# C++ Templates - Smart Pointer

Object-oriented languages like Java and C# employ automatic *garbage collection*, which periodically deletes objects from the heap when they are no longer reachable, reclaiming the memory that they occupied. C++ does not have automatic garbage collection, and so the programmer is responsible for reclaiming memory by explicitly deleting objects which have been allocated on the heap. This allows programmers to write very memory efficient programs, but also increases the risk of introducing bugs such as *memory leaks*, which occur when an object is created on the heap, but never deleted.

We have seen in the lectures that *templates* make it easy to write C++ code (e.g. for extendable arrays, ordered sets, queues, stacks, etc.) in a generic and type-safe way: we are able to write the code once and *instantiate* it using many different types. In this tutorial, we will see that templates can also be used to write classes whose instances behave like pointers while taking care of garbage collection at the same time, relieving the programmer from the burden of memory management.

## 1 Implementation

The technique that we will use to automatically delete objects on the heap when they are no longer referenced is called *reference counting*. We will create a class that acts just like a standard pointer, and instances of this class will also keep track of the total number of instances referencing any given object at any particular time. Each time a smart pointer is created we increase the reference count of the object that it points to and each time a smart pointer is discarded (i.e. goes out of scope) we will decrease the reference count. If the reference count for an object reaches zero we will delete the object from the heap. There is one subtlety, however: if we have several smart pointers referencing the *same* object, then they all need to share the same reference count. This means that we cannot store the reference count in any one particular smart pointer instance, so instead we will store it on the *heap*, and give each instance a pointer to it.

We want our smart pointers to do everything that standard C++ pointers do - we want to be able to create them, copy them, reassign them, and we want them to be *type safe*.

We will implement our smart pointer in several steps:

1. Write a template class `SmartPtr`. It should have two (private) data members, one of which points to the object on the heap which the smart pointer is to reference, and the other which points to a integer (stored on the *heap*) which keeps a count of the number of references to the object (the *reference count*).

2. Write a constructor for the class which takes as an argument a pointer to the object that the smart pointer will reference. This constructor should also initialise the reference count to 1.

3. When the smart pointer object is destructed, make sure it decrements the reference count and then deletes the heap object if the reference count is zero.

4. Overload the assignment operator `=` which assigns the value of one smart pointer to another, i.e. *copies* a smart pointer, making the destination smart pointer to reference the same object as the source smart pointer. Be careful when implementing this: executing the expression `sp1 = sp2` should *discard* `sp1`'s reference to the object it previously referenced, and make it reference the same object as `sp2`, thus creating a *new* reference to that object. (You should also consider what should happen if you try to assign a smart pointer to itself!). You could also implement a *copy* constructor which carries out a similar process and creates a new smart pointer by copying an already existing one (make sure that you do not duplicate any code!).

5. Overload the indirection (dereference) operator `*` to return the stored object by reference. Also, overload the member access operator `->` to return a (standard) pointer to the stored object - this will allow your smart pointer class to transparently reference objects, and not just values of simple types (like `int` or `char`).

**Important:** Make sure that your code handles *null* pointers gracefully and efficiently!

## 2 Testing

Now write a `main` function to test out your smart pointer class. You might want to add another member function which returns the reference count, and then create several smart pointers which reference the same object checking that the reference count returned by each smart pointer has the correct value. You should also test that objects referenced by smart pointers are actually deleted from the heap when the last smart pointer reference to them is removed.

Make sure to instantiate your template class using at least two different types (for example, you could choose a primitive type like `int` or `float`, or you could use object types) and check that the compiler still enforces type safety: that is, if the two types are incompatible so that you cannot assign a (standard) pointer to an object of one type to a pointer of the other type, then you cannot assign a smart pointer of one type to a smart pointer of another type.

## 3 Extra Tasks

1. There are some situations in which our implementation above will not quite function correctly. Can you create an object referenced by a smart pointer which is *not* deleted from the heap when it is no longer reachable by the main program code? (Hint: you need to create a *cycle* of pointers in memory). The C++ standard libraries provides several different types of smart pointer to avoid this 'bug': a `weak_ptr` acts like a smart pointer but does not contribute to the reference count, so they can be used when pointer cycles are necessary.

2. There are other ways to implement reference counting. For example, instead of storing the reference count separately on the heap, you could store it as a *data member* of the object being referenced. In this way, objects are responsible for keeping track of their *own* reference count. Can you write an alternative implementation of a smart pointer which uses this technique? (Hint: write a *base* class which objects should inherit from that takes care of the reference counting logic; a disadvantage to this approach is that smart pointers can only point to objects that inherit from your base class).

3. Research and find out what existing support (e.g. libraries) there is for smart pointers in C++. As a start you could look at the Standard for Programming Language C++ (`http://www.open-std.org/`), or you could investigate the Boost libraries (`http://www.boost.org/`).