A smart pointer is a class that wraps a 'raw' (or 'bare') C++ pointer, to manage the lifetime of the object being pointed to. There is no single smart pointer type, but all of them try to abstract a raw pointer in a practical way.

Smart pointers should be preferred over raw pointers. If you feel you need to use pointers (first consider if you *really* do), you would normally want to use a smart pointer as this can alleviate many of the problems with raw pointers, mainly forgetting to delete the object and leaking memory.

With raw pointers, the programmer has to explicitly destroy the object when it is no longer useful.

// Need to create the object to achieve some goal MyObject\* ptr = new MyObject(); ptr->DoSomething(); // Use the object in some way delete ptr; // Destroy the object. Done with it. // Wait, what if DoSomething() raises an exception...?

A smart pointer by comparison defines a policy as to when the object is destroyed. You still have to create the object, but you no longer have to worry about destroying it.

SomeSmartPtr<MyObject> ptr(new MyObject()); ptr->DoSomething(); // Use the object in some way. // Destruction of the object happens, depending // on the policy the smart pointer class uses. // Destruction would happen even if DoSomething() // raises an exception

The simplest policy in use involves the scope of the smart pointer wrapper object, such as implemented by [boost::scoped\_ptr](http://www.boost.org/doc/libs/release/libs/smart_ptr/scoped_ptr.htm) or [std::unique\_ptr](http://en.cppreference.com/w/cpp/memory/unique_ptr).

void f() { { boost::scoped\_ptr<MyObject> ptr(new MyObject()); ptr->DoSomethingUseful(); } // boost::scopted\_ptr goes out of scope -- // the MyObject is automatically destroyed. // ptr->Oops(); // Compile error: "ptr" not defined // since it is no longer in scope. }

Note that scoped\_ptr instances cannot be copied. This prevents the pointer from being deleted multiple times (incorrectly). You can, however, pass references to it around to other functions you call.

Scoped pointers are useful when you want to tie the lifetime of the object to a particular block of code, or if you embedded it as member data inside another object, the lifetime of that other object. The object exists until the containing block of code is exited, or until the containing object is itself destroyed.

A more complex smart pointer policy involves reference counting the pointer. This does allow the pointer to be copied. When the last "reference" to the object is destroyed, the object is deleted. This policy is implemented by [boost::shared\_ptr](http://www.boost.org/doc/libs/release/libs/smart_ptr/shared_ptr.htm) and [std::shared\_ptr](http://en.cppreference.com/w/cpp/memory/shared_ptr).

void f() { typedef std::shared\_ptr<MyObject> MyObjectPtr; // nice short alias MyObjectPtr p1; // Empty { MyObjectPtr p2(new MyObject()); // There is now one "reference" to the created object p1 = p2; // Copy the pointer. // There are now two references to the object. } // p2 is destroyed, leaving one reference to the object. } // p1 is destroyed, leaving a reference count of zero. // The object is deleted.

Reference counted pointers are very useful when the lifetime of your object is much more complicated, and is not tied directly to a particular section of code or to another object.

There is one drawback to reference counted pointers — the possibility of creating a dangling reference:

// Create the smart pointer on the heap MyObjectPtr\* pp = new MyObjectPtr(new MyObject()) // Hmm, we forgot to destroy the smart pointer, // because of that, the object is never destroyed!

Another possibility is creating circular references:

struct Owner { boost::shared\_ptr<Owner> other; }; boost::shared\_ptr<Owner> p1 (new Owner()); boost::shared\_ptr<Owner> p2 (new Owner()); p1->other = p2; // p1 references p2 p2->other = p1; // p2 references p1 // Oops, the reference count of of p1 and p2 never goes to zero! // The objects are never destroyed!

To work around this problem, both Boost and C++11 have defined a weak\_ptr to define a weak (uncounted) reference to a shared\_ptr.

**UPDATE**

This answer is rather old, and so describes what was 'good' at the time, which was smart pointers provided by the Boost library. Since C++11, the standard library has provided sufficient smart pointers types, and so you should favour the use of [std::unique\_ptr](http://en.cppreference.com/w/cpp/memory/unique_ptr), [std::shared\_ptr](http://en.cppreference.com/w/cpp/memory/shared_ptr) and [std::weak\_ptr](http://en.cppreference.com/w/cpp/memory/weak_ptr).

There is also [std::auto\_ptr](http://en.cppreference.com/w/cpp/memory/auto_ptr). It is very much like a scoped pointer, except that it also has the "special" dangerous ability to be copied — which also unexpectedly transfers ownership! **It is deprecated in the newest standards, so you shouldn't use it. Use the** [**std::unique\_ptr**](http://en.cppreference.com/w/cpp/memory/unique_ptr)**instead.**

std::auto\_ptr<MyObject> p1 (new MyObject()); std::auto\_ptr<MyObject> p2 = p1; // Copy and transfer ownership. // p1 gets set to empty! p2->DoSomething(); // Works. p1->DoSomething(); // Oh oh. Hopefully raises some NULL pointer exception.