hence belonging to the same group. The above code is fine. Let's see another piece of code.

```
void main( )
{
 int* p = new int;
 shared_ptr<int> sptr1( p);
 shared_ptr<int> sptr2( p );
}
```

The above piece of code is going to cause an error because two `shared_ptr`s from different groups share a single resource. The below table gives you a picture of the root cause.

	Reference count
shared_ptr<int> sptr1(p)	1
shared_ptr<int> sptr2(p)	1
main ends -> sptr1 goes out of scope	0 -> memory block pointed by sptr1 or p is destroyed as ref count is 0.
main ends -> sptr2 goes out of scope	0 -> Crashhhhhhh!!!!!. Because this tries to release memory block associated with sptr2, which is already being destroyed.

To avoid this, better not create the shared pointers from a naked pointer.

```
class B;
class A
{
public:
 A(  ) : m_sptrB(nullptr) { };
 ~A( )
 {
  cout<<" A is destroyed"<<endl;
 }
 shared_ptr<B> m_sptrB;
};
class B
{
public:
```

```
 B(  ) : m_sptrA(nullptr) { };
 ~B( )
 {
  cout<<" B is destroyed"<<endl;
 }
 shared_ptr<A> m_sptrA;
};
//*****************************************************************
void main( )
{
 shared_ptr<B> sptrB( new B );
 shared_ptr<A> sptrA( new A );
 sptrB->m_sptrA = sptrA;
 sptrA->m_sptrB = sptrB;
}
```

The above code has cyclic reference. I mean class A holds a shared pointer to B and class B holds a shared pointer to A. In this case, the resource associated with both `sptrA` and `sptrB` are not released. Refer to the below table.

	Reference count
shared_ptr<B> sptrB(new B)	1
shared_ptr<A> sptrA(new A)	1
sptrB->m_sptrA = sptrA	Ref count of sptrA -> 2
sptrA->m_sptrB = sptrB	Ref count of sptrB ->2
main ends -> sptrA goes out of scope	Ref count of sptrA ->1
main ends -> sptrB goes out of scope	Ref count of sptrB ->1

Reference counts for both `sptrA` and `sptrB` go down to 1 when they go out of scope and hence the resources are not released!!!!!

1. If a memory is block is associated with `shared_ptr`s belonging to a different group, then there is an error. All `shared_ptr`s sharing the same reference count belong to a group. Let's see an example.
2. There is another issue involved with creating a shared pointer from a naked pointer. In the above code, consider that only one shared pointer is created using `p` and the code works fine. Consider by mistake if a programmer deletes the naked pointer `p` before the scope of the shared pointer ends. Oooppss!!! Yet another crash..
3. Cyclic Reference: Resources are not released properly if a cyclic reference of shared pointers are involved. Consider the following piece of code.

To resolve the cyclic reference, C++ provides another smart pointer class called `weak_ptr`.

Weak_Ptr

A weak pointer provides sharing semantics and not owning semantics. This means a weak pointer can share a resource held by a `shared_ptr`. So to create a weak pointer, some body should already own the resource which is nothing but a shared pointer.

A weak pointer does not allow normal interfaces supported by a pointer, like calling `*`, `->`. Because it is not the owner of the resource and hence it does not give any chance for the programmer to mishandle it. Then how do we make use of a weak pointer?

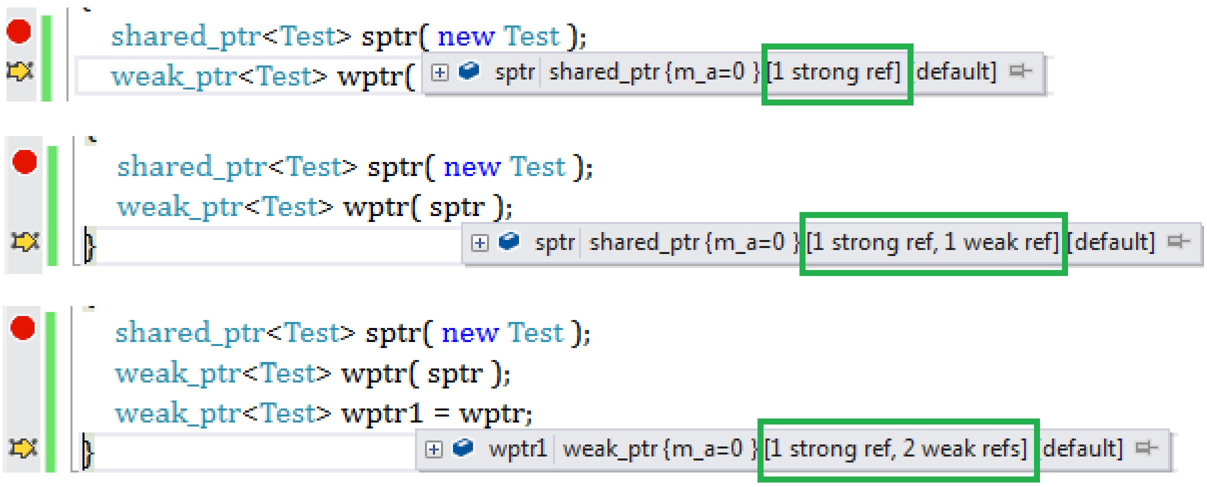
The answer is to create a `shared_ptr` out of a `weak _ptr` and use it. Because this makes sure that the resource won't be destroyed while using by incrementing the strong reference count. As the reference count is incremented, it is sure that the count will be at least 1 till you complete using the `shared_ptr` created out of the `weak_ptr`. Otherwise what may happen is while using the `weak_ptr`, the resource held by the `shared_ptr` goes out of scope and the memory is released which creates chaos.

Creation

A weak pointer constructor takes a shared pointer as one of its parameters. Creating a weak pointer out of a shared pointer increases the *weak reference* counter of the shared pointer. This means that the shared pointer shares it resource with another pointer. But this counter is not considered to release the resource when the shared pointer goes out of scope. I mean if the strong reference of the shared pointer goes to 0, then the resource is released irrespective of the weak reference value.

```
void main( )
{
 shared_ptr<Test> sptr( new Test );
 weak_ptr<Test> wptr( sptr );
 weak_ptr<Test> wptr1 = wptr;
}
```

We can watch the reference counters of the shared/weak pointer.



Assigning a weak pointer to another weak pointer increases the weak reference count.

So what happens when a weak pointer points to a resource held by the shared pointer and the shared pointer destroys the associated resource when it goes out of scope? The weak pointer gets expired.

How to check whether the weak pointer is pointing to a valid resource? There are two ways:

1. Call the `use_count( )` method to know the count. Note that this method returns the strong reference count and not the weak reference.
2. Call the `expired( )` method. This is faster than calling `use_count( )`.

To get a `shared_ptr` from a `weak_ptr` call `lock( )` or directly casting the `weak_ptr` to `shared_ptr`.

```
void main( )
{
 shared_ptr<Test> sptr( new Test );
 weak_ptr<Test> wptr( sptr );
 shared_ptr<Test> sptr2 = wptr.lock( );
}
```

Getting the `shared_ptr` from the `weak_ptr` increases the strong reference as said earlier.

Now let's see how the cyclic reference issue is resolved using the `weak_ptr`.

```
class B;
class A
{
public:
 A(  ) : m_a(5)  { };
 ~A( )
 {
  cout<<" A is destroyed"<<endl;
```

```
 }
 void PrintSpB( );
 weak_ptr<B> m_sptrB;
 int m_a;
};
class B
{
public:
 B(  ) : m_b(10) { };
 ~B( )
 {
  cout<<" B is destroyed"<<endl;
 }
 weak_ptr<A> m_sptrA;
 int m_b;
};

void A::PrintSpB( )
{
 if( !m_sptrB.expired() )
 {
  cout<< m_sptrB.lock( )->m_b<<endl;
 }
}

void main( )
{
 shared_ptr<B> sptrB( new B );
 shared_ptr<A> sptrA( new A );
 sptrB->m_sptrA = sptrA;
 sptrA->m_sptrB = sptrB;
 sptrA->PrintSpB( );
}
```

	Reference Count
shared_ptr<B> sptrB(new B)	Strong Ref = 1
shared_ptr<A> sptrA(new A)	Strong Ref = 1
sptrB->m_sptrA = sptrA	Strong Ref = 1, Weak Ref = 1
sptrA->m_sptrB = sptrB	Strong Ref = 1, Weak Ref = 1
Main ends -> sptrA goes out of scope	Strong Ref = 0 , Weak Ref = 1 As strong reference goes to 0, the resource is released.
Main ends -> sptrB goes out of scope	Strong Ref = 0 , Weak Ref = 1 As strong reference goes to 0, the resource is released.

Unique_ptr

This is almost a kind of replacement to the error prone `auto_ptr`. `unique_ptr` follows the exclusive ownership semantics, i.e., at any point of time, the resource is owned by only one `unique_ptr`. When `unique_ptr` goes out of scope, the resource is released. If the resource is overwritten by some other resource, the previously owned resource is released. So it guarantees that the associated resource is released always.

Creation

unique_ptr is created in the same way as shared_ptr except it has an additional facility for an array of objects.

```
unique_ptr<int> uptr( new int );
```

The unique_ptr class provides the specialization to create an array of objects which calls delete[ ] instead of delete when the pointer goes out of scope. The array of objects can be specified as a part of the template parameter while creating the unique_ptr. In this way, the programmer does not have to provide a custom deallocator, as unique_ptr does it.

```
unique_ptr<int[ ]> uptr( new int[5] );
```

Ownership of the resource can be transferred from one unique_ptr to another by assigning it.

Keep in mind that unique_ptr does not provide you copy semantics [copy assignment and copy construction is not possible] but move semantics.

In the above case, if upt3 and uptr5 owns some resource already, then it will be destroyed properly before owning a new resource.

Interface

The interface that unique_ptr provides is very similar to the ordinary pointer but no pointer arithmetic is allowed.

unique_ptr provides a function called release which yields the ownership. The difference between release() and reset( ), is release just yields the ownership and does not destroy the resource whereas reset destroys the resource.

Which one to use?

It purely depends upon how you want to own a resource. If shared ownership is needed then go for shared_ptr, otherwise unique_ptr.

Apart from that, shared_ptr is a bit heavier than unique_ptr because internally it allocates memory to do a lot of book keeping like strong reference, weak reference, etc. But unique_ptr does not need these counters as it is the only owner for the resource.

Using the code

I have attached the worked out code to explain the details of each pointer. I have added enough comments to each instruction. Ping me back if you find any problems with the code. The weak pointer example demonstrates the problems with shared pointers in the case of cyclic reference and how the weak pointer resolves it.

History

This is the first version of the article. I'll keep you updated based on feedback and comments.

License

About the Author

Babu_Abdulsalam

Software Developer (Senior)
India

I'm working as Senior software Engineer since 7 years and interested in MFC and COM programming.

Comments and Discussions

54 messages have been posted for this article Visit **https://www.codeproject.com/Articles/541067/Cplusplus-Smart-Pointers** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Web02 2.8.200113.1