C++ Smart Pointers

They are crucial to the *RAII* or *Resource Acquisition Is Initialialization* programming idiom. In practical terms, the main principle of RAII is to give ownership of any heap-allocated resource—for example, dynamically-allocated memory or system object handles—to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated cleanup code.

In modern C++, raw pointers are only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership.

Access the encapsulated pointer by using the familiar pointer operators, -> and \*, which the smart pointer class overloads to return the encapsulated raw pointer.

|  |
| --- |
| **Important** |
| Always create smart pointers on a separate line of code, never in a parameter list, so that a subtle resource leak won't occur due to certain parameter list allocation rules.  See Effective C++ item 18 for the reasons. |

void SmartPointerDemo()  
{   
 // Create the object and pass it to a smart pointer  
 std::unique\_ptr<LargeObject> pLarge(new LargeObject());  
  
 //Call a method on the object  
 pLarge->DoSomething();  
  
 // Pass a reference to a method.  
 ProcessLargeObject(\*pLarge);  
  
} //pLarge is deleted automatically when function block goes out of scope.

void ProcessLargeObject(const LargeObject& lo){}

//Note the type conversion here.

Smart pointers have their own member functions, which are accessed by using “dot” notation. For example, some STL smart pointers have a reset member function that releases ownership of the pointer. This is useful when you want to free the memory owned by the smart pointer before the smart pointer goes out of scope, as shown in the following example.

**C++**

void SmartPointerDemo2()  
{  
 // Create the object and pass it to a smart pointer  
 std::unique\_ptr<LargeObject> pLarge(new LargeObject());  
  
 //Call a method on the object  
 pLarge->DoSomething();  
  
 // Free the memory before we exit function block.  
 pLarge.reset();  
  
 // Do some other work...  
  
}

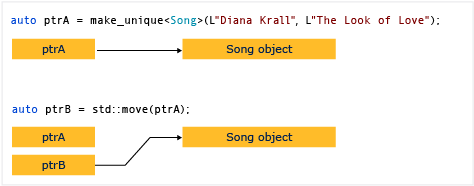
Smart pointers usually provide a way to access their raw pointer directly.

## C++ Standard Library Smart Pointers

Use these smart pointers as a first choice for encapsulating pointers to plain old C++ objects (POCO).

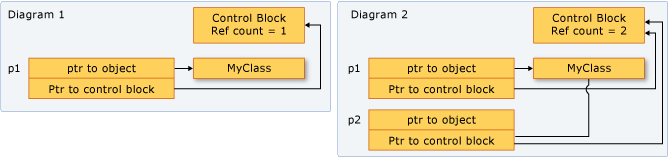
**unique\_ptr**

* Allows exactly one owner of the underlying pointer. Use as the default choice for POCO unless you know for certain that you require a **shared\_ptr**. Can be moved to a new owner, but not copied or shared. Replaces **auto\_ptr**, which is deprecated. **unique\_ptr** is small and efficient; the size is one pointer and it supports rvalue references for fast insertion and retrieval from STL collections.



**shared\_ptr**

* Reference-counted smart pointer. Use when you want to assign one raw pointer to multiple owners, for example, when you return a copy of a pointer from a container but want to keep the original. The raw pointer is not deleted until all **shared\_ptr** owners have gone out of scope or have otherwise given up ownership. The size is two pointers; one for the object and one for the shared control block that contains the reference count.



**weak\_ptr**

* Special-case smart pointer for use in conjunction with **shared\_ptr**. A **weak\_ptr** provides access to an object that is owned by one or more**shared\_ptr** instances, but does not participate in reference counting. Use when you want to observe an object, but do not require it to remain alive.

## Smart Pointers for COM Objects (Classic Windows Programming)

When you work with COM objects, wrap the interface pointers in an appropriate smart pointer type. The Active Template Library (ATL) defines several smart pointers for various purposes. You can also use the **\_com\_ptr\_t** smart pointer type, which the compiler uses when it creates wrapper classes from .tlb files. It's the best choice when you do not want to include the ATL header files.

## ATL Smart Pointers for POCO Objects

In addition to smart pointers for COM objects, ATL also defines smart pointers, and collections of smart pointers, for plain old C++ objects. In classic Windows programming, these types are useful alternatives to the STL collections, especially when code portability is not required or when you do not want to mix the programming models of STL and ATL.

We recommend that you restrict an object to one owner, because multiple ownership adds complexity to the program logic. Therefore, when you need a smart pointer for a plain C++ object, use **unique\_ptr**, and when you construct a **unique\_ptr**, use the [make\_unique](http://msdn.microsoft.com/en-us/library/dn439780.aspx) helper function.

The following example shows how to create **unique\_ptr** instances and pass them between functions.

**C++**

unique\_ptr<Song> SongFactory(const std::wstring& artist, const std::wstring& title){  
 // Implicit move operation into the variable that stores the result.   
 return make\_unique<Song>(artist, title);  
}  
  
void MakeSongs()  
{  
 // Create a new unique\_ptr with a new object.  
 auto song = make\_unique<Song>(L"Mr. Children", L"Namonaki Uta");  
  
 // Use the unique\_ptr.  
 vector<wstring> titles = { song->title };  
  
 // Move raw pointer from one unique\_ptr to another.  
 unique\_ptr<Song> song2 = std::move(song);  
  
 // Obtain unique\_ptr from function that returns by value.  
 auto song3 = SongFactory(L"Michael Jackson", L"Beat It");  
}

These examples demonstrate this basic characteristic of **unique\_ptr**: it can be moved, but not copied. "Moving" transfers ownership to a new **unique\_ptr** and resets the old **unique\_ptr**.

// Create a unique\_ptr to an array of 5 integers.  
 auto p = make\_unique<int[]>(5);

Whenever possible, use the [make\_shared (<memory>)](http://msdn.microsoft.com/en-us/library/ee410595.aspx) function to create a **shared\_ptr** when the memory resource is created for the first time. **make\_shared** is exception-safe. It uses the same call to allocate the memory for the control block and the resource, and thereby reduces the construction overhead

/ Use make\_shared function when possible.  
auto sp1 = make\_shared<Song>(L"The Beatles", L"Im Happy Just to Dance With You");  
  
// Ok, but slightly less efficient.   
// Note: Using new expression as constructor argument   
// creates no named variable for other code to access.  
shared\_ptr<Song> sp2(new Song(L"Lady Gaga", L"Just Dance"));

//Initialize with copy constructor. Increments ref count.  
auto sp3(sp2);  
  
//Initialize via assignment. Increments ref count.  
auto sp4 = sp2;  
  
//Initialize with nullptr. sp7 is empty.  
shared\_ptr<Song> sp7(nullptr);  
  
// Initialize with another shared\_ptr. sp1 and sp2   
// swap pointers as well as ref counts.  
sp1.swap(sp2);

SharePoint in STL:

vector<shared\_ptr<Song>> v;  
  
v.push\_back(make\_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"));  
...  
vector<shared\_ptr<Song>> v2;  
remove\_copy\_if(v.begin(), v.end(), back\_inserter(v2), [] (shared\_ptr<Song> s)   
{  
 return s->artist.compare(L"Bob Dylan") == 0;   
});  
  
for (const auto& s : v2)  
{  
 wcout << s->artist << L":" << s->title << endl;  
}

You can use dynamic\_pointer\_cast, static\_pointer\_cast, and const\_pointer\_cast to cast a **shared\_ptr**. These functions resemble the dynamic\_cast,static\_cast, and const\_cast operators.

vector<shared\_ptr<MediaAsset>> assets;  
  
assets.push\_back(shared\_ptr<Song>(new Song(L"Himesh Reshammiya", L"Tera Surroor")));  
assets.push\_back(shared\_ptr<Song>(new Song(L"Penaz Masani", L"Tu Dil De De")));  
assets.push\_back(shared\_ptr<Photo>(new Photo(L"2011-04-06", L"Redmond, WA", L"Soccer field at Microsoft.")));  
  
vector<shared\_ptr<MediaAsset>> photos;  
  
copy\_if(assets.begin(), assets.end(), back\_inserter(photos), [] (shared\_ptr<MediaAsset> p) -> bool  
{  
 // Use dynamic\_pointer\_cast to test whether   
 // element is a shared\_ptr<Photo>.  
 shared\_ptr<Photo> temp = dynamic\_pointer\_cast<Photo>(p); //Just like dynamic cast   
 return temp.get() != nullptr;  
});

equality of shared ptr.

// Unrelated shared\_ptrs are never equal.  
wcout << "p1 < p2 = " << std::boolalpha << (p1 < p2) << endl;  
wcout << "p1 == p2 = " << std::boolalpha <<(p1 == p2) << endl;  
  
// Related shared\_ptr instances are always equal.  
shared\_ptr<Song> p3(p2);  
wcout << "p3 == p2 = " << std::boolalpha << (p3 == p2) << endl;

* Pass the **shared\_ptr** by value. This invokes the copy constructor, increments the reference count, and makes the callee an owner. There is a small amount of overhead in this operation, which may be significant depending on how many **shared\_ptr** objects you are passing. Use this option when the code contract (implied or explicit) between the caller and callee requires that the callee be an owner.
* Pass the **shared\_ptr** by reference or const reference. In this case, the reference count is not incremented, and the callee can access the pointer as long as the caller does not go out of scope. Or, the callee can decide to create a **shared\_ptr** based on the reference, and thereby become a shared owner. Use this option when the caller has no knowledge of the callee, or when you must pass a **shared\_ptr** and want to avoid the copy operation for performance reasons.
* Pass the underlying pointer or a reference to the underlying object. This enables the callee to use the object, but does not enable it to share ownership or extend the lifetime. If the callee creates a **shared\_ptr** from the raw pointer, the new **shared\_ptr** is independent from the original, and does not control the underlying resource. Use this option when the contract between the caller and callee clearly specifies that the caller retains ownership of the **shared\_ptr**lifetime.
* When you are deciding how to pass a **shared\_ptr**, determine whether the callee has to share ownership of the underlying resource. An "owner" is an object or function that can keep the underlying resource alive for as long as it needs it. If the caller has to guarantee that the callee can extend the life of the pointer beyond its (the function's) lifetime, use the first option. If you don't care whether the callee extends the lifetime, then pass by reference and let the callee copy it or not.
* If you have to give a helper function access to the underlying pointer, and you know that the helper function will just use the pointer and return before the calling function returns, then that function does not have to share ownership of the underlying pointer. It just has to access the pointer within the lifetime of the caller's **shared\_ptr**. In this case, it is safe to pass the **shared\_ptr** by reference, or pass the raw pointer or a reference to the underlying object. Passing this way provides a small performance benefit, and may also help you express your programming intent.
* Sometimes, for example in a **std:vector<shared\_ptr<T>>**, you may have to pass each **shared\_ptr** to a lambda expression body or named function object. If the lambda or function is not storing the pointer, then pass the **shared\_ptr** by reference to avoid invoking the copy constructor for each element.

When cyclic references are unavoidable, or even preferable for some reason, use weak\_ptr to give one or more of the owners a weak reference to another shared\_ptr. You can use a weak\_ptr to try to obtain a new copy of the shared\_ptr with which it was initialized. If the memory has already been deleted, abad\_weak\_ptr exception is thrown. If the memory is still valid, the new shared pointer increments the reference count and guarantees that the memory will be valid as long as the shared\_ptr variable stays in scope.

for COM, use **CComPtr** instead.