

References, Pointers, and “Smart” Pointers

Options when returning objects that are created by a procedure:

```
String get_token(istream& s); // return object by value
String& get_token(istream& s); // return object by reference
String* get_token(istream& s); // return object by pointer
```

Creating and returning an object from inside of a procedure involves some careful choices. You have to consider the run-time cost and “ownership” issues. For example:

```
String get_token(istream& s)
{
    String token;
    ... // read token from stream
    return token; // return by value implies copying
}
```

```
String x = get_token(cin);
```

We end up with two `String` copies in this case since the token object returned by `get_token` is first copied into a temporary location on the call stack, and then copied from the temporary to `x`. You can think of it as follows:

```
String x = String temp(token);
```

But what we want is for `x` to be constructed with the token object returned by `get_token`:

```
String x(get_token());
```

Some optimizing C++ compilers can optimize this to eliminate the extra temporary copy. In fact GNU g++ has a special flag for just this case, which eliminates (elides) the extra temporary constructor call.

```
g++ -felide-constructors ...
```

What if we return a pointer instead?

```
String* get_token(istream& s)
{
    String* token= new String;
    ... // get token from stream
    return token;
}
```

How/when does the `String` object get deleted? I.E., who owns the pointer?

```
String *s = get_token(cin);
...
delete s;
```

Problem is that you have to remember to delete the object, and this may be a problem if you pass the pointer around to other procedures or store it in some other data structure. Basically, pointer ownership can become ambiguous. Something has to delete it later, and also make sure that you did not stuff the pointer into a data structure somewhere that could result in a dangling pointer pointing to an object that you already deleted. What we would like is some kind of “Smart Pointer” that encapsulates the actual pointer and deals with it automatically.

```
StringPtr get_token(istream& s)
{
    String* s = new String;
    ...
    return s; // calls StringPtr(String*) constructor with s
}

// some scope in which get_token is called
StringPtr sp = get_token(cin);
...
sp->length(); // calls StringPtr::operator->(), then String::length()
} // StringPtr is deleted when scope is exited, and so is the String object
```

An alternative approach to returning an object by value:

```
String& get_token(istream& s)
{
    // what sort of String object do we use in this case:
    // static, stack or heap allocated?
    ... // get token from stream
    return token;
}
```

NEVER DO THE FOLLOWING!

```
String& get_token(istream& s)
{
    String* token = new String;
    ...
    return *token; // dereference pointer and return ref to object on heap
};
```

This means you have to do the following to delete the object:

```
String& s = get_token(cin);
...
String *p = &s; // take the address of the object that the reference points to
delete p;
```

The only time you probably want to return a reference to an object is in a member function of an object that returns a reference to itself.

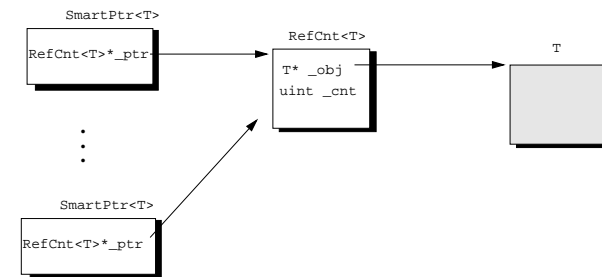
```
class Foo()
{
    Foo& self() { return *this; }
    ...
}
```

“Smart” Pointer Implementation

Multiple `SmartPtr<T>` template instances point to a single `RefCnt<T>` instance, which holds a pointer and a reference count to an instance of a class `T` (e.g., `String`).

The `SmartPtr<T>` template is a *handle* that is used to access a reference counted object and which updates the reference count.

- on construction of a `SmartPtr<T>` instance, the count is incremented
- on destruction the count is decremented.
- when the last `SmartPtr<T>` object is destructed, the `RefCnt<T>` is deleted, which in turn deletes the object of type `T`.



```

/* A ``smart pointer'' to a reference counted object */
template<class T> class SmartPtr {
    RefCnt<T>* _ptr; // pointer to a reference counted object of type T

    /* hide new/delete to disallow allocating a SmartPtr<T> from the heap */
    static void* operator new(size_t) {}
    static void operator delete(void*) {}

public:
    SmartPtr(T* p) { _ptr = new RefCnt<T>(p); _ptr->inc(); }

    SmartPtr(const SmartPtr<T>& p) {
        _ptr = p._ptr;
        _ptr->inc();
    }

    ~SmartPtr() { if (_ptr->dec() == 0) delete _ptr; }

    void operator=(const SmartPtr<T>& p) {
        if (this != &p) {
            if (_ptr->dec() == 0)
                delete _ptr;
            _ptr = p._ptr;
            _ptr->inc();
        }
    }

    T* operator->() const { return _ptr->object(); }
    T& operator*() const { return *(_ptr->object()); }
};

```

```

#include <String.h>
#include "Ref.h"

typedef SmartPtr<String> StringPtr;
typedef SmartPtr<Shape> ShapePtr;

main()
{
    StringPtr p = new String("Hello, world");
    StringPtr s = p; // invokes copy constructor

    /* invoke a String method on a StringPtr object */

    cout << "length is " << s->length() << endl;
    ...
    ShapePtr r = new Rectangle;
    ...
    r->draw();// invokes Rectangle::draw by invoking operator-> on r
    ...
};

```

The overloaded operator->() method is a unary operator applied to an object instance and is used here to access the methods of the String object by "calling through" the SmartPtr<String> and RefCnt<String> objects, which are just wrappers around the String object.

writing s->length() is equivalent to writing:

```

String* temp = s.operator->();
temp->length();
where s.operator->() calls _ptr->RefCnt<String>::object(), returning _obj as a String* value.

```

The operator->() method must return either an object that has operator->() defined or a pointer to which operator-> can be applied.

```

/* Ref.h - abstractions for reference counting garbage collection of objects */
#include <assert.h>
#include <iostream.h>
#include <new.h>

/* A reference count template that is used to encapsulate an object
 * that is to be reference counted (traced). The reference count is
 * manipulated using the inc/dec methods. The overloaded operator->
 * method provides a way to extract a pointer to the reference counted
 * object for the purpose of calling its methods.
 */

template<class T> class RefCnt {
private:
    T* _obj;
    unsigned long _cnt;

public:
    RefCnt(T* p) : _obj(p), _cnt(0) {}

    ~RefCnt() { assert(_cnt == 0); delete _obj; }

    T* object() const { return _obj; }

    int inc() { _cnt++; return _cnt; }
    int dec() { _cnt--; return _cnt; }
};

```

auto_ptr template in ANSI C++ (see section 8.4.2 in C++ Primer)

auto_ptr is a template defined in the standard C++ library in the <memory> header file. It acts like a smart pointer, but it provides a non-reference counting implementation. It uses a simple "ownership" rule to determine which auto_ptr instance has responsibility for deleting a pointer. auto_ptr can be used with any object allocated via new, except for arrays.

```

auto_ptr<string> get_token(istream& is )
{
    auto_ptr<string> token = new string;
    ... // get token from istream
    return token;
}

```

auto_ptr<string> x = get_token(cin); // x takes ownership of token string pointer

Each time an auto_ptr goes out of scope, a test is made to see if it owns the pointer. If so, then the pointer is deleted. Ownership is determined by which auto_ptr object was initialized with the pointer.

```

auto_ptr<string> f()
{
    auto_ptr<string> p (new string("hello")); // p owns pointer to hello object
    auto_ptr<string> q (new string("goodbye")); // q owns pointer to goodbye object
    p = q; // p deletes hello pointer, take ownership of goodbye pointer, q loses ownership
    return p;
}

```

NOTE: p and q are automatically destructed on exit from f(). Since q no longer owns a pointer, it's destructor does not delete anything. In returning p as the result, the auto_ptr<string> copy constructor is called, and p loses ownership as a result of it being copied out of the procedure scope, so the string object containing the characters "goodbye" is not deleted, ownership is passed back to the caller. IF using g++, compile with -felide-constructors to eliminate a copy.

An auto_ptr Template Implementation

```

template <class X> class auto_ptr {
private:
    // note that the sizeof(auto_ptr<X>) does not fit in a register!
    X* ptr;
    mutable bool owns; // see below for reason why 'owns' is declared mutable
public:
    typedef X element_type;

    auto_ptr() : ptr(0), owns(false) {}

    // explicit constructor declaration means cannot do implicit object construction
    explicit auto_ptr(X* p) : ptr(p), owns(true) {}

    auto_ptr(const auto_ptr<X>& a) : ptr(a.ptr), owns(a.owns) { a.owns = false; }

    void operator=(const auto_ptr<X>& that) { // const argument is intended to be immutable
        if (this != &that) {
            if (owns) delete ptr;
            owns = that.owns;
            ptr = that.ptr;
            that.owns = false; // 'owns' must be mutable for const that.owns to be set to false
        }
    }

    ~auto_ptr() { if (owns) delete ptr; }

    X& operator*() const { return *ptr; }
    X* operator->() const { return ptr; }
    X* get() const { return ptr; }
    X* release() const { owns = false; return ptr; }
    void reset(X* p) { if (owns) delete ptr; ptr = p; owns = (p != 0) ? true : false; }
};

```

auto_ptr usage examples

The auto_ptr template utilizes the ANSI C++ explicit keyword as a qualifier to a constructor. An explicit constructor requires that an object be initialized explicitly, rather than implicitly, when an explicit constructor is called. For example:

```

int *x = new int(5);
auto_ptr<int> p = x; // illegal, implicit call to auto_ptr<int>::auto_ptr(int*)
auto_ptr<int> q = new int(6); // illegal call

auto_ptr<int> p(x); // okay, explicit initialization with int*
auto_ptr<int> q(new int(6)); // okay, explicit initialization with int*

auto_ptr<int> r;
r = new int(10); // illegal -- cannot assign int* directly to r;
r.reset(new int(10)); // okay, can reset the auto_ptr

```

You can still end up with ambiguous ownership of a pointer if you are not careful. For example:

```

auto_ptr<string> s = new string("before"); // s owns string "before"
auto_ptr<string> t(s.get()); // now both s and t own string "before"!!

```

Instead, you should explicitly release ownership before assigning it to another auto_ptr

```

auto_ptr<string> t (s.release());

```

Just as with a smart pointer, an auto_ptr allows you to call methods using operator-> or dereference using operator*.

```

int len = t->length(); // calls t.operator->(), which results in call to ptr->length()
(*t).length(); // calls t.operator*, which results in call to (*ptr).length()

```