24 - Lecture - Advanced C++: smart pointer

Optional reading assignment

After you understand the SmartPtr class in this lecture note completely, you can optionally read ATC++ 11.2.

Smart pointer

What is it?

- A C++ class that behaves like a pointer.

Why?

- We want automatic life-time management of heap-allocated objects.

- We want value semantics without having to copy large objects.

- In short, we want Java reference!

SmartPtr class

- Light-weight handle for heap-allocated objects.

- Manages the object life-time using reference counting.

- Provides value semantics, thus can be put into standard containers.

SmartPtr definition

```
template <class T>
class SmartPtr {

    private:
        T *ptr;  // the underlying pointer

        int *count;  // the reference count

    public:

        // constructor
        //
        // - p is assumed to point to an object created by "new T(...)"
        // - we hold the pointer and initialize ref count to 1.
        //
        //   note: explicit keyword
        //   note: default argument
        //
        explicit SmartPtr(T *p = 0)
        {
            ptr = p;
    count = new int(1);
}

// copy constructor
//
// - copy the data members and increment the reference count
//
//   note: member initialization syntax
//
SmartPtr(const SmartPtr<T>& sp)
    : ptr(sp.ptr), count(sp.count)
{
    ++*count;
}

// destructor
//
// - delete the underlying object if this was the last owner
//
~SmartPtr()
{
    if (--*count == 0) {
        delete count;
        delete ptr;
    }
}

// assignment operator
//
// - detach this SmartPtr from the underlying object and
//   attach to the object that sp is pointing to.
//
SmartPtr<T>& operator=(const SmartPtr<T>& sp)
{
    if (this != &sp) {
        // first, detach.
        if (--*count == 0) {
            delete count;
            delete ptr;
        }
        // attach to the new object.
        ptr = sp.ptr;
        count = sp.count;
        ++*count;
    }
    return *this;
}

// operator*() and operator->() make SmartPtr class behave
// just like a regular pointer.

T& operator*() const { return *ptr; }

T* operator->() const { return ptr; }

// access to the underlying pointer for those cases when you
// need it.
```

```
        T* getPtr() const { return ptr; }

        // operator void*() makes "if (sp) ..." possible.

        operator void*() const { return ptr; }

};
```

Using SmartPtr

```
SmartPtr<vector<int> >  createLargeObject()
{
    SmartPtr<vector<int> >  pv(new vector<int>());

    for (int i = 0; i < 100000; i++)
        pv->push_back(i);

    return pv;
}

int main()
{
    SmartPtr<string> ps1;
    SmartPtr<string> ps2(new string("hello "));
    SmartPtr<string> ps3(new string("world "));
    ps1 = ps2 = ps3;
    // should print "world world world "
    cout << *ps1 << *ps2 << *ps3 << endl;

    SmartPtr<vector<int> >  pv1;
    pv1 = createLargeObject();
    for (int i = 0; i < 100000; i++) {
        assert(i == (*pv1)[i]);
    }

    // Note that we do not call delete anywhere in this code.

    return 0;
}
```

shared_ptr

The C++11 standard includes a smart pointer template class (finally!) called "shared_ptr":

- Works basically the same way as our SmartPtr, but more powerful.

- Atomic reference counting for thread safety.

- Can attach "weak_ptr", which does not participate in ref counting.

- Delete operation customizable.

Using shared_ptr (optional material)

```
// Sample code showing how to use shared_ptr for an array of objects.
//
//     Compile with clang (recent versions with c++11 support):
//         clang++ -std=c++11 -stdlib=libc++ shared_ptr-test.cpp
//
//     Or with older g++ (versions with at least c++0x support):
//         g++ -std=c++0x shared_ptr-test.cpp

#include <memory> // for shared_ptr
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

shared_ptr<string>  createLargeArray(size_t n)
{
    shared_ptr<string>  sp(new string[n],
            // shared_ptr takes an optional 2nd argument, a deleter,
            // which can be any callable object like
            // a function pointer, a function object, or a lamda.
            [] (string *a) {
                delete[] a;
            }
            );
    for (size_t i = 0; i < n; ++i) {
        // shared_ptr does not overload operator[],
        // so we need to apply &* to get back the underlying string*.
        (&*sp)[i] = "hello";
    }
    return sp;
}

int main()
{
    size_t n = 100000;
    shared_ptr<string>  p = createLargeArray(n);
    for (size_t i = 0; i < n; ++i) {
        assert((&*p)[i] == "hello");
    }
}
```