Exercise 3 Smart Pointers

1. (First Encounters with Smart Pointers: *Unique Pointers*)
Consider the following code that uses raw pointers:

```
{ // Block with raw pointer lifecycle

   double* d = new double (1.0);
   Point* pt = new Point(1.0, 2.0);     // Two-d Point class

   // Dereference and call member functions
   *d = 2.0;
   (*pt).X(3.0);        // Modify x coordinate
   (*pt).Y(3.0);        // Modify y coordinate

   // Can use operators for pointer operations
   pt->X(3.0);          // Modify x coordinate
   pt->Y(3.0);          // Modify y coordinate

   // Explicitly clean up memory (if you have not forgotten)
   delete d;
   delete pt;
}
```

Answer the following questions:

a) Type, run and test the above code. Introduce a *try-catch* block in which memory is allocated and directly afterwards an exception is thrown. You need to throw an exception in the body of the code. Since the code is not *re-entrant*, the memory is not reclaimed and hence it introduces a *memory leak* (in more general cases it would be a *resource leak*).
b) Now port the above code by replacing raw pointers by `std::unique_ptr`. Run the code. Are there memory leaks now?
c) Experiment with the following: initialising two unique pointers to the same pointer, assigning a unique pointer to another unique pointer and *transferring ownership* from one unique pointer to another unique pointer.
d) Use *alias template* (template typedef) to make the code more readable.

2. (Shared Pointers)
The objective of this exercise is to show *shared ownership* using smart pointers in C++.
Create two classes `C1` and `C2` that share a common heap-based object `d` as data member:

```
std::shared_ptr<double> d;
```

Answer the following questions:

a) Create the code for the classes `C1` and `C2` each of which contains the shared object `d` above, for example:

```
class C1
{
private:
      //double* d; OLD WAY
      std::shared_ptr<double> d;
public:
      C1(std::shared_ptr<double> value) : d(value) {}
      virtual ~C1() { cout << "\nC1 destructor\n";}
      void print() const { cout << "Value " << *d; }
};
```

The interface for class `C2` is similar to that of `C1`.

b) Create instances of these classes in scopes so that you can see that resources are being automatically released when no longer needed. To this end, employ the member function `use_count()` to keep track of the number of shared owners. This number should be equal to 0 when the program exits.

c) Carry out the same exercise as in steps a) and b) but with a user-defined type as shared data:

```
std::shared_ptr<Point> p;
```

d) Now extend the code in parts a) to c) to include the following operations on shared pointers: assigning, copying and comparing two shared pointers `sp1` and `sp2`. Furthermore, test the following features (some research needed here):
   - Transfer ownership from `sp1` to `sp2`.
   - Determine if a shared pointer is the only owner of a resource.
   - Swap `sp1` and `sp2`.
   - Give up ownership and reinitialise the shared pointer as being empty.

3. (Custom Deleter)
Shared and unique pointers support deleters. A *deleter* is a callable object that executes some code before an object goes out of scope. A deleter can be seen as a kind of *callback function*. We first give a simple example to show what we mean: we create two-dimensional points as smart pointers. Just before a point goes out of scope a callback function will be called:

```
auto deleter = [](Point* pt)-> void
               { std::cout << "Bye:" << *pt; };
```

The corresponding code is:

```
using SmartPoint = std::shared_ptr<Point>;
SmartPoint p1(new Point(), deleter);
```

We now discuss the exercise. To this end, answer the following questions:

a) The goal of this exercise is to open a file, write some records to the file and then close it when we are finished. Under normal circumstances we are able to close the file. However, if an exception occurs before the file has been closed the file pointer will remain open and hence it cannot be accessed. In order to ensure *exception safety* we employ a shared pointer with a deleter in the constructor, for example by using a function object:

```
std::shared_ptr<FILE> mySharedFile(myFile, FileFinalizer());
```

where `FileFinalizer` is a function object. Similar to the deleter that we gave of the beginning at the exercise.

b) Create a free function and a stored lambda function that also play the role of custom deleters for this problem.
c) Test the code for the three kinds of deleter function (function object, free function, lambda function).
d) Create a small loop in which records are added to the file; throw an exception at some stage in the loop, catch the exception and then open the file again. Does it work?

4. (Weak Pointers)
A *weak pointer* is an observer of a shared pointer. It is useful as a way to avoid dangling pointers and also when we wish to use shared resources without assuming ownership.

Answer the following questions:

a) Create a shared pointer, print its use count and then create a weak pointer that observes it. Print the use count again. What are the values?
b) Assign a weak pointer to a shared pointer and check that the weak pointer is not empty.
c) Assign a weak pointer to another weak pointer; assign a weak pointer to a shared pointer. What is the use count in both cases?

5. (Alias Template and its Advantages compared to `typedef`)
The keyword `typedef` does not work with templates (at least not directly) and we need to do a lot of contortions when working with template classes. To this end, we create a class that is composed of a container:

```
template <typename T>
   class Client
{ // An example of Composition
private:
   typename Storage<T>::type data; // typename mandatory
public:
   Client(int n, const T& val) : data(n, val) {}

   void print() const
   {
      std::for_each(data.begin(), data.end(), [](const T& n)
                    { std::cout << n << ","; });
      std::cout << '\n';
   }
};
```

where the storage ADT is:

```
// C++03 approach
// Data storage types
template <typename T> struct Storage
{
   // Possible ADTs and their memory allocators
   // typedef std::vector<T, CustomAllocator<T>> type;
   // typedef std::vector<T, std::allocator<T>> type;

   typedef std::list<T, std::allocator<T>> type;
};
```

An example of use is:

```
// Client of storage using typedef
int n = 10; int val = 2;
Client<int> myClient(n, val); myClient.print();
```

Answer the following questions:

a) Create the class using alias template instead of `typedef`.
b) Test your code. Do you get the same output as before? What are the advantages?