# Advanced Programming in C++

## Exercise – Smart pointer I

In this exercise you shall, given a simple class definition for a smart pointer type, first implement the given member functions and then successively develop the class so the use of such smart pointers as much as possible resembles the use of ordinary raw pointers, but with certain improved features and implementing some common semantic model for smart pointers.

There is a more realistic exercise Smart pointer II. That exercise is about designing a more realistic smart pointer type, i.e. a class template, so the stored type can be chosen.

Basically it's a question of designing a single, non-trivial class and overloading pointer related operators. On the last page you will fins a list of test cases.

On the last page you will fins a list of test cases. Please note, some of these are relevant only if the smart pointer class is a template, and then different instances for more or less compatible types may occur.

## Smart pointers

A smart pointer type shall, basically, have the same features as an ordinary pointer type – initialization, copying, assignment, dereferencing (with `*`) and indirection (using `->`). In certain aspects we want a smart pointer to have other and/or better characteristics, e.g. that a smart pointer always is initialized and the memory for the referred object is always deallocated automatically. Other important aspects concerns copying and assignment and these may be implemented according to different models. Typing, to be able to use constant and non-constant smart pointers of a certain type in the same way as ordinary pointers are other aspects.

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, i.e., should there be a default constructor doing this, or 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 copy (move semantics) and the raw pointer in the copied auto_ptr 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, sp1 becomes a null pointer.
sp1 = sp2;                        // sp1 takes over, sp2 becomes a null pointer.
```

The same kind of over-taking takes place, e.g., when a function having a value parameter of type auto_ptr is called.

## Introduction to the exercise

Start out from the smart_pointer below:

```
class smart_pointer
{
  public:
    explicit smart_pointer(int* ptr = 0);
    smart_pointer(const smart_pointer& rhs);
    ~smart_pointer();
    smart_pointer& operator=(const smart_pointer& rhs);
    int& operator*();
    int* operator->();
  private:
    int* ptr_;
    void copy(const smart_pointer& ptr);
};
```

The type of objects smart_pointer can refer to is **int**, which, of course, should be a template type parameter T. Member functions for initialization, copying, destruction and the two basic pointer operations *dereferencing* (*) and *indirection* (->) are declared. The raw pointer to be referring to (pointing at) the object the smart pointer is responsible for is declared in the private part. The function copy shall implement the semantics for copying, 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 making 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, but only some examples are given.

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

sp1 = sp2;                          // Copy assignment

int* p = new int(2):
smart_pointer 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<X> sp;
X* p = new X;
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 should be 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)
```

## 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 when implementing 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.
- The constructor taking a raw pointer of type int* shall initialize the smart pointer object with the raw pointer. Default argument shall be 0.
- The copy constructor shall implement copy on create/assign.
- The destructor shall deallocate the memory for the referred object, if there is such.
- 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 the pointer ptr_.

**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.

One question when exceptions have been covered: 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()".

## 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 which is 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**, without using overloaded template versions, the following operators:

- **operator**!, to return **true** if a smart pointer is a null pointer, otherwise **false**.
- **operator**== and **operator**!= to compare, with respect to equality and inequality, two smart pointer of the same type, or one smart pointer and one corresponding raw pointer of corresponding type, or one smart pointer and a null pointer constant.

**Test** these operators.

## 3. Smart pointers and const

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

```
int*              p;                    // pointer to int
const int*        pc;                   // pointer to constant int
int* const        cp  = new int(1);     // constant pointer to int
const int* const  cpc = new int(2);     // constant pointer to constant int
```

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

Our smart pointer is of type **int***. Which of the pointer types above can consequently our smart pointer refer to?

The possibility of declaring constant smart pointers, e.g.

```
const smart_pointer csp0;
const smart_pointer csp1(new int(1));
```

has not yet been mentioned. If you haven't thought about that already, modify smart_pointer so one can operate also on constant smart pointer, using operations that should be possible to apply to constant smart pointer objects.

To design a more realistic smart pointer type, i.e. as a class template, se Smart pointer II.

## Testing

Please note: some cases are not really relevant unless the smart pointer type is a template.

### 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)