TR1 Intro - Smart Pointers An introduction to the TR1 smart pointers. Rate Topic: 1 Votes

bszmyd
Posted 15 February 2009 - 01:04 PM

Intro to TR1: Smart Pointers

Many of you may or may not be aware that we are on the verge of a new C++ standard.

This introduction series will cover some of the already excepted concepts and libraries dubbed TR1 (Technical Report 1).

These libraries already exist in most modern C++ compilers and though the new standard has not been finalized you can begin to use them in your code today!

I will give a nod to the Boost group for providing most of these implementations. You can find a slew of useful C++ libraries, some of which I will cover in later topics, at the Boost Homepage (http://www.boost.org) . Libraries range in everything from meta programming to sockets and SSL.

This first section will cover the smart pointers.

What are they

As most of us have had to find out the hard way the current C++ standard does little for memory management. Sure they gave us the auto_ptr, but that has a lot of limitations. The next two pointers, shared_ptr and weak_ptr try to relieve some headache with resource allocation and release.

They can do much more than help you release dynamically allocated memory! They can help release any kind of resource whether it be memory, file handles or semaphores.

The Headers

Like all libraries you need to know where the headers are so you can get to the declarations. One of the things about the TR1 is that all the libraries are purely in headers, so there is no need to link against any new binaries!

You may need to double check your compiler, but for gcc the header you need to include in order to use smart pointers is:

```
#include <tr1/memory>
```

shared_ptr

After including the appropriate header you now have access to **shared_ptr**. **shared_ptr** is a templatized class, so before we can go and use it we need to know what kind of type we will be storing in it. For a preliminary example let's assume we are holding a dynamically allocated int, this is how we would declare that:

```
std::tr1::shared_ptr<int> sp;
```

Incidentally everything that comes from the TR1 project will reside in the **std::tr1** namespace. Now that we have our pointer we need to assign something to it. Here's the trick...the shared_ptr does not implement a templatized version operator=(T*) so you cannot use the assignment operator to assign your pointer to it. The following would fail to compile:

```
sp = new int(5);
```

We can, however, use either the constructor, copy constructor or copy assignment to assign our memory to it:

```
sp = std::tr1::shared_ptr<int>( new int(5) );
```

Great...now how do we use our shared_ptr? That's the best part, we can use it just like any other pointer (aside from passing it to a function that takes a C pointer).

We can use all the normal dereferencing syntax to get at the value the pointer points to. For instance:

```
std::cout << "The value of sp is: " << *sp << std::endl;
```

If sp was pointing at a more complex structure that had a member function called *getName* which returned a string we could write:

```
std::cout << "Name: " << sp->getName();
```

You can use the implicit bool conversion operator to test if the shared_ptr is still valid:

```
if ( sp ) // pointer usage;
```

"So what??" you're probably asking yourself. Right, so all that with regular pointers, what makes this one so special. Well, the biggest difference is that if properly done, we need not ever call delete on it! That's right, unless we make some huge error somewhere we should never find ourselves with a memory leak. This is because shared_ptr is what we call a *reference counted pointer*. So when it knows that nobody has a handle anymore it can just delete whatever it is pointing at.

The way this works in essense is that each *shared_ptr* is actually a shell that points to the real shared pointer. This pointer internally has a counter. This counter gets incremented when anyone makes a new shell out of it, including the originator. As these shells get destroyed, by going out of scope, their destructors are called. These destructors decrement the real pointer and when it reaches zero it knows that nobody can reference what it's pointing to (with a couple of buts).

This allows us to write code like:

```
01 #include <tr1/memory>
02 #include <iostream>
03 typedef std::tr1::shared_ptr<int> int_ptr;
04
05 void foo( int_ptr a )
06 {
07   std::cout << "Foo got value of: " << *a << std::endl;
08 }
09
10 int main( int, char*[] )
11 {
12   for ( int i = 0; i < 10; ++i )
13   {
14     int_ptr number( new int( i ) );
15     foo( number );
16   }
17 }
```

Now, I know this code doesn't make much sense, but it illustrates my point when I say that it does not leak any memory. Even though we are calling new 10 times, with no deletes, each time number is reassigned to a new pointer the old shell decrements the previous pointer causing it to delete the data. Even foo participates by taking the argument as a **shared_ptr**. When in foo the pointer has a reference of 2. So if we were in a multi-threaded environment we don't need to coordinate when to reliquish memory back to the pool.

Explicitly Releasing

Often times we need to explicitly release our handles to the memory we are pointing to and decrement the reference counter. To do this with a smart pointer call the **shared_ptr::release()** method. attempting to set it to NULL will fail.

Deleters

Now let's talk about what to do when you don't have something that delete is going to release. For instance, a malloc'd character array. You definitely don't want to call delete on that, so you're stuck right? Wrong. Another beauty of these pointers are that they can talk a second argument that is a function to call instead of delete. In the case we just described, that'd be **free**. We pass this in the constructor as the second argument called the *Deleter*:

```
std::tr1::shared_ptr<char> smart_char_array((char*)malloc( 16 ), free);
```

The only requirement that a Deleter has is that it is has a type **void** and it's argument list is a single pointer to the thing we are holding. In the case of free that looks like:

```
void free(void*)
```

We can use this to our advantage with things like FILE handles

Microsoft : Shaped Windows
Microsoft : Implementing an Indexed Table : Part III
Microsoft : Implementing an Indexed Table : Part II
Processing Data Held in A Comma Separated File
The reasons for using vectors
Introduction to C++ Metaprogramming: Basics
Pointers, and a possible problem - if you're not careful!
Generating Random Numbers - The C++ Way
Hello World: Your first C and C++ Programs
351 More C++ Tutorials...

Reference Sheets

Code Snippets
C Snippets
C++ Snippets
Java Snippets
Visual Basic Snippets
C# Snippets
VB.NET Snippets
ASP.NET Snippets
PHP Snippets
Python Snippets
Ruby Snippets
ColdFusion Snippets
SQL Snippets
Assembly Snippets
Functional Programming Snippets
Perl Snippets
HTML/CSS Snippets

C and C++
VB.NET
Java
C#
Python
PHP
Mobile Development
ASP.NET
.NET Framework
Ruby
Game Development
Assembly
Databases
ColdFusion
VB6
Other Languages
52 Weeks Of Code

Web Development
Web Development
HTML & CSS
JavaScript
Graphic Design
Flash & ActionScript
Blogging
SEO & Advertising
Web Servers & Hosting
Site Check

obtained with fopen see in the following code:

```
01 #include <cstdio>
02 #include <string>
03 #include <tr1/memory>
04
05 typedef std::tr1::shared_ptr<FILE> file_ptr;
06
07 void writeString( file_ptr fp, std::string s )
08 {
09    fwrite( s.c_str(), sizeof(char), s.size(), fp.get() );
10 }
11
12 int main( int, char*[] )
13 {
14    file_ptr fp( fopen("tr1_rocks.txt", "w"), fclose );
15
16    writeString( fp, "I love smart pointers.\n" );
17    writeString( fp, "They make life easy.\n" );
18
19    return 0; // No need to fclose the file here!
20 }
```

This example also brings to attention the situation in which you need access to the basic C pointer to pass to some older library call. Using the **get()** method will return the C pointer (NULL if invalid). Use **SPARINGLY** as it defeats the purpose of using reference counted pointers if you were to make copies of it. The following code is an example of what not to do:

```
1 char* c_ptr;
2 {
3    std::tr1::shared_ptr<char> my_ptr( (char*) malloc( 16 ), free );
4    c_ptr = my_ptr.get();
5 }
6 printf("%s", c_ptr); // << BAD!
```

The reason this is going to cause problems is that after my_ptr goes out of scope then it will call free on the memory it allocated thus leaving c_ptr dangling into the ether. Had c_ptr been another shared_ptr we would not have this problem.

weak_ptr

I'll just briefly talk about the *weak_ptr*. Weak pointers must be made from a shared pointer. That is, their constructor takes a shared_ptr of the same template.

The reason is that they do no participate in reference counting. If we use our previous example with the dangling pointer and use a weak pointer instead:

```
1 std::tr1::weak_ptr<char> w_ptr;
2 {
3    std::tr1::shared_ptr<char> my_ptr( (char*) malloc( 16 ), free );
4    w_ptr = my_ptr;
5 }
```

This code is perfectly acceptable. The memory from my_ptr still is dereferenced when my_ptr goes out of scope, the difference is we can not directly use a weak pointer. You must first obtain a shared pointer from it and then use the shared pointer:

```
1 std::tr1::weak_ptr<char> w_ptr;
2 {
3    std::tr1::shared_ptr<char> my_ptr( (char*)
```

```
    malloc( 16 ), free );
    w_ptr = my_ptr;
}
std::tr1::shared_ptr<char> s_ptr( w_ptr );
printf("%s", s_ptr.get());
```

When you attempt to create a shared pointer from a weak pointer that was created from a expired shared pointer it will throw an exception. Thus our **printf** above will never be reached in this example. We might put try/catch blocks around the creation of s_ptr.

Use **weak_ptr** in replacement of a C pointer when you don't want to use a **shared_ptr** that participates in reference counting. This is found frequently when you have a class that contains a pointer to some resource, but you don't want its existence to prevent the release of that resource.

Conclusion

I'll end with a quick mention of the STL. Often we find ourselves wanting to store pointers in the STL containers. For instance:

```
std::vector<char*> char_vec;
char_vec.push_back( (char*) malloc( 16 ) );
```

We do this so we can easily modify the contents of the container without having to push and pop the elements in and out of the container. The problem is that when we are done with an element or the container as a whole we have to release the resources they maintained. Not a trival thing to do and usually ends up looking like a loop over the container calling **delete** or **free** many times.

If you store smart pointers in the container it's a no brainer. As soon as you pop the element off the container and it goes out of scope cleanup is done auto-magically! Even if the whole container goes out of scope this applies which makes for cleaner, safer code:

```
typedef std::tr1::shared_ptr<char> smart_cptr;
std::vector<smart_cptr> char_vec;
char_vec.push_back( smart_cptr( (char*) malloc( 16 ), free ) );
```

This post has been edited by **bszmyd**: 19 February 2009 - 10:51 PM

Replies To: TR1 Intro - Smart Pointers

vividexstance
Posted 06 February 2011 - 11:20 AM

I wanted to see for myself that the smart pointers clean-up after themselves, so I modified your first example, and of course, the smart pointers DO clean-up for you(like the author of this tutorial said).

```
// smart-pointers.cpp
// example usage of shared-pointers in the TR1 library <tr1/memory>
#include <tr1/memory>
#include <iostream>
using namespace std;

class Int
{
    int value;

public:
    Int(int val = 0) : value(val)
        { cout << "Constructing Int with value: " << value << endl; }
    ~Int()
        { cout << "\nDestructing Int with value: " << value << endl; }

    friend ostream& operator<<(ostream& os, const Int& i)
```

```
        { return os << i.value; }
};

typedef tr1::shared_ptr<Int> iptr;

void foo(iptr a)
{
    cout << "foo() called with value = " << *a << endl;
}

int main(int, char*[])
{
    for(int i = 0; i < 10; i++)
    {
        iptr num( new Int(i));
        foo(num);
    }

    return 0;
}
```

Page 1 of 1

C++ Basic

Pointers

Question