5.1 - An Introduction to the Boost C++ Libraries

Goals and Objectives

The objective of these exercises is to get you acquainted with the essentials of the Boost library, including installation (see guide), understanding the Boost syntax in general and learning to understand and use some specific Boost libraries that are useful for the material in Level 9 (Financial Applications).

I recommend the following process:

a) Listen to the audio by Daniel Duffy discussing the Boost demo code.
b) Compile and run the code; make sure you understand the code.
c) Start and complete the exercises (they are closely related to the demos and you may copy the demo code and modify it to suit your needs)
d) For more information, see www.boost.org (Math Toolkit that has excellent online documentation) and possibly the 2-volume set on Boost by Robert Demming and Daniel J. Duffy.

Exercise 1: Smart Pointers

When we want to store different kind of shapes in the *Array<T>* class we created earlier, we need to store *Shape** in the array. But when we are finished, we need to delete all shapes in the array explicitly which is easy to forget.

The *boost::shared_ptr<T>* class stores a pointer and will delete the object automatically when nobody is referencing the object anymore. Thus instead of creating an array of *Shape** we can create an array with *boost::shared_ptr<Shape>* and the deletion of the shapes will be done automatically.

Thus create a program that creates an array with shared pointers for shapes (The template array class and shape hierarchy was created in earlier exercises). Fill it with various shapes and print them. Check if the shapes are automatically deleted.

Tip:
Use the following typedefs to simplify the code:

```
// Typedef for a shared pointer to shape and
// a typedef for an array with shapes stored as shared pointers.
typedef boost::shared_ptr<Shape> ShapePtr;
typedef Array<ShapePtr> ShapeArray;
```

Exercise 2: Tuple

Boost tuples can be used to store different kinds of data as one entity. It can be used if we need to combine data without the need to create a separate class.

In this exercise, create a typedef for a *Person* tuple that contains a name, age and length. Also create a function that prints the person tuple. Use the *get<T>()* member functions the retrieve the data. Create a few person tuple instances and print them.

Also increment the age of one of the persons. Note that to change a value of one of the tuple elements, you can also use the *get<T>()* function since it returns a reference to the value.

Exercise 3: Variant

In contrast to tuple, a *boost::variant* stores one value that can be of one of the specified types. It is thus similar to a C union but it is type-safe.

In this exercise we want to create a function that ask the user what kind of shape to create and returns that. In this case we can return the created shape as *Shape** because all shapes have a common base class but that involves creating the shapes with *new*. Instead we can return the created shape a *boost::variant* which would also be needed if the shapes did not have a common base class.
Thus create a typedef for a *ShapeType* variant that can contain a *Point*, *Line* or *Circle*. Next create a function that returns the variant. Within this function ask the user for the shape type to create. Then create the requested shape and assign it to the variant and return it.

In the main program, call the function and print the result by sending it to *cout*. Next try to assign the variant to a *Line* variable by using the global *boost:get<T>()* function. This will throw a *boost::bad_get* exception when the variant didn't contain a line.
Test the application.

Checking what kind a type is stored in the variant is cumbersome. Therefore the *boost::variant* supports visitors. A visitor is a class derived from *boost::static_visitor<T>* that has for each type that can be stored an *operator ()* with the type as argument. The template argument is the return type of the *operators ()*.
Now create a variant visitor that moves the shapes. The visitor is derived from *boost::static_visitor<void>* and must have members for the x- and y-offset that are set in the constructor. For each shape, create an operator () that changes the coordinates of the shape. For example the function for *Point* is defined as:

```
// Visit a point.
void operator () (Point& p) const
{
  p.X(p.X()+m_dx);
  p.Y(p.Y()+m_dy);
}
```

In the main program, create an instance of the visitor and use the *boost::apply_visitor(visitor, variant)* global function to move the shape. Print the shape afterwards to check if the visitor indeed changed the coordinates.

Exercise 4: Random Number Generation

In this example we simulate dice throwing. We use functionality in Random library by creating a discrete uniform random number generator whose outcomes are in the closed range [1,6]:

```
// Throwing dice.
// Mersenne Twister.
boost::random::mt19937 myRng;

// Set the seed.
myRng.seed(static_cast<boost::uint32_t> (std::time(0)));

// Uniform in range [1,6]
boost::random::uniform_int_distribution<int> six(1,6);
```

We now create a map that holds the frequency of each outcome:

```
map<int, long> statistics; // Structure to hold outcome + frequencies
int outcome;               // Current outcome
```

A typical outcome is generated as follows:

```
      outcome = six(myRng);
```

This will generate a number in the range [1,6].

Answer the following questions:

a) Generate a large number of trials and place their frequencies in map.
b) Produce the following kind of output:

```
How many trials? 1000000

Trial 1 has 16.6677% outcomes
Trial 2 has 16.6551% outcomes
Trial 3 has 16.6881% outcomes
Trial 4 has 16.7103% outcomes
Trial 5 has 16.6273% outcomes
Trial 6 has 16.6515% outcomes
```

Exercise 5: Statistical Functions

In this exercise we experiment with the exponential and Poisson distributions that have many applications to queueing theory, scheduling and jump (Levy) processes in finance. Some typical examples of use are:

```
double scaleParameter = 0.5;

// Default type is 'double'
exponential_distribution<> myExponential(scaleParameter);

cout << "Mean:" << mean(myExponential)
     << ",standard deviation: "
     << standard_deviation(myExponential) << endl;
```

and

```
double mean = 3.0;
poisson_distribution<double> myPoisson(mean);
```

Modify the code in the file "TestNormalDistribution.cpp" to work with exponential distribution instead of normal distribution and Poisson distribution instead of gamma distribution.