Smart Pointers - What, Why, Which?

Yonat Sharon

Translations:

- Ukrainian

What are they?

Smart pointers are objects that look and feel like pointers, but are smarter. What does this mean?

To look and feel like pointers, smart pointers need to have the same interface that pointers do: they need to support pointer operations like dereferencing (operator *) and indirection (operator ->). An object that looks and feels like something else is called a proxy object, or just proxy. The proxy pattern and its many uses are described in the books Design Patterns and Pattern Oriented Software Architecture.

To be smarter than regular pointers, smart pointers need to do things that regular pointers don't. What could these things be? Probably the most common bugs in C++ (and C) are related to pointers and memory management: dangling pointers, memory leaks, allocation failures and other joys. Having a smart pointer take care of these things can save a lot of aspirin...

The simplest example of a smart pointer is auto_ptr, which is included in the standard C++ library. You can find it in the header <memory>, or take a look at Scott Meyers' auto_ptr implementation. Here is part of auto_ptr's implementation, to illustrate what it does:

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                 {delete ptr;}
    T& operator*()              {return *ptr;}
    T* operator->()             {return ptr;}
    // ...
};
```

As you can see, auto_ptr is a simple wrapper around a regular pointer. It forwards all meaningful operations to this pointer (dereferencing and indirection). Its smartness in the destructor: the destructor takes care of deleting the pointer.

For the user of auto_ptr, this means that instead of writing:

```
void foo()
{
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

You can write:

```
void foo()
{
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething();
}
```

And trust p to cleanup after itself.

What does this buy you? See the next section.

Why would I use them?

Obviously, different smart pointers offer different reasons for use. Here are some common reasons for using smart pointers in C++.

Why: *Less bugs*

Automatic cleanup. As the code above illustrates, using smart pointers that clean after themselves can save a few lines of code. The importance here is not so much in the keystrokes saved, but in reducing the probability for bugs: you don't need to remember to free the pointer, and so there is

no chance you will forget about it.

Automatic initialization. Another nice thing is that you don't need to initialize the auto_ptr to NULL, since the default constructor does that for you. This is one less thing for the programmer to forget.

Dangling pointers. A common pitfall of regular pointers is the dangling pointer: a pointer that points to an object that is already deleted. The following code illustrates this situation:

```
MyClass* p(new MyClass);
MyClass* q = p;
delete p;
p->DoSomething();    // Watch out! p is now dangling!
p = NULL;            // p is no longer dangling
q->DoSomething();    // Ouch! q is still dangling!
```

For auto_ptr, this is solved by setting its pointer to NULL when it is copied:

```
template <class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
    if (this != &rhs) {
        delete ptr;
        ptr = rhs.ptr;
        rhs.ptr = NULL;
    }
    return *this;
}
```

Other smart pointers may do other things when they are copied. Here are some possible strategies for handling the statement q = p, where p and q are smart pointers:

- **Create a new copy** of the object pointed by p, and have q point to this copy. This strategy is implemented in copied_ptr.h.
- **Ownership transfer**: Let both p and q point to the same object, but transfer the responsibility for cleaning up ("ownership") from p to q. This strategy is implemented in owned_ptr.h.
- **Reference counting**: Maintain a count of the smart pointers that point to the same object, and delete the object when this count becomes zero. So the statement q = p causes the count of the object pointed by p to increase by one. This strategy is implemented in counted_ptr.h. Scott Meyers offers another reference counting implementation in his book More Effective C++.
- **Reference linking**: The same as reference counting, only instead of a count, maintain a circular doubly linked list of all smart pointers that point to the same object. This strategy is implemented in linked_ptr.h.
- **Copy on write**: Use reference counting or linking as long as the pointed object is not modified. When it is about to be modified, copy it and modify the copy. This strategy is implemented in cow_ptr.h.

All these techniques help in the battle against dangling pointers. Each has each own benefits and liabilities. The *Which* section of this article

discusses the suitability of different smart pointers for various situations.

Why: *Exception Safety*

Let's take another look at this simple example:

```
void foo()
{
    MyClass* p(new MyClass);
    p->DoSomething();
    delete p;
}
```

What happens if DoSomething() throws an exception? All the lines after it will not get executed and p will never get deleted! If we're lucky, this leads only to memory leaks. However, MyClass may free some other resources in its destructor (file handles, threads, transactions, COM references, mutexes) and so not calling it my cause severe resource locks.

If we use a smart pointer, however, p will be cleaned up whenever it gets out of scope, whether it was during the normal path of execution or during the stack unwinding caused by throwing an exception.

But isn't it possible to write exception safe code with regular pointers? Sure, but it is so painful that I doubt anyone actually does this when there is an alternative. Here is what you would do in this simple case:

```
void foo()
{
    MyClass* p;
    try {
        p = new MyClass;
        p->DoSomething();
        delete p;
    }
    catch (...) {
        delete p;
        throw;
    }
}
```

Now imagine what would happen if we had some if's and for's in there...

Why: *Garbage collection*

Since C++ does not provide automatic garbage collection like some other languages, smart pointers can be used for that purpose. The simplest garbage collection scheme is reference counting or reference linking, but it is quite possible to implement more sophisticated garbage collection

schemes with smart pointers. For more information see the garbage collection FAQ.

Why: *Efficiency*

Smart pointers can be used to make more efficient use of available memory and to shorten allocation and deallocation time.

A common strategy for using memory more efficiently is copy on write (COW). This means that the same object is shared by many COW pointers as long as it is only read and not modified. When some part of the program tries to modify the object ("write"), the COW pointer creates a new copy of the object and modifies this copy instead of the original object. The standard string class is commonly implemented using COW semantics (see the <string> header).

```
string s("Hello");

string t = s;       // t and s point to the same buffer of characters

t += " there!";     // a new buffer is allocated for t before
                    // appending " there!", so s is unchanged.
```

Optimized allocation schemes are possible when you can make some assumptions about the objects to be allocated or the operating environment. For example, you may know that all the objects will have the same size, or that they will all live in a single thread. Although it is possible to implement optimized allocation schemes using class-specific new and delete operators, smart pointers give you the freedom to choose whether to use the optimized scheme for each object, instead of having the scheme set for all objects of a class. It is therefore possible to match the allocation scheme to different operating environments and applications, without modifying the code for the entire class.

Why: *STL containers*

The C++ standard library includes a set of containers and algorithms known as the standard template library (STL). STL is designed to be *generic* (can be used with any kind of object) and *efficient* (does not incur time overhead compared to alternatives). To achieve these two design goals, STL containers store their objects by value. This means that if you have an STL container that stores objects of class Base, it cannot store of objects of classes derived from Base.

```
class Base { /*...*/ };
class Derived : public Base { /*...*/ };

Base b;
Derived d;
vector<Base> v;

v.push_back(b); // OK
v.push_back(d); // error
```

What can you do if you need a collection of objects from different classes? The simplest solution is to have a collection of pointers:

```
vector<Base*> v;

v.push_back(new Base);       // OK
v.push_back(new Derived);    // OK too

// cleanup:
for (vector<Base*>::iterator i = v.begin(); i != v.end(); ++i)
    delete *i;
```

The problem with this solution is that after you're done with the container, you need to manually cleanup the objects stored in it. This is both error prone and not exception safe.

Smart pointers are a possible solution, as illustrated below. (An alternative solution is a smart container, like the one implemented in pointainer.h.)

```
vector< linked_ptr<Base> > v;
v.push_back(new Base);       // OK
v.push_back(new Derived);    // OK too

// cleanup is automatic
```

Since the smart pointer automatically cleans up after itself, there is no need to manually delete the pointed objects.

Note: STL containers may copy and delete their elements behind the scenes (for example, when they resize themselves). Therefore, all copies of an element must be equivalent, or the wrong copy may be the one to survive all this copying and deleting. This means that some smart pointers cannot be used within STL containers, specifically the standard auto_ptr and any ownership-transferring pointer. For more info about this issue, see C++ Guru of the Week #25.

Which one should I use?

Are you confused enough? Well, this summary should help.

Which: *Local variables*

The standard auto_ptr is the simplest smart pointer, and it is also, well, standard. If there are no special requirements, you should use it. For local variables, it is usually the right choice.

Which: *Class members*

Although you can use auto_ptr as a class member (and save yourself the trouble of freeing objects in the destructor), copying one object to another will nullify the pointer, as illustrated Below.

```
class MyClass
{
    auto_ptr<int> p;
    // ...
};

MyClass x;
// do some meaningful things with x
MyClass y = x; // x.p now has a NULL pointer
```

Using a copied pointer instead of auto_ptr solves this problem: the copied object (y) gets a new copy of the member.

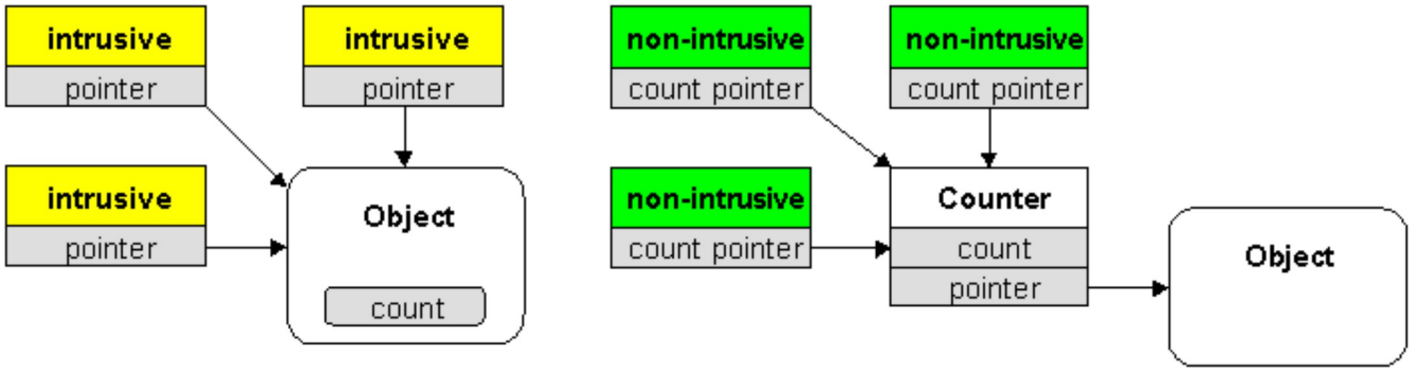
Note that using a reference counted or reference linked pointer means that if y changes the member, this change will also affect x! Therefore, if you want to save memory, you should use a COW pointer and not a simple reference counted/linked pointer.

Which: *STL containers*

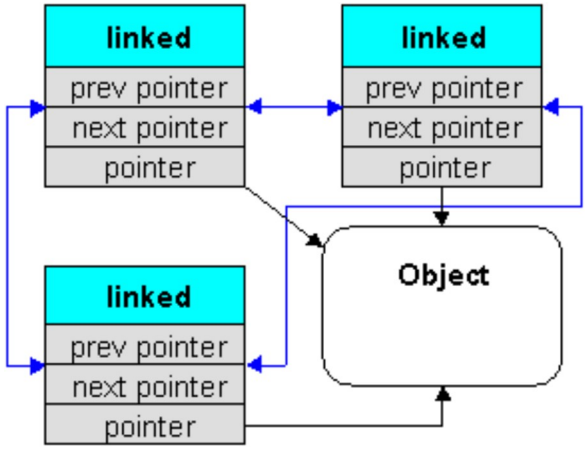
As explained above, using garbage-collected pointers with STL containers lets you store objects from different classes in the same container.

It is important to consider the characteristics of the specific garbage collection scheme used. Specifically, reference counting/linking can leak in the case of circular references (i.e., when the pointed object itself contains a counted pointer, which points to an object that contains the original counted pointer). Its advantage over other schemes is that it is both simple to implement and deterministic. The deterministic behavior may be important in some real time systems, where you cannot allow the system to suddenly wait while the garbage collector performs its housekeeping duties.

Generally speaking, there are two ways to implement reference counting: intrusive and non-intrusive. Intrusive means that the pointed object itself contains the count. Therefore, you cannot use intrusive reference counting with 3-rd party classes that do not already have this feature. You can, however, derive a new class from the 3-rd party class and add the count to it. Non-intrusive reference counting requires an allocation of a count for each counted object. The counted_ptr.h is an example of non-intrusive reference counting.



Reference linking does not require any changes to be made to the pointed objects, nor does it require any additional allocations. A reference linked pointer takes a little more space than a reference counted pointer - just enough to store one or two more pointers.



Both reference counting and reference linking require using locks if the pointers are used by more than one thread of execution.

Which: *Explicit ownership transfer*

Sometimes, you want to receive a pointer as a function argument, but keep the ownership of this pointer (i.e. the control over its lifetime) to yourself. One way to do this is to use consistent naming-conventions for such cases. Taligent's Guide to Designing Programs recommends using "adopt" to mark that a function adopts ownership of a pointer.

Using an owned pointer as the function argument is an explicit statement that the function is taking ownership of the pointer.

Which: *Big objects*

If you have objects that take a lot of space, you can save some of this space by using COW pointers. This way, an object will be copied only when necessary, and shared otherwise. The sharing is implemented using some garbage collection scheme, like reference counting or linking.

Which: *Summary*

For this:	**Use that:**
Local variables	auto_ptr
Class members	Copied pointer
STL Containers	Garbage collected pointer (e.g. reference counting/linking)

Explicit ownership transfer	Owned pointer
Big objects	Copy on write

Conclusion

Smart pointers are useful tools for writing safe and efficient code in C++. Like any tool, they should be used with appropriate care, thought and knowledge. For a comprehensive and in depth analysis of the issues concerning smart pointers, I recommend reading Andrei Alexandrescu's chapter about smart pointers in his book Modern C++ Design.

Feel free to use my own smart pointers in your code.
The Boost C++ libraries include some smart pointers, which are more rigorously tested and actively maintained. Do try them first, if they are appropriate for your needs.
