Creating the C++ auto_ptr<> utility for Symbian OS

Sander van der Wal
Revision 1.0, Aug 2002

Introduction

Before becoming a Symbian OS C++ developer, Sander van der Wal worked with Pascal as a systems designer and project leader for a Dutch software company building, maintaining and selling a Hospital Information System. The programs were of the traditional, non-event driven kind. Some of the projects he worked on include a subset SQL compiler and an almost complete rewrite of the company's text editing system.

After establishing mBrain Software, Sander published three highly successful programs including the Pdf+ (an Adobe PDF file viewer) and FontMachine (an Open Font System font renderer). Sander has extensive experience with Symbian OS phones – Pdf+, for example, is available for both the Symbian OS v5 powered Psion (and compatible) devices and also for the Symbian OS v6.0 powered Nokia 9200 Series Communicator.

1.1. Managing memory in Symbian OS applications

Memory is a scarce resource on Symbian OS phones. It is vital that applications conserve memory where possible and certainly never 'leak' it. Memory leaks very quickly result in a phone that needs to be rebooted, undermining the robust nature of Symbian OS and its own applications.

Symbian OS has various tools to help developers prevent memory leaks, one of which is the Cleanup Stack. A local pointer to memory is pushed on to the Cleanup Stack right after the memory has been allocated. The pointer is popped when it is no longer relevant – for example if the memory address is assigned to a class member variable or when the memory has been freed.

A basic example is:

```
void Example1L()
	{
	CClass* localPtr = CClass::NewL();
	CleanupStack::PushL(localPtr);
	(void)DoSomeWorkL(localPtr);
	CleanupStack::PopAndDestroy(); // localPtr
	}
```

If `DoSomeWorkL()` leaves (i.e. exits with some error condition), the memory pointed to by `localPtr` will be properly destroyed by the Cleanup Stack automatically.

Using the Cleanup Stack this way is common to every well-written Symbian OS C++ application and is tried and tested. However, developers should be aware of some potential issues:

1) Use of the Cleanup Stack is not portable to other operating systems. This is especially important when you want to reuse your engine code

2) It is still possible to introduce some subtle defects, as the following example shows:

```
void Example2L()
	{
	CClass* localPtr = CClass::NewL();
	CleanupStack::PushL(localPtr);
	TInt res = DoSomeWorkL(localPtr);
	if (KErrNone == res)
		return; // Oops, localPtr is still on the Cleanup Stack!
	DoSomeMoreWorkL(localPtr);
	CleanupStack::PopAndDestroy(); // localPtr
```

```
	}
```

In this example, it is clear that there is a defect - but it is easy to imagine a case in a more complicated function where such a defect could be overlooked. Additionally, if an alternative code path is not executed very often, a user might find the defect which regular testing had not revealed. It would be nice to have a way of coding that would make introducing such subtle defects impossible.

3) Finally, multiple calls to `CleanupStack::PopAndDestroy()` can make code bulkier and (in my eyes at least) harder to read:

```
void Example3L()
	{
	CClass* localPtr = CClass::NewL();
	CleanupStack::PushL(localPtr);
	TInt res = DoSomeWorkL(localPtr);
	if (KErrNone == res)
		{
		CleanupStack::PopAndDestroy(); // localPtr
		return;
		}
	DoSomeMoreWorkL(localPtr);
	CleanupStack::PopAndDestroy(); // localPtr
	}
```

1.2. About auto_ptr

The C++ language standard defines the template class `auto_ptr<>` as the standard way of handling the automatic destruction of local memory, even when an exception is thrown. Examine the following function:

```
void Example4L()
	{
	auto_ptr<CClass> localPtr(CClass::NewL());
	TInt res = DoSomeWork(localPtr);
	if (KErrNone == res)
		return; // Automatic destruction of *localPtr
	DoSomeMoreWork(localPtr);
	// Automatic destruction of *localPtr
	}
```

It is clear that the second problem mentioned above does not occur in `Example4L();` - as soon as `localPtr` goes out of scope, the memory pointed to is released automatically. Further, because `auto_ptr<>` is part of the C++ standard, it is portable to all operating systems with a standard C++ implementation.

An implementation of `auto_ptr<>` looks like this (I have ignored a lot of details that aren't important for the discussion at this moment)

```
template<class X> class auto_ptr
	{
public:
	auto_ptr(X* aPtr = 0):  iPtr(aPtr) {}
	~auto_ptr() { delete iPtr; }
	X& operator*() const { return *iPtr; }
	X* operator->() const { return iPtr; }
	X* get() const { return iPtr; }
	X* release() { X* temp = iPtr; iPtr = 0; return temp; }
```

```
private:
	X* iPtr;
	};
```

The constructor receives as an argument the pointer to the newly created resource. The `auto_ptr<X>` destructor calls the destructor of class X using the delete operator.

The two methods `operator*` and `operator->` make `auto_ptr<Class>` act as a 'smart pointer', using `auto_ptr<X> var` looks identical to using `X* var`.

The `get()` method returns the address of the pointer. The `release()` method is there to transfer ownership of the memory, for example to an instance variable.

It is easy to see why this `auto_ptr<>` class works as advertised - C++ guarantees that the destructor of a class is called when a variable of that class goes out of scope, whether that happens in the normal flow of the code execution, or when an exception is thrown.

1.3. Implementing auto_ptr for Symbian OS C++

Symbian OS C++ also calls the destructor of a class when a variable of that class goes out of scope. This means that the declaration of `auto_ptr<class X>` above works for Symbian OS as well - but only when the function where it is used does not Leave. However, if the function does not Leave then it isn't necessary to protect resources in this way anyway, so we gain nothing.

As mentioned in the introduction, in Symbian OS C++ the Cleanup Stack is used to keep track of resources that must be destroyed. What we need to do is try to use the Cleanup Stack to keep track of our `auto_ptr<class X>` automatic variables. Take a look at this implementation:

```
template<class X> class auto_ptr
	{
public:
	auto_ptr(X* aPtr = 0): iPtr(aPtr) { CleanupStack::PushL(iPtr); }
	~auto_ptr() { CleanupStack::Pop(); delete iPtr; }
	X& operator*() const { return *iPtr; }
	X* operator->() const { return iPtr; }
	X* get() const { return iPtr; }
	X* release() { X* temp = iPtr; iPtr = 0; return temp; }
private:
	X* iPtr;
	};
```

The constructor puts a pointer to itself on the Cleanup Stack. The destructor removes this pointer, and then calls the destructor of class X using the standard `delete` operator. This means that normal function exits work effectively.

If a Leave occurs, the copy of the pointer on the Cleanup Stack is used to call the destructor of class X too, either because in Symbian OS C++ class X is derived from `CBase`, or because the class has no destructor. This means that cleaning up after a Leaving function works too.

Does this mean that we now have a working `auto_ptr` for Symbian OS? Unfortunately not!

The problem is the `release()` method. If you want to transfer ownership of a class X variable, you must use the `release()` method. Because of it's ownership transfer semantics, `release()` sets `iPtr` to zero, to stop the destructor deleting the resource. Now, let's take a look at the code below.

```
void Example5L()
	{
	auto_ptr<CClass> p1(CClass::NewL(1));
	auto_ptr<CClass> p2(CClass::NewL(2));
	// Transfer ownership
	p2->iMember = p1.release();
	DoSomeMoreWorkL(p2.get());
	}
```

So if `DoSomeMoreWorkL()` Leaves, the resource will be deleted twice - once through `p2->iMember`, and once again by the Cleanup Stack. This defect is worse than the original one we had - we have traded a memory leak for an unfriendly panic, which is visible to the user!

1.4. Implementing a fully working auto_ptr for Symbian OS C++

Fortunately, there is a simple enough solution. Adding an extra level of indirection can solve all programming problems. In this case, the solution is not to push the pointer, but the `auto_ptr<class X>` variable itself, like this:

```
template<class X> class auto_ptr
	{
public:
	auto_ptr(X* aPtr = 0):iPtr(aPtr) { TCleanupItem(Close,(void*)this); }
	~auto_ptr() { CleanupStack::Pop(); delete iPtr; }
	X& operator*() const { return *iPtr; }
	X* operator->() const { return iPtr; }
	X* get() const { return iPtr; }
	X* release() { X* temp = iPtr; iPtr = 0; return temp; }
private:
	static void Close(void* aPtr)
		{
		auto_ptr<X>* self = (auto_ptr<X>*)aPtr;
		delete self->iPtr;
		}
private:
	X* iPtr;
	};
```

Note that the complete definition of auto_ptr is contained in the file auto_ptr\stdcpp\include\auto_ptr.h, the above code is an edited, cut-down version.

The trick here is that we use a feature of the Cleanup Stack that allows us to put any kind of pointer on it alongside a special function (called `Close()` here) which knows how to properly destroy that specific pointer.

The destructor now pops this item, and then destroys the resource correctly via the `Close()` function. When a leave happens, the Cleanup Stack pops the item, and passes it to the `Close()` function. The `Close()` function knows that it has a pointer to an `auto_ptr<X>` variable, so it casts the `void` pointer and the calls the destructor using the standard `delete` operator.

Now, if `iPtr` becomes zero because of a `release()` call, nothing goes wrong. The `auto_ptr<X>` pointer gets popped in the destructor, and `delete` operator is still called on `iPtr` - but because `iPtr` is zero, nothing happens. If a Leave occurs, `Close()` gets called. `Close()` calls the delete operator on `iPtr`, which is still zero, and once again nothing bad happens.

1.5. Using auto_ptr<> instead of the Cleanup Stack

A number of Symbian OS C++ coding idioms can be formulated using auto_ptr<> as well.

1) Implementing the `NewL()` method is straightforward:

```
static CClass* CClass::NewL()
      {
      auto_ptr<CClass> self(new(ELeave) CClass);
      self->ConstructL();
      return self.release();
      }
```

2) It is not possible to create a `NewLC()` method that returns a pointer and leaves an `auto_ptr<>` on the Cleanup Stack. However, it is possible for the `NewLC()` method to return an `auto_ptr<>`.

```
static auto_ptr<CClass> CClass::NewLC()
      {
      auto_ptr<CClass> self(new(ELeave) CClass);
      self->ConstructL();
      return self;
      }

// ...
// Use copy constructor
auto_ptr<CClass> ptr(CClass::NewLC());
```

The only problem with this construct is that Microsoft Visual C++ 6.0 does not like it.

3) I do not recommend mixing `auto_ptr` and the normal Cleanup Stack calls. The problem here is that it is very hard to keep track of what is where on the Cleanup Stack, especially when you pop and push multiple types of variable. If you need to protect pointers to non-class resources, such as memory buffers, you are better off writing auto_ptr-like wrapper classes.

4) Finally, remember once again that using the Cleanup Stack directly is not portable.

1.6. Installing and running the example program

In the accompanying auto_ptr.zip file you will find the implementation of `auto_ptr` in the file auto_ptr\stdcpp\include\auto_ptr.h. I have also created a small example program that contains all the example functions mentioned above. You can compile the program and run it on the emulator or the device.

I have tested the example program on both Symbian OS v5, and on the Nokia 9200 Series SDK, v1.2 (Symbian OS v6.0)

The folder ER5 has the .mmp file for use in with a Symbian OS v5 SDK. You need to copy the folder auto_ptr to the root disk of your SDK installation. Then open a DOS box and run the commands:

```
cd \auto_ptr\ER5
makmake -makework auto_ptr_test wins
makmake auto_ptr_test  vc5
```

to create an MS VC++ workspace in the ER5 folder. Open the workspace, set the “active Configuration” to Win32 Debug, compile and run using F5.

The folder V6 has the bld.inf file for Symbian OS v6.0. I expect this to work on Symbian OS v6.1 (for example with the Series 60 SDK for the Nokia 7650 imaging phone) as well, but I haven't yet tried this. Open a DOS box and run the following commands (where <path> is the full pathname of the folder that contains auto_ptr):

```
cd <path>\auto_ptr\V6
bldmake bldfiles
abld build wins udeb
abld makefile vc6
```

to create an MS VC++ 6.0 workspace. Open the workspace, compile and run using F5.

If you are using the MetroWerks or Borland IDE, please follow the appropriate user guides as I don't use these new development suites yet.

It is also possible to create EXE files for the target devices – please check the appropriate SDK if you do not know how to do this.

On the emulator, try setting breakpoints at each `ExamplexL()` function and single-stepping through the complete example to get a feel for what is happening. There is also a function called `Test()` that runs all example functions in a memory-leak test harness. On a target device, memory leak testing is not very useful, but is included anyway.

Finally, as proof that it really does work, `auto_ptr<>` is already being successfully used in production code - Pdf+ 1.25 for both the Psion and the Nokia 9200 range implement the version included in this tutorial.

1.7. What next?

This implementation of `auto_ptr<>` is not complete, as you'll see when examining the auto_ptr.h header file in auto_ptr\stdcpp\include\auto_ptr.h. The next step would be to create an auto_ptr implementation that can be used on all current Symbian OS SDK's, including the v5 one. If `auto_ptr<>` is to become a part of normal Symbian OS C++ programming, it should be optimized for handling `CBase`-derived classes and handle-based (`RHandleBase`-derived) classes.

1.8. References

[1] Stroustrup, Bjarne. The C++ programming language, third edition. Addison-Wesley

1.9. Acknowledgements

I would like to thank Andrew Johnson for examining the first implementation of auto_ptr. Published with permission of Sander van der Wal. © Sander van der Wal 2002.