Advanced Programming in C++

Exercise – Smart pointers II

In this exercise you shall, given a simple class template definition for a smart pointer type, first implement the given member functions and then successively develop the class so its use, as close as possible, resembles the use of ordinary raw pointers, but with certain improved features.

More about smart pointers is found at InformIT network (15 web pages written by a real expert on C++, Andrei Alexandrescu): http://www.informit.com/articles/article.aspx?p=31529 (excerpt from the book Modern C++ Design, Generic Programming and Design Patterns Applied, Addison Wesley). Some of the thing to be done are really difficult and this text surely need to be consulted now and then.

Smart pointers

A smart pointer type shall basically have the same features as an ordinary pointer type, specifically **operator*** for dereferencing and **operator->** for indirection. In certain aspects we want smart pointers to have better qualities, e.g. that smart pointers always are initialized and that the memory for the referred objects is always deallocated automatically. Other important aspects concerns copying, assignment, typing (i.e. if and how different smart pointer types can be mixed in operations), and using constant and non-constant smart pointers in the same way as raw pointers.

There is a number of implementation models and optimizing techniques for smart pointers. The most known are *deep copy (copy-on-create/assign)*, *destructive copy*, *reference counting*, *reference linking* and *copy on write* (COW). One specific question concerning copying is if, and in such case how, copying can be performed for smart pointers to polymorphic objects. A base class pointer can point to an object of a derived class and the question is then if it is possible to make a correct copy.

Another question is, if it should be possible to initialize a smart pointer to a null pointer. In that case, should there be a default constructor for doing. Should it be allowed to explicitly initialize with a null pointer constant, 0?

The C++ standard library includes a smart pointer type, auto_ptr (deprecated in C++0x and replaced by other smart pointer types). It is implemented for *destructive copy*, which means that when an auto_ptr is copied to another auto_ptr, the former will be "destroyed". This actually means that the raw pointer is *transferred* to the destination (move semantics) and the raw pointer in the source is set to null. Example:

```
auto_ptr<int> sp1(new int(1));    // sp1 is initialized with a pointer to an int object
auto_ptr<int> sp2(sp1);           // sp2 takes over the object, sp1 becomes a null pointer
sp1 = sp2;                        // sp1 takes over the object, sp2 becomes null
```

The same kind of over-taking takes place, e.g., when a function having a value parameter of type auto_ptr is called. In this exercise you are going to implement a different semantics for smart pointers.

Introduction to the exercise

The exercise consist of six steps. Work “monolitically” the first five steps, i.e. define all functions, also non-members, inclass. In the last, sixth, step the code is to be divided upon two files, the include file and an accompanying implementation file with separate function definitions. The starting point for the exercise is the following class template smart_pointer.

```
template<typename T>
class smart_pointer
{
  public:
    smart_pointer(T* ptr = 0);
    smart_pointer(const smart_pointer<T>& other);
    ~smart_pointer();
    smart_pointer<T>& operator=(const smart_pointer<T>& rhs);
    T* operator*();
    T& operator->();
  private:
    T* ptr_;
    void copy(const smart_pointer<T>& ptr);
};
```

The type of objects smart_pointer can refer to is T, and the so called stored type is T*. Member functions for initialization, copying, destruction and the two basic pointer operations *dereferencing* (*) and *indirection* (->) are declared. The private function copy() shall implement the semantics for copying smart pointer objects, in connection to initialization, copy construction and copy assignment. Other member functions, performing operations regarding ownership of the referred object, shall use copy() for the purpose of code sharing and to make it easy to modify copy semantics.

Below are examples of declarations and expressions which shall or shall *not* be allowed. For comparisons, e.g., there are a number of symmetric cases not given.

```
smart_pointer<int> sp1(new int(1));    // Initialization with a raw pointer
smart_pointer<int> sp2(sp1);           // Copy constructor

sp1 = sp2;                    // Copy assignment

int* p = new int(2):
smart_pointer<int> sp(p);

delete p;                     // Problem – the destructor for sp shall to this to!
delete sp;                    // Shall not be allowed

if (sp) …                     // Direct null pointer test
if (!sp) …                    // Negated direct null pointer test.
if (sp == 0) …                // Explicit test if sp is a null pointer
if (0 == sp) …                // Symmetric case
if (sp != 0) …                // Explicit test if sp is not a null pointer
if (0 != sp) …                // Symmetric case

if (sp1 == sp2) …             // Comparison if two smart pointers are equal
if (sp1 != sp2) …             // Comparison if two smart pointers are not equal

smart_pointer<int> sp;
int* p = new int(3);
if (sp == p) …                // Comparing with raw pointer for equality
if (sp != p) …                // Comparing with raw pointer for inequality
```

If a smart pointer could point to an object of class type, the dot an arrow operators are used for accessing members.

```
spc->m          // operator-> (m is supposed to be a member of the class type in question)
(*spc).m        // operator* (the smart pointer is first dereferenced)
```

If two smart pointers points to different type of objects it should work as for raw pointers, i.e. what is allowed for raw pointers it should be allowed for corresponding smart pointers.

```
smart_pointer<A> spa;
smart_pointer<B> spb;
B* p;
if (spa == spb) …           // Shall work if it is a meaningful comparison, i.e. if
if (spa == p) …             // the comparison of corresponding raw pointers is allowed
```

If a smart pointer points to objects of polymorphic type, if possible, the same rules apply as for raw polymorphic pointers, i.e. pointers of base class type shall also be able to point to objects of subclass type, and that the rules otherwise works in the same way as for raw pointers.

```
smart_pointer<Base> spp;    // Base is a polymorphic base class
Derived* pd;                // Derived is a subclass to Base

if (spp == pd) …            // Should work
if (spp != pd) …
```

The purpose of this exercise is to make as many as possible of desired constructs to work for the smart pointer, at the same time as non-desired features (e.g applying **delete** to a smart pointer) as far as possible is avoided.

Testing

Basic functionality

- Initialization: constructor taking a raw pointer to an object, also 0 (default)
- Copying: copy constructor and copy assignment operator
- Destruction: deallocate the memory of the referred object
- Dereferencing operator, **operator***
- Indirection operator, **operator**-> (compile error, if the referred object is not of class type)

Destruction

- It shall not be possible to **delete** a smart pointer (compile error)

Null pointer tests

- Direct null pointer test, e.g. **if** (sp)
- Negated direct null pointer test, e.g., **if** (!sp)
- Explicit equality test for null pointers, sp == 0, 0 == sp
- Explicit inequality test for null pointers, sp != 0, 0 != sp

Equality and inequality tests (== and !=)

- Compare two smart pointer of the same type
- Compare on smart pointer and a raw pointer of the same type
- Compare two polymorphic smart pointers of different subtypes
- Compare a polymorphic smart pointer and a raw pointer of different subtypes

Keeping constness

- Assignment in different variants, for smart pointers, constant smart pointers, constant raw pointers, raw pointer to constant, and constant raw pointer to constant.

Strong typing

The following operation s shall *not* be allowed (compile error):

- Compare two smart pointers of incompatible types (the referred objects are of incompatible type)
- Compare a smart pointer with a raw pointer of incompatible type
- Assignment involving two smart pointers of incompatible type
- Assign a smart pointer with a raw pointer of incompatible type
- Initialize a smart pointer with a raw pointer of incompatible type
- Initialize a smart pointer with another smart pointer of incompatible type

Function calls

- Parameter passing: value passing (call by value), reference passing (call by reference)
- Return value: value passing (return by value), reference passing (return by reference)

Test program for most cases above is available, see the directory with given code (also found in the directories for each step).

Step 1 – Implementing the given member functions

Implement the member functions in the given class smart_pointer for copy on create/assign, i.e., deep copy.

Start with copy(), which is to implement the copy semantics. Then use copy() to implement the other member functions where copying is going to be performed.

- The helper function copy() shall make a copy of the object that the parameter ptr refers to and assign the pointer to that copy to ptr_ in the current smart_pointer object. The function shall return nothing. The parameter ptr_ can be a null pointer.

  Correct copying of polymorphic objects will be an unsolved issue, or an optional step!
- The constructor taking a raw pointer of type T* shall initialize the smart pointer object with that raw pointer. Default argument shall be 0.

  Is implicit type conversion to be allowed?
- The copy constructor shall implement copy on create/assign.
- The destructor shall deallocate the memory for the referred object, if there is such (ptr_ can be a null pointer.
- The copy assignment operator shall perform assignment, according to the semantics for copy on create/assign.
- **operator***() shall return a reference to the referred object.
- **operator**->() shall return a pointer to the referred object, i.e. ptr_.
- Support for operating on constant objects is left to step 4.

Test the implemented member functions to verify their basic functionality, and also how much of the desired semantics that is achieved by this first implementation, and also if there are any undesired effects.

There are two specific test programs for this step:

```
smart_pointer-test-basics.cpp
smart_pointer-test-function-calls.cpp
```

Which of the member functions of smart_pointer can be declared to not throw any exceptions, i.e. with an empty exception specification, **throws**()? Or, the comment "// throws()".

Step 2 – Implement basic null pointer tests and equality tests

A simple way to allow certain null pointer test and equality test (== and !=) would be to define implicit type conversion from smart_pointer till some other type. The following type conversion operators may be considered as being more or less natural:

- **operator** T*()
- **operator void***()
- **operator const void***()
- **operator bool**()

However, implicit type conversion often creates more problems than it solves. Type conversion may occur in situations where it is not at all desired, e.g. making it possible to apply **delete** to a smart pointer, or allow incompatible smart pointer to be mixed in expressions, or making it possible to use a smart pointer as it if were of arithmetic type. Such possibilities would break the semantics for smart pointers and making it possible to write nonsense code.

If one defines **operator** T*() for allowing implicit type conversion to a raw pointer type, one can also define **operator void***() to eliminate the possibility to use **delete** on smart pointers. Delete expressions will then be ambiguous, since it is not possible to choose one of these conversions as better than the other. But, this will only solve some problems and also looks a awkward.

If one overloads **operator**! to return **true**, if the smart pointer is a null pointer, otherwise **false**, some null pointer test will be possible, e.g:

```
if (!sp) …

if (!!sp) …        // A somewhat awkward way of testing if (sp) …
```

A test not possible with simple means (without using implicit type conversion) is direct null pointer test:

```
if (sp) …
```

There is an advanced trick for solving this, without allowing for **delete**, but we postpone this to later.

A good solution is to overload the operators !, == and !=. This will take some coding but it can be regarded as reasonable, since we will achieve much of what is desired, without undesired side effects.

Implement the following for smart_pointer instantiated for the same type T and corresponding raw pointer type T*:

- **operator**!, to return **true** if a smart pointer is a null pointer, otherwise **false**.
- **operator**== and **operator**!=. Comparison regarding equality and inequality is to be allowed for two smart pointers of the same type, or for one smart pointer and a raw pointer of corresponding type, or a smart pointer an a null pointer constant.

```
smart_pointer<T> sp, sp2;
T* p;

sp == sp2     sp != sp2

sp == p     sp != p     p == sp     p != sp
```

Test the operators. There is a specific test program for this step:

```
smart_pointer-test-compare.cpp
```

Test also with the test programs for the previous step.

Step 3 – Implement more null pointer tests and equality tests

In this step, comparisons regarding equality (==) and inequality (!=) shall also include the following.

- Comparison of two smart pointers, instantiated for different but comparable types, T and U. If comparison of the raw pointer types T* and U* is accepted, i.e. compile, comparison of the corresponding smart pointers is also to be accepted.
- Comparisons of a smart pointer instantiated for a type T, and a raw pointer of type U*, where U is another type than T. If comparison of the raw pointer types T* and U* is accepted, i.e. compile, comparison of the corresponding smart pointer and the raw pointer is also to be accepted.

```
smart_pointer<T> spt;
smart_pointer<U> spu;
T* pt;
U* pu;
```

If pt == pu (pt != pu) is allowed, also the following comparisons are to be allowed:

```
spt == spu    spt != spu

spt == pu     spt != pu        pu == spt     pu != spt
```

Smart pointers shall also in these respects follow the same rules that apply for corresponding raw pointers.

Implement and test. There is a specific test program for this step:

```
smart_pointer-test-compare-more.cpp
```

Test also with the test programs for the previous steps.

Step 4 – Smart pointers and const

For raw pointers we have the following possibilities to use **const** qualifiers::

```
T*             p;                    // pointer to T
const T*       pc;                   // pointer to constant T
T* const       cp  = new T;          // constant pointer to T
const T* const cpc = new const T;    // constant pointer to constant T
```

The following smart pointers should correspond to the raw pointers above:

```
smart_pointer<T>             sp;
smart_pointer<const T>       spc;
const smart_pointer<T>       csp(new T);
const smart_pointer<const T> cspc(new const T);
```

It is desired that smart pointers follow the same rules that apply to raw pointers, regarding also **const**. E.g., that a constant smart pointer must be initialized and cannot be modified thereafter, that a smart pointer to constant can point at either a constant or a non-constant object, etc.

Find out how much of **const** semantics that is already supported by the implementation so far. Complete and modify smart_pointer.

It may occur ambiguities and there are advanced solutions for such problems which are not obvious. See the referred text from Alexandrescu's book Modern C++ Design.

Implement and test. There is a specific test program for this step:

```
smart_pointer-test-const.cpp
```

Test also with the test programs for the previous steps.

The rules for raw pointers regarding **const** is demonstrated by the program on the file pointer-const-tests.cc.

Step 5 – Implementing direct null pointer test

Direct null pointer test refers to the possibility of writing code as:

```
if (sp1) …          // false if sp1 is a smart null pointer, true otherwise
```

As mentioned earlier, this can be made possible by defining implicit type conversion, e.g. to T*, **void***, **const void*** or **bool**. Such type conversions can introduce undesired effects, such as applying **delete** to a smart pointer, or using a smart pointer as an arithmetic type.

To make it possible to use a smart pointer where an expression of type **bool** is expected, implicit type conversion must be allowed. The question is then, what to convert to, to allow direct null pointer test but nothing more? The solution is an advanced trick, not easy to find out:

- Define a *private* nested class null_pointer_tester in smart_pointer:

```
class null_pointer_tester
{
  void operator delete(void*);  // private, so not callable
};
```

- Define a public type conversion operator, converting a smart pointer to null_pointer_tester*.

```
operator null_pointer_tester*() const
{
  static class null_pointer_tester test;
  if (!ptr)
    return 0;
  return &test;
}
```

If the smart pointer is a null pointer, i.e. ptr_ is 0, the type converting operator returns 0, otherwise it returns the address to a local static object of type null_pointer_tester. This makes it possible to write null pointer tests, without at the same time open for undesired effects.

- If a smart pointer is a null pointer, 0 is returned, which can be converted to **false**, in a context where a **bool** value is required.
- If a smart pointer is not a null pointer, an address which is not 0 is returned, which can be converted to **true**, in a context where a **bool** value is required.
- If **delete** is applied to the pointer returned by the type conversion operator, a compile error is issued, since **operator** delete is private in null_pointer_tester.
- null_pointer_tester is private, so nothing else can be done with it.

After this step, most of the desired operations for a smart pointer are available, and hopefully no undesired features will occur.

Implement and test. There is a specific test program for this step:

```
smart_pointer-test-null-pointer.cpp
```

Test also with the test programs for the previous steps.

Step 6 – Divide the code for the smart pointer

The idea is that we shall divide the code according to tradition, i.e. with the c lass definition and function declarations on an include file, and function definitions on an accompanying implementation file. Some compilers can handle this, in principle, in the “usual way”, others not. GNU GCC g++ used the so called *inclusion model*, which means that template code must either be written in the files where it is to be used (the most primitive model), or made available in the files where it is to be used by inclusion. Template code i just template code, so there is no problem with having not instantiated template code in several instances in the same translation.

Move all functions definitions to a separate file, e.g. named smart_pointer.tcc (t as in template), and replace them with their declarations in smart_pointer.h. To follow the inclusion model, let the include file include this file at its end, within a possible namespace for smart_pointer.

There may occur some complications, e.g. ambiguities, which should be possible to solve relatively simple and without further complications.

There is no specific test program for this step, use the ones for the previous steps to test that all still works.

At least one problem still remain – copying does not work for subtypes to a (polymorphic) type T.