A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

# Smart Pointers in C++

Consider the following simple C++ code with normal pointers.

|  |
| --- |
| *MyClass \*ptr = new MyClass();*  *ptr->doSomething();*  *//* ***We must do delete(ptr) to avoid memory leak*** |

Using [smart pointers](http://en.wikipedia.org/wiki/Smart_pointer), we can make pointers to work in way that we don’t need to explicitly call delete. [Smart pointer](http://en.wikipedia.org/wiki/Smart_pointer) is a wrapper class over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more.  
The idea is to make a class with a pointer, destructor and [overloaded operators](http://geeksquiz.com/operator-overloading-c/) like \* and ->. Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically deleted (or reference count can be decremented). Consider the following simple smartPtr class.

|  |
| --- |
| #include<iostream>  using namespace std;    class SmartPtr  {     int \*ptr;  // Actual pointer  public:     // Constructor: Refer <https://www.geeksforgeeks.org/g-fact-93/>     // for use of explicit keyword     explicit SmartPtr(int \*p = NULL) { ptr = p; }       // Destructor     ~SmartPtr() { delete(ptr); }       // Overloading dereferencing operator     int &operator \*() {  return \*ptr; }  };    int main()  {      SmartPtr ptr(new int());      \*ptr = 20;      cout << \*ptr;        // We don't need to call delete ptr: when the object      // ptr goes out of scope, destructor for it is automatically      // called and destructor does delete ptr.        return 0;  } |

Run on IDE

Output:

20

**Can we write one smart pointer class that works for all types?**  
Yes, we can use [templates](http://geeksquiz.com/templates-cpp/) to write a generic smart pointer class. Following C++ code demonstrates the same.

|  |
| --- |
| #include<iostream>  using namespace std;    // A generic smart pointer class  template <class T>  class SmartPtr  {     T \*ptr;  // Actual pointer  public:     // Constructor     explicit SmartPtr(T \*p = NULL) { ptr = p; }       // Destructor     ~SmartPtr() { delete(ptr); }       // Overloading dereferncing operator     T & operator \* () {  return \*ptr; }       // Overloding arrow operator so that members of T can be accessed     // like a pointer (useful if T represents a class or struct or     // union type)     T \* operator -> () { return ptr; }  };    int main()  {      SmartPtr<int> ptr(new int());      \*ptr = 20;      cout << \*ptr;      return 0;  } |

Run on IDE

Output:

20

Smart pointers are also useful in management of resources, such as file handles or network sockets.

C++ libraries provide implementations of smart pointers in the form of [auto\_ptr](http://en.wikipedia.org/wiki/Auto_ptr" \t "_blank), [unique\_ptr](http://en.wikipedia.org/wiki/Smart_pointer" \l "unique_ptr" \t "_blank), [shared\_ptr and weak\_ptr](http://en.wikipedia.org/wiki/Smart_pointer" \l "shared_ptr_and_weak_ptr" \t "_blank)

# auto\_ptr

**auto\_ptr** type is provided by the C++ standard library as a sort of smart pointer that helps to avoid resource leaks when exceptions are thrown.

Here is a typical example which has potential of memory leak.

void memory\_leak()

{

ClassA \* ptr = new ClassA;

...

delete ptr;

}

The reason why this function is source of trouble is that the deletion of the object might be forgotten especially if we have **return** inside of it. Also an exception would exit the function before the **delete** statement at the end of the function causing a resource leak.

Usually, we do try to capture all exceptions as in the example below.

void memory\_leak()

{

ClassA \* ptr = new ClassA;

try {

...

}

catch(...) {

delete ptr;

throw;

}

delete ptr;

}

As we see in the example, trying to handle the deletion of this object properly in the event of an exception makes the code more complicated and redundant.

So, we need a pointer which can free the data to which it points whenever the pointer itself gets destroyed. Because the pointer is a local variable, it will be destroyed automatically when the function is exited regardless of whether the exit is normal or caused by an exception.

In other words, if an exception occurs after successful memory allocation but before the delete statement executes, a memory leak could occur. The C++ standard provides class template auto\_ptr in header file <memory> to deal with this situation.

Our **auto\_ptr** is a pointer that serves as **owner** of the object to which it refers. So, an object gets destroyed automatically when its **auto\_ptr** gets destroyed.

The function in the 1st example can be rewritten using **auto\_ptr**

#include <memory>

void memory\_leak()

{

std::auto\_ptr<ClassA> ptr(new ClassA);

...

}

The **delete** statement and **catch** clause are no longer needed.

An **auto\_ptr** has the same interface as an ordinary pointer. Operator \* dereferences the object and operator -> provides access to a member if the object is a class or a structure.

But the pointer arithmetic such as ++ is not defined.

One more thing we should be careful about the usage of the pointer is that **auto\_ptr** does not allow us to initialize an object with an ordinary pointer by using the assignment syntax. So, we must initialize the **auto\_ptr** directly by using **its value**.

std::auto\_ptr<ClassA> ptr1(new ClassA); // RIGHT

std::auto\_ptr<ClassA> ptr1 = new ClassA; // WRONG

Here is the example of **auto\_ptr** in action:

#include <iostream>

#include <memory>

using namespace std;

class Double

{

public:

Double(double d = 0) : dValue(d) { cout << "constructor: " << dValue << endl; }

~Double() { cout << "destructor: " << dValue << endl; }

void setDouble(double d) { dValue = d; }

private:

double dValue;

};

int main()

{

auto\_ptr<Double> ptr(new Double(3.14));

(\*ptr).setDouble(6.28);

return 0;

}

The example creates **auto\_ptr** object **ptr** and initializes it with a pointer to a dynamically allocated **Double** object.

Because **ptr** is a local automatic variable in **main()**, **ptr** is destroyed when main terminates. The auto\_ptr destructor forces a delete of the Double object pointed to by **ptr**, which in turn calls the Double class destructor. The memory that Double occupies is released. The Double object will be deleted automatically when the auto\_ptr object's destructor gets called.

Only **one** auto\_ptr at a time can **own** a dynamically allocated object. Thus, the object cannot be an array. By using its overloaded assignment operator or copy constructor, an auto\_ptr can transfer ownership of the dynamic memory it manages. The last auto\_ptr object that maintains the pointer to the dynamic memory will delete the memory. This makes auto\_ptr an ideal mechanism for returning dynamically allocated memory to client code. When the auto\_ptr goes out of scope in the client code, the auto\_ptr's destructor deletes the dynamic memory.

Though **std::auto\_ptr** is responsible for managing dynamically allocated memory and automatically calls delete to free the dynamic memory when the auto\_ptr is destroyed or goes out of scope, auto\_ptr have some limitations.

1. An auto\_ptr can't point to an array. When deleting a pointer to an array we must use **delete[]** to ensure that destructors are called for all objects in the array, but auto\_ptr uses **delete**.
2. It can't be used with the STL containers-elements in an STL container. When an auto\_ptr is copied, ownership of the memory is transferred to the new auto\_ptr and the original is set to NULL. In other words, auto\_ptrs don't work in STL containers because the containers, or algorithms manipulating them, might copy the stored elements. Copies of auto\_ptrs aren't equal because the original is set to NULL after being copied. An STL container may make copies of its elements, so you can't guarantee that a valid copy of the auto\_ptr will remain after the algorithm processing the container's elements finishes.

An **auto\_ptr** is simply an object that holds a pointer for us within a function. Holding a pointer to guarantee deletion at the end of a scope is what **auto\_ptr** is for, and for other uses requires very specialized skills from a programmer.

The **Boost.Smart\_ptr library** provides additional smart pointers to fill in the gaps where auto\_ptrs don't work. TR1 includes two of the six types of smart pointers in the Boost.Smart\_ptr library, namely **shared\_ptr** and **weak\_ptr**. These smart pointers are not meant to replace auto\_ptr. Instead, they provide additional options with different functionality.

# smart pointers

A **smart pointer** is an **object that acts like a pointer** for most intents and purposes but avoids most of the problems inherent with C++ pointers. At its simplest, a smart pointer contains a native **pointer** as a data member and provides a set of **overloaded operators** that make it act like a pointer in most ways. Pointers can be dereferenced, so the \* and -> operators are overloaded to return the address as expected. Pointers can undergo pointer arithmetic operations, so the +, -, ++, and -- operators are also overloaded appropriately.

Because a smart pointer is an **object**, it can contain additional meta-data and take additional steps not possible with a regular pointer. For example, a smart pointer might contain information that allows it to recognize when the object to which it points has been deleted and start returning a NUll if so.

Smart pointers can also help with object lifetime management by cooperating with one another to determine the number of references to a particular object. This is called **reference counting**. When the number of smart pointers that reference a particular object drops to zero, we know that the object is no longer needed, so it can be automatically deleted. This can free the programmer from having to worry about object ownership and orphaned object (an object that still occupied memory but is no longer needed or referenced by any other object in the system).

Smart pointers have their share of problems. For one thing, they are relatively easy to implement, but they are extremely tough to get right. There are a great many cases to handle, and the [std::auto\_ptr](http://www.bogotobogo.com/cplusplus/autoptr.php" \t "_blank) class provided by the standard C++ library is widely recognized to be inadequate in many situations, and it's now deprecated.

1. **shared\_ptr**

The **auto\_ptr** has unusual characters: copying it whether via copy constructor or copy assignment operator sets it to null, and the copying pointer assumes ownership of the resource as we see in the example below:

#include <iostream>

#include <memory>

using namespace std;

class A{};

int main()

{

auto\_ptr<A> pA(new A);

cout << pA.get() << endl;

auto\_ptr<A> pB(pA);

cout << pA.get() << endl;

cout << pB.get() << endl;

return 0;

}

Output is:

001B0950

00000000

001B0950

In the example, the **get()** method returns a pointer to the object pointed by the auto\_ptr object, if any, or zero if it does not point to any object.

Note that the second output is **null**. So, in the copy constructor, **pA** transferred the ownership of **A** object to **pB**.

This behavior and the underlying requirement that resources managed by **auto\_ptr**s must never have more than one **auto\_ptr** pointing to them, means that **auto\_ptr**s aren't the best way to handle resources which are dynamically allocated.

So, as an alternative to **auto\_ptr**, we have a **referencing-counting smart pointer**. It keeps track of how many objects point to a particular resource and deletes the resource automatically when nothing is pointing to it.

By replacing **auto\_ptr** with **share\_ptr**, with an almost same code below, it produces the output we want to:

#include <boost/smart\_ptr/shared\_ptr.hpp>

#include <iostream>

#include <memory>

class A{};

int main()

{

boost::shared\_ptr<A> pA(new A);

std::cout << pA.get() << std::endl;

boost::shared\_ptr<A> pB(pA);

std::cout << pA.get() << std::endl;

std::cout << pB.get() << std::endl;

return 0;

}

Output is:

002C0950

002C0950

002C0950

Since copying **boost::shared\_ptr** works as we expect, it can be used in **STL** containers while we cannot use **std::auto\_ptr** for STL containers.

The major problem being solved using **share\_ptr** is to know the correct time to delete a resource that is shared. The following example has two classes, **A** and **B**. The classes are sharing an instance of **int**, and store a **shared\_ptr<int>**. When we create the instances of each class, the **shared\_ptr pTemp** is passed to the constructors. In other words, all three **shared\_ptr**s, are now referring to the same instance of an **int**. If we had used pointers to achieve such sharing of an **int**, each class would have had a hard time figuring out when it should be deleted. In the example, until the end of **main()**, the reference count is **3**. If all of the pointers go out of scope, the reference count reaches **0**, allowing the shared instance of **int** to be deleted.

**shared\_ptr** holds an internal pointer to a resource such as a dynamically allocated object that may be shared with other objects in the program. We can have any number of**shared\_ptr**s to the same resource. **shared\_ptr** really does share the resource, if we change the resource with one shared\_ptr, the changes also will be seen by the other**shared\_ptr**s. The internal pointer is deleted once the last **shared\_ptr** to the resource is destroyed. **shared\_ptr** uses **reference counting** to determine how many **shared\_ptr**s point to the resource. Each time a new **shared\_ptr** to the resource is created, the reference count increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted and the memory is released.

#include <boost/smart\_ptr/shared\_ptr.hpp>;

#include <iostream>

#include <memory>

class classA

{

boost::shared\_ptr<int> ptA;

public:

classA(boost::shared\_ptr<int> p) : ptA(p) {}

void setValue(int n) {

\*ptA = n;

}

};

class classB

{

boost::shared\_ptr<int> ptB;

public:

classB(boost::shared\_ptr<int> p) : ptB(p) {}

int getValue() const {

return \*ptB;

}

};

int main()

{

boost::shared\_ptr<int> pTemp(new int(2013));

classA a(pTemp);

classB b(pTemp);

a.setValue(2014);

std::cout << "b.getValue() = " << b.getValue() << std::endl;

return 0;

}

Output is:

b.getValue() = 2014

**shared\_ptr** also allows us to determine how the resource will be destroyed. For most dynamically allocated objects, **delete** is used. However, some resources require more complex cleanup. In that case, we can supply a custom deleter function, or function object, to the **shared\_ptr** destructor. The deleter determines how to destroy the resource. When the reference count reaches zero and the resource is ready to be destroyed, the **shared\_ptr** calls the **custom deleter function**. This functionality enables a **shared\_ptr** to manage almost any kind of resource.

For **dynamically allocated arrays**, we shouldn't use either of them because they use **delete** in their destructor but not **delete[]**. We can use **vector** instead. If we insist on using boost, we can use either **boost::shared\_array** or **boost::scoped\_array**.