Smart Pointers in C++

Introduction

What are smart pointers? The answer is fairly simple; a smart pointer is a pointer which is smart. What does that mean? Actually, smart pointers are objects which behave like pointers but do more than a pointer. These objects are flexible as pointers and have the advantage of being an object (like constructor and destructors called automatically). A smart pointer is designed to handle the problems caused by using normal pointers (hence called smart).

Problems with pointers

What are the common problems we face in C++ programs while using pointers? The answer is memory management. Have a look at the following code:

```
char* pName  = new char[1024];
…
SetName(pName);
…
…
if(null != pName)
{
       delete[] pName;
}
```

How many times have we found a bug which was caused because we forgot to delete `pName`. It would be great if someone could take care of releasing the memory when the pointer is not useful (we are not talking about the garbage collector here). What if the pointer itself takes care of that? Yes, that's exactly what smart pointers are intended to do. Let us write a smart pointer and see how we can handle a pointer better.

We shall start with a realistic example. Let's say we have a class called `Person` which is defined as below.

```
class Person
{
    int age;
    char* pName;

    public:
        Person(): pName(0),age(0)
        {
        }
        Person(char* pName, int age): pName(pName), age(age)
        {
```

```
        }
        ~Person()
        {
        }

        void Display()
        {
            printf("Name = %s Age = %d \n", pName, age);
        }
        void Shout()
        {
            printf("Ooooooooooooooooo",);
        }
};
```

Now we shall write the client code to use `Person`.

```
void main()
{
    Person* pPerson  = new Person("Scott", 25);
    pPerson->Display();
    delete pPerson;
}
```

Now look at this code, every time I create a pointer, I need to take care of deleting it. This is exactly what I want to avoid. I need some automatic mechanism which deletes the pointer. One thing which strikes to me is a destructor. But pointers do not have destructors, so what? Our smart pointer can have one. So we will create a class called `SP` which can hold a pointer to the `Person` class and will delete the pointer when its destructor is called. Hence my client code will change to something like this:

```
void main()
{
    SP p(new Person("Scott", 25));
    p->Display();
    // Dont need to delete Person pointer..
}
```

Note the following things:

- We have created an object of class `SP` which holds our `Person` class pointer. Since the destructor of the `SP` class will be called when this object goes out of scope, it will delete the `Person` class pointer (as its main responsibility); hence we don't have the pain of deleting the pointer.
- One more thing of major importance is that we should be able to call the `Display` method using the `SP` class object the way we used to call using the `Person` class pointer, i.e., the class should behave exactly like a pointer.

Interface for a smart pointer

Since the smart pointer should behave like a pointer, it should support the same interface as pointers do; i.e., they should support the following operations.

- Dereferencing (operator *)
- Indirection (operator ->)

Let us write the SP class now.

```
class SP
{
private:
    Person*    pData; // pointer to person class
public:
    SP(Person* pValue) : pData(pValue)
    {
    }
    ~SP()
    {
        // pointer no longer requried
        delete pData;
    }

    Person& operator* ()
    {
        return *pData;
    }

    Person* operator-> ()
    {
        return pData;
    }
};
```

This class is our smart pointer class. The main responsibility of this class is to hold a pointer to the Person class and then delete it when its destructor is called. It should also support the interface of the pointer.

Generic smart pointer class

One problem which we see here is that we can use this smart pointer class for a pointer of the Person class only. This means that we have to create a smart pointer class for each type, and that's not easy. We can solve this problem by making use of templates and making this smart pointer class generic. So let us change the code like this:

```
template < typename T > class SP
{
    private:
```

```
    T*    pData; // Generic pointer to be stored
    public:
    SP(T* pValue) : pData(pValue)
    {
    }
    ~SP()
    {
        delete pData;
    }

    T& operator* ()
    {
        return *pData;
    }

    T* operator-> ()
    {
        return pData;
    }
};

void main()
{
    SP<PERSON> p(new Person("Scott", 25));
    p->Display();
    // Dont need to delete Person pointer..
}
```

Now we can use our smart pointer class for any type of pointer. So is our smart pointer really smart? Check the following code segment.

```
void main()
{
    SP<PERSON> p(new Person("Scott", 25));
    p->Display();
    {
        SP<PERSON> q = p;
        q->Display();
        // Destructor of Q will be called here..
    }
    p->Display();
}
```

Look what happens here. `p` and `q` are referring to the same `Person` class pointer. Now when `q` goes out of scope, the destructor of `q` will be called which deletes the `Person` class pointer. Now we cannot call `p->Display();` since `p` will be left with a dangling pointer and this call will fail. (Note that this problem would have existed even if we were using normal pointers instead of smart pointers.) We should not delete the `Person` class pointer unless no body is using it. How do we do that? Implementing a reference counting mechanism in our smart pointer class will solve this problem.

Reference counting

What we are going to do is we will have a reference counting class RC. This class will maintain an integer value which represents the reference count. We will have methods to increment and decrement the reference count.

```
class RC
{
    private:
    int count; // Reference count

    public:
    void AddRef()
    {
        // Increment the reference count
        count++;
    }

    int Release()
    {
        // Decrement the reference count and
        // return the reference count.
        return --count;
    }
};
```

Now that we have a reference counting class, we will introduce this to our smart pointer class. We will maintain a pointer to class RC in our SP class and this pointer will be shared for all instances of the smart pointer which refers to the same pointer. For this to happen, we need to have an assignment operator and copy constructor in our SP class.

```
template < typename T > class SP
{
private:
    T*    pData;       // pointer
    RC* reference; // Reference count

public:
    SP() : pData(0), reference(0)
    {
        // Create a new reference
        reference = new RC();
        // Increment the reference count
        reference->AddRef();
    }

    SP(T* pValue) : pData(pValue), reference(0)
    {
        // Create a new reference
        reference = new RC();
        // Increment the reference count
        reference->AddRef();
    }
```

```
    SP(const SP<T>& sp) : pData(sp.pData), reference(sp.reference)
    {
        // Copy constructor
        // Copy the data and reference pointer
        // and increment the reference count
        reference->AddRef();
    }

    ~SP()
    {
        // Destructor
        // Decrement the reference count
        // if reference become zero delete the data
        if(reference->Release() == 0)
        {
            delete pData;
            delete reference;
        }
    }

    T& operator* ()
    {
        return *pData;
    }

    T* operator-> ()
    {
        return pData;
    }

    SP<T>& operator = (const SP<T>& sp)
    {
        // Assignment operator
        if (this != &sp) // Avoid self assignment
        {
            // Decrement the old reference count
            // if reference become zero delete the old data
            if(reference->Release() == 0)
            {
                delete pData;
                delete reference;
            }

            // Copy the data and reference pointer
            // and increment the reference count
            pData = sp.pData;
            reference = sp.reference;
            reference->AddRef();
        }
        return *this;
    }
};
```

Let us have a look at the client code.

```
void main()
{
    SP<PERSON> p(new Person("Scott", 25));
```

```
    p->Display();
    {
        SP<PERSON> q = p;
        q->Display();
        // Destructor of q will be called here..

        SP<PERSON> r;
        r = p;
        r->Display();
        // Destructor of r will be called here..
    }
    p->Display();
    // Destructor of p will be called here
    // and person pointer will be deleted
}
```

When we create a smart pointer `p` of type `Person`, the constructor of `SP` will be called, the data will be stored, and a new `RC` pointer will be created. The `AddRef` method of `RC` is called to increment the reference count to 1. Now `SP<person> q = p;</person>` will create a new smart pointer `q` using the copy constructor. Here the data will be copied and the reference will again be incremented to 2. Now `r = p;` will call the assignment operator to assign the value of `p` to `q`. Here also we copy the data and increment the reference count, thus making the count 3. When `r` and `q` go out of scope, the destructors of the respective objects will be called. Here the reference count will be decremented, but data will not be deleted unless the reference count becomes zero. This happens only when the destructor of `p` is called. Hence our data will be deleted only when no body is referring to it.

Applications

Memory leaks: Using smart pointers reduces the work of managing pointers for memory leaks. Now you could create a pointer and forget about deleting it, the smart pointer will do that for you. This is the simplest garbage collector we could think of.

Exceptions: Smart pointers are very useful where exceptions are used. For example, look at the following code:

```
void MakeNoise()
{
    Person* p = new Person("Scott", 25);
    p->Shout();
    delete p;
}
```

We are using a normal pointer here and deleting it after using, so every thing looks okay here. But what if our `Shout` function throws some exception? `delete p;` will never be called. So we have a memory leak. Let us handle that.

```
void MakeNoise()
{
```

```
    Person* p = new Person("Scott", 25);
    try
    {
        p->Shout();
    }
    catch(...)
    {
        delete p;
        throw;
    }
    delete p;
}
```

Don't you think this is an overhead of catching an exception and re-throwing it? This code becomes cumbersome if you have many pointers created. How will a smart pointer help here? Let's have a look at the same code if a smart pointer is used.

```
void MakeNoise()
{
    SP<Person> p(new Person("Scott", 25));
    p->Shout();
}
```

We are making use of a smart pointer here; yes, we don't need to catch the exception here. If the `Shout` method throws an exception, stack unwinding will happen for the function and during this, the destructor of all local objects will be called, hence the destructor of `p` will be called which will release the memory, hence we are safe. So this makes it very useful to use smart pointers here.

Conclusion

Smart pointers are useful for writing safe and efficient code in C++. Make use of smart pointers and take the advantage of garbage collection.