Chapter 13

The Boost Library: An Introduction

13.1 Introduction and Objectives

In the previous chapters of Part II we introduced system decomposition techniques to partition a system into more manageable subsystems (POSA 1996), namely in chapters 8 and 9. We then designed each subsystem using design patterns (GOF 1995), namely chapter 10. Eventually, the design is detailed enough to allow it to be implemented in C++ as discussed in chapters 11 and 12.

In this chapter we give an introduction to the boost library (www.boost.org). The library will become part of the official C++ standard in due course and the authors of this book plan to migrate their code to support these libraries in the future. This chapter is an overview of what the boost library offers.

Using the boost library offers a number of advantages:

- Functionality: the library has functionality for many of the data structures and algorithms that we can directly use in applications. It is suitable for scientific and engineering applications and in particular it is already popular with quantitative developers. If boost does not have the functionality you need you can consider writing new code yourself or using a (possibly proprietary) third-party software product. Finally, the interfaces are consistent and standardised.

- Usability: boost is easy to install, use and apply in your applications. It uses templates and many of the design techniques and patterns that we introduced in Part II of the current book, in particular chapters 10 and 11.

- Efficiency: boost uses templates and compile-time structures. This makes it very efficient because all data and object types are known before the program starts (that is, run-time). The concepts in the object-oriented paradigm - such as subtype polymorphism - are used sparingly if not at all.

- Maintainability: the burden of maintaining, debugging and extending the boost library has been removed from the user's shoulders to the developers of the boost library. New improved versions of the library can be downloaded from the boost Web site.

- Portability: the boost library is vendor-independent and operating-system independent. This means that you can port Windows applications to linux applications with minimal effort, for example. We now give an overview of the features in the boost library; first, we discuss smart pointers in detail because it is vital to use them if you wish to write reliable C++ applications.

13.2 A Taxonomy of C++ Pointer Types

One of the most difficult topics to master in C++ is heap memory management. In C++ we use pointers in conjunction with the *new* and *delete* operators to realise object lifecycle. The developer is responsible for cleaning up heap memory when it is no longer needed. The main action points that we address are:

- A1: Who (which object) creates the memory?
- A2: Who deletes the memory?
- A3: Who is the owner of the memory?
- A4: Can memory ownership be transferred to another object?
- A5: When is memory allocated/deallocated?
- A6: Can the heap become corrupted?

It is a major challenge to resolve these problems using raw C++ pointers (as has been experienced in many applications) and for this reason we decide to use the so-called smart pointers from the boost library (see www.boost.org and Karlsson 2006). As developer, you need to decide which specific smart pointer type to use in a particular context:

- Avoiding *dangling pointers* (pointers that do not point to a valid object)
- Make shared ownership of resources effective and safe
- Automatically deleting pointers when they go out of scope (or at the right time)

We now discuss memory allocation in the following subsections.

13.2.1 The Smart Pointer Types

Smart pointers solve the problem of managing the lifecycle of resources, and in particular dynamically allocated objects. There are five smart pointer types in the library `Boost.Smart_ptr`.

We discuss scoped and shared pointers in this book. In general, we use smart pointers for the following scenarios (Karlson 2006):

- *Shared ownership*: in this case two or more objects use a third, shared object. They may be able to modify the third object and the vital issue is to decide when to delete this third object. To this end, we use a smart pointer whose responsibility is to decide when and where a shared object may be deleted.
- *Exception safety*: in general, dynamically allocated objects using 'raw' C++ pointers are not deleted when an exception is thrown. What happens is that the stack is unwound, thus deleting the pointer but not the associated resources which remain inaccessible until the program terminates. Smart pointers resolve these problems, even when exceptions occur.

- *Avoiding common errors*: programmers are only human and they can forget to clean up memory after they have created it and when that memory is no longer needed. Smart pointers hide these deallocation details, thus removing the burden from the developer's shoulders.

We advise the use of smart pointers in projects, especially in large-scale applications. They are more robust and safer than using 'raw' C++ pointers or the STL `auto_ptr`.

13.2.2 Scoped Pointers

We use `scoped_ptr` to ensure that a dynamically allocated object is deleted when it goes out of scope; in other words, when we create an object in a function body it is then automatically deleted when it is popped from the stack. Here is an example:

```
#include "boost/scoped_ptr.hpp"
#include "Point.hpp"


int main()
{

     // Create dynamic memory
     boost::scoped_ptr <Point> myPoint (new Point(1.0, 23.3));


     // The memory is cleaned up automatically
     // NO delete needed

     // Check by placing print statement in the destructor
     // of the Point class

     return 0;
}
```

Next, having defined a scoped pointer, what can we do with it? First, a scoped pointer assumes ownership of the resource to which it points and this ownership will never be surrendered. In particular, you will get a compiler error if you assign one scoped pointer to another one. The use of scoped pointers does not affect performance and it improves the robustness and reliability of code. It overloads the major operators that raw pointers use and this makes it easy to understand, as the following (documented) code shows:

```
#include "boost/scoped_ptr.hpp"
#include "Point.hpp"


int main()
{

{
           // Create dynamic memory
           boost::scoped_ptr <Point> myPoint (new Point(1.0, 23.3));

           // Scoped pointer has same syntax as a raw pointer
           if (myPoint != 0)
```

```
          {
               cout << *myPoint;
          }

          // Assign to another point
          Point yourPoint (7.3, -9.9);

          *myPoint = yourPoint;
          cout << *myPoint;

          // Use operator '->'
          myPoint -> X(8.8);
          cout << *myPoint;

          // Cannot assign scoped pointers,operator '=' is private
          boost::scoped_ptr <Point> myPoint2 (new Point(1.0, 23.3));
          // THIS CODE DOES NOT COMPILE ' myPoint = myPoint2;

          // Illegal, cannot convert
          // boost::scoped_ptr <Point> illegalVar (new double);

     }

     return 0;
}
```

Finally, we discuss the *pimpl* idiom and its relationship with scoped pointers. This idiom was used in early C++ applications and it can still be found in legacy systems. Its main advantage is that it insulates clients from having to know about the private parts of a class. In particular, the constructor of the class allocates the pimpl type while the destructor deletes it, hence removing implementation dependencies from the header file. Here is an example in which we create a simple class that uses a struct. We forward declare the struct in the header file of the pimpl class:

```
#ifndef PIMPL_HPP
#define PIMPL_HPP

template <typename V>
     struct HestonStruct; // Forward reference


template <typename V>
     class Pimpl
{
private:
     HestonStruct<V>* data;
public:
     Pimpl();
     ~Pimpl();

     void initialiseData();
     void print() const;

};
```

```
#endif
```

The declaration of the struct containing the Heston data is:

```
#ifndef HestonStructure_HPP
#define HestonStructure_HPP

template <typename V> struct HestonStruct
{ // Defining properties of class, already discussed in chapter 5

     V T;
     V r, q;
     V kappa, theta, epsilon;
     V rho;                    // Correlation

     V IC_S;                   // Initial conditions S, V
     V IC_V;

     void initialiseData();    // Default data
     void print() const;
};
```

The code for the pimpl class is given by:

```
template <typename V>
     Pimpl<V>::Pimpl() : data(new HestonStruct<V>)
{

}

template <typename V>
     Pimpl<V>::~Pimpl()
{

     delete data;
}

template <typename V>
     void Pimpl<V>::initialiseData()
{

     data -> initialiseData();
}

template <typename V>
     void Pimpl<V>::print() const
{
     data->print();
}
```

The code looks robust enough. Unfortunately, it is not *exception safe*, which means in this case that if the constructor in the pimpl class throws an exception after the embedded pointer has

been created then the latter's destructor will not be called when the stack is unwound. In other words, we get a *memory leak* and for this reason we advise against the use of the current idiom. Instead, we use scoped pointers.

Scoped pointers should be used in the following situations:

- The lifetime of a dynamically allocated object is limited to a specific scope.
- A pointer is used in a scope where exceptions may be thrown.
- There are several control paths in a function.
- Exception safety is important (for example, when an exception occurs in a constructor of a class having pointer member data, then this pointer should be cleaned up).

13.2.3 Shared Pointers

In this case we are interested in the dynamic creation of objects that are subsequently shared by two or more other objects (this is an implementation of the GOF *Flyweight* pattern). The major problem is to know when to delete the original dynamically allocated object. If we delete it when another object needs it we will probably get a run-time error at some stage while if it is no longer needed we would like to remove it from memory. The solution is to use the *reference counting* principle; we define an integral reference count that is incremented (by one) when an object creates or accesses the shared object and it is decremented (by one) when the shared object is no longer needed. Only when the reference count is zero are we allowed to delete the shared object, and this is done automatically.

We discuss the shared pointer class in the boost library: it allows us to *non-intrusively* manage the lifecycle of a shared object among other objects. To this end, we have programmed some representative scenarios:

- Two classes that share common member data (built-in types or user-defined types).
- Determining the reference count of an object.
- Using shared pointers with STL containers.

In the first example we create two disjoint classes (that is, they have no common base class) called C1 and C2 that have shared member data:

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

class C2
{
private:
     //double* d; // OLD WAY
```

```
     boost::shared_ptr<double> d;
public:
     C2(boost::shared_ptr<double> value) : d(value) {}
     virtual ~C2() { cout << "\nC2 destructor\n"; }
     void print() const { cout << "Value " << *d; }
};
```

We now create instances of these classes; even though the first instance `object1` goes out of scope we see that the shared memory `commonValue` is still accessible to the instance `object2` of the second class:

```
     boost::shared_ptr<double> commonValue(new double (3.1415));
     {
           C1 object1(commonValue);

     }

     C2 object2(commonValue);
```

In the second example, we use the shared_ptr member function `use_count()` that tells us how many objects are accessing the shared memory at any given time. To this end, we create two classes D1 and D2 that share a common instance of class `Point`:

```
class D1
{
private:

     boost::shared_ptr<Point> p;
public:
     D1(boost::shared_ptr<Point> value) : p(value) {}
     virtual ~D1() { cout << "\nD1 destructor\n";}
     void print() const { cout << "\nValue " << *p; }
};

class D2
{
private:

     boost::shared_ptr<Point> p;
public:
     D2(boost::shared_ptr<Point> value) : p(value) {}
     virtual ~D2() { cout << "\nD2 destructor\n";}
     void print() const { cout << "\nValue " << *p; }
};

     boost::shared_ptr <Point> myPoint (new Point(1.0, 23.3));
     cout << "Reference count: " << myPoint.use_count() << endl;
     {
           D1 point1(myPoint);
           cout << "Reference count: " << myPoint.use_count() << endl;
           D1 point2(myPoint);
           cout << "Reference count: " << myPoint.use_count() << endl;

     }
     cout << "Reference count: " << myPoint.use_count() << endl;
```

```
    {
          D2 object3(myPoint);
          cout << "Reference count: " << myPoint.use_count() << endl;
    }
    cout << "Reference count: " << myPoint.use_count() << endl;
```

When we run this code, the following values for the reference count will be printed: 1, 2, 3, 1, 2, 1.

Finally, we show how to use shared pointers in STL containers. In the following example we create vectors whose elements are shared pointers. The base and derived class interfaces are defined as:

```
class Base
{ // Class with non-virtual destructor
private:

public:
     Base() { }
     virtual ~Base() { cout << "Base destructor\n\n"; }
     virtual void print() const = 0;
};

class Derived : public Base
{ // Derived class
private:

public:
     Derived() : Base() { }
     ~Derived() { cout << "Derived destructor\n"; }
     void print() const { cout << "derived object\n";}
};
```

We define a useful function that gives us a pointer to the base class (in real applications it would return the address of one of a number of derived classes):

```
// Simple creator function
boost::shared_ptr<Base> createBase()
{
     boost::shared_ptr<Base> result(new Derived());
     return result;
}
```

The data container is defined as:

```
     // Use in STL containers
     typedef std::vector<boost::shared_ptr<Base> > ContainerType;
     typedef ContainerType::iterator iterator;
```

We add elements to the container as follows:

```
     // Create a vector of objects
     ContainerType con;
     for (int j = 0; j < 10; ++j)
     {
          con.push_back(createBase());     // Add pointers to vector
     }
```

Finally, we print the elements of the container using an iterator:

```
// Now iterate and print
iterator myIter;
for (myIter = con.begin(); myIter != con.end(); ++myIter)
{
     (*myIter) -> print();
}
```

This example gave the essentials of using shared pointers with STL containers. Shared pointers are useful in the following situations:

- When we implement the *Flyweight* design pattern (GOF 1995); we create objects on the heap and they can be shared by other objects.
- Storing pointers in STL containers.
- When there are multiple clients of an object, but no explicit owner.
- When passing objects to and from libraries without any clear ownership regime.
- When managing resources that need special cleanup procedures, for example using *custom deleters*. A customer deleter is one that augments the simple C++ *delete* keyword. For example, resource handles are good candidates for customer deleters. This feature could be implemented in conjunction with the *Builder* pattern.
- As a replacement for the pimpl idiom.

13.2.4 Using Smart Pointers in Monte Carlo Applications

We now step back from the details of smart pointers and we think about where they can be used in Monte Carlo applications. In this book we used raw pointers in the main, readability and understandability reasons and also because smart pointers are - strictly speaking - not (yet) part of the C++ standard. In future applications we shall use them and our advice to readers is to apply them whenever possible because they improve code reliability.

Since we use a number in the following situations of high-level structural patterns (such as *Whole-Part* and *PAC*) to model the classes and objects in Monte Carlo applications it is relatively easy to define object creational policies and object lifecycles. In general, scoped and shared pointers are to be recommended:

- If we need to create an object whose lifetime is limited (for example, a factory object), then we can use scoped pointers. Factories are typically objects whose scope is a single subsystem and do not have to be known outside that subsystem.
- If we need to create an object that will be shared among several objects and subsystems, then we use shared pointers.
- We could consider the use of the *Builder* pattern that creates objects on behalf of the whole role object in the *Whole-Part pattern*. The builder then creates the parts. The advantage of this approach is that all object creation and destruction is localised in one place, namely the builder object itself.
- If shared objects are not needed in an application, then the use of shared pointers is probably superfluous. In that case, the use of raw C++ pointers (don't forget delete) should be recommended.

13.3 Modelling Homogeneous and Heterogeneous Data in Boost

In chapter 12 we discussed a number of ways of modeling data in C++. But boost has some of the same capabilities. We note that the boost libraries will become part of the standard C++ language in due time.

When determining which library to use when modeling data we need to ask some questions:

- Is data homogeneous or heterogeneous?
- Are data collections of fixed or of variable size?
- Is the data defined at compile-time or can be it changed at run-time?

13.3.1 Tuples

The *Boost Tuple* library can be seen as a generalisation of `std::pair` (this is a struct that groups pairs of objects.) It is now possible to define so-called *n-tuples*; these structures model fixed-sized collections of values of specified types (for example, STL pairs are considered to be 2-tuples). Tuples correspond to *record structures* in languages such as Cobol and we need them in many kinds of applications, for example:

- When we wish to have multiple return types from functions.
- When we need tuples input arguments to functions.
- When we group logically related types (for example, defining records and tables).

Some languages have built-in support for n-tuples, but not C++. In order to fill the gap, the boost library developers created the `Boost.Tuple`.

We now take some examples to show how to create and use tuples. The first example creates 2-tuples and populates them using a constructor and the convenience function `make_tuple()`:

```
#include "boost/tuple/tuple.hpp"
#include <string>
#include <iostream>
using namespace std;

int main()
{
     // Using declaration, for readability purposes
     using boost::tuple;

     // Creating tuples
     tuple<string, double> myTuple(string("Hello"), 3.1415);
     tuple<string, int>
     myTuple2 = boost::make_tuple(string("position x"), 0);

     return 0;
}
```

We now show how to retrieve the elements of a 3-tuple using the template member functions `get<>()` that accepts an index value that must be less than the number of elements in the tuple:

```
    // Retrieving values from a tuple
    tuple<long, double, string>
              newTuple(100, 2.17, string("a new tuple"));

    long first = newTuple.get<0>();
    double second = newTuple.get<1>();
    string third = newTuple.get<2>();

    cout << "Elements of tuple: " << first << ", " << second
                << ", " << third << endl;
```

Summarising, we use tuples when modelling fixed sized collections whose elements can be of heterogeneous data type; the data types can be built-in or user-defined types.

13.3.2 Variants and discriminated Unions

The *Boost.Variant* library is used for typesafe storage and retrieval of a bounded set of types, that is *discriminated unions*. The main uses are:

- Typesafe storage and retrieval of user-specified sets of types.
- Storing heterogeneous types in STL containers.
- Compile-time checked visitation of variants (using *Visitor* pattern).
- Efficient, stack-based storage for variants.

We describe the functionality in this library and we refer the reader to Karlsson 2006 for a more detailed description, in particular the application of the *Visitor* pattern. We take an example of a variant with three different types in the discriminated union:

```
#include "boost/variant.hpp"
int main()
{
    // Using declaration, for readability purposes
    using boost::variant;

    variant<long, string, Point> myVariant;

    myVariant = 24;
    print(myVariant);

    myVariant = string("It's amazing");
    print(myVariant);
    myVariant = Point();
    print(myVariant);
    return 0;
}
```

The function for printing the current value in the variant is given by:

```
template <typename V> void print(V& v)
{
    if (long* pi = boost::get<long> (&v))
    {
```

```
          cout << "It's a long " << *pi;
     }
     else
     if (string* st = boost::get<string> (&v))
     {
          cout << "It's a long " << *st;
     }
     else
     if (Point* pt = boost::get<Point> (&v))
     {
          cout << "It's a point " << *pt;
     }

     cout << endl;
}
```

The *Variant* library complements the *Any* library, which we now describe.

13.3.3 Any and undiscriminated Types

The *Boost.Any* library resolves a number of the shortcomings of C++, in particular when we create collections whose elements are of any type. In the past, we modelled these collections with `void*` but this lacks type safety. The *Any* library resolves this problem by ensuring that there is no way to get a value without knowing its type; in this way we preserve type safety. Some of the advantages and uses of this library are:

- Storing heterogeneous types in STL containers.
- Typesafe storage and safe retrieval of arbitrary types.
- In layered applications, objects send and receive data without having to know anything about the precise types of the data that they send and receive.
- We can use the `any` class as a kind of variant data type.

The `any` class preserves the type of the data that it stores and you cannot get at the stored value unless you know the correct type. It is possible to query for the type using run-time type information (RTTI) and casting functions in C++. We use the `any` class in cases when we do not wish to know about the type. At some stage we determine what the actual type is, but this can be done in a specialised function.

We take an example of a list of `any` types:

```
#include "boost/any.hpp"
#include <string>
#include <list>
#include <iostream>
using namespace std;

int main()
{
     // Using declaration, for readability purposes
     using boost::any;
     list<any> myList;
```

```
     long v = 123;
     double d = 3.14;
     string s = string("D");
     Point p;

     // Populate the list with all kinds of types
     myList.push_back(v);
     myList.push_back(d);
     myList.push_back(s);
     myList.push_back(p);
     myList.push_back(make_pair<bool, double>(true, 3.14));

     return 0;
}
```

The list also contained a variable P that is a Point:

```
struct Point
{
     double x, y;

     Point() { x = y = 0.0;}

     friend ostream& operator << (ostream& os, const Point& pt)
     {
          os << "(" << pt.x << "," << pt.y << ")" << endl;
          return os;
     }
};
```

13.3.4 A Property Class

A useful application of the *Any* library is to define a class that represents name-value pairs. We call this the *Property class* and we have used it in the past as well as in the current book to model class' attributes (see Duffy 2004 and Duffy 2006). We have created our own *Any* data type and we use it in the applications in Part III of this book but in general we recommend using the boost version. To this end, we have a simple Property class as follows:

```
using boost::any;
template <typename Key>
          class Property
{
private:
          Key nam;
          any val;
public:
     Property(const Key& key, const any& value)
                                : nam(key), val(value)
     {
     }

     string Name() const { return nam;}
     any& Value() { return val; }
```

```
     // Allows storage in STL containers
     friend bool operator < (const Property<Key>&lhs,
                             const Property<Key>&rhs)
     {
          return lhs.nam < rhs.nam;
     }
};
```

We now create a number of instances of this class and we add them to a vector. In this way we can create simple *property maps* (which we call *property sets* in Duffy 2006):

```
int main()
{
     // Using declaration, for readability purposes
     using boost::any;

     long v = 123;
     double d = 3.14;
     string s = string("D");

     Property<string> myProp(string("1"),v);
     Property<string> myProp2(string("2"),d);
     Property<string> myProp3(string("3"),s);

     vector<Property<string> > properties;
     properties.push_back(myProp);
     properties.push_back(myProp2);
     properties.push_back(myProp3);

     return 0;
}
```

13.3.5 Boost Arrays

The array class in the *Boost.Array* library is close in spirit to the authors' `VectorSpace<V, N>` class. Instead of providing the semantics of dynamic arrays, we note in this case that the size parameter of a Boost array is fixed at initialization time and this parameter is a template parameter. The interface of this class has most of the member functions that you would expect:

- No constructors, only the assignment statement.
- Indexing operator [].
- Iterators.
- Functions to retrieve the first and last elements of the array.
- It is possible to compare arrays (lexicographical compare).

We take an example to show how to use the class:

```
#include "boost/array.hpp"
#include "Point.hpp"
```

```
int main()
{

     boost::array<long, 4> myArr = { {1, 2, 3, 4}};


     // Using iterators
     boost::array<long, 4>::const_iterator it;

     for (it = myArr.begin(); it != myArr.end(); ++it)
     {
          cout << *it << endl;
     }

     // Indexing operator
     for (long i = 0; i < myArr.size(); ++i)
     {
          cout << myArr[i] << endl;
     }

     boost::array<Point, 4> myPointArr;
     myPointArr[0] = Point();
     myPointArr[1] = Point();
     myPointArr[2] = Point();
     myPointArr[3] = Point();

     return 0;
}
```

We now review some other libraries in Boost. The objective is to help the reader become aware of the many useful resources that it has to offer.

13.4 Boost Signals: Notification and Data Synchronisation in the Application

The *Signals* library is the Boost implementation of the *Observer* or *Publisher-Subscriber* pattern (GOF 1995, POSA 1996). It allows developers to manage events while ensuring that inter-object dependencies are kept to a minimum, a problem that plagues the more traditional object-oriented approach taken in the *Observer* pattern. Using the *Signals* library ensures that the emissions of signals or events (by *subjects*) are decoupled from the slots (also known as *observers* or *subscribers*). The *Signals* library is similar to the *Delegates mechanism* in the Microsoft .NET framework, but the former is more flexible.

Some of the features of *Signals* are:

- Flexible multicast callbacks for functions and function objects.
- A robust method for triggering and handling events.
- Compatible with function object factories (for example *Boost.Bind* and *Boost.Lambda*).
- In general, implementing *callbacks*.

We take a simple example to show the Signals library works. The main steps are:

1. Create a signal.
2. Define functions or function objects (these are slots).
3. Connect the slot to the signal.
4. Emit the signal (and the slots are called immediately).
5. Optionally, disconnect a slot.
6. Emit the signal, again.

The corresponding C++ code is given by:

```
#include "boost/signals.hpp"
#include <iostream>
using namespace std;

// Define a normal function (a Slot)
void mySlot()
{
     cout << "a slot \n";
}

// Define a function object (a Slot)
struct SecondSlot
{
     void operator () () { cout << "a second slot \n";}
};

int main()
{

     // 1. Create signal
     boost::signal<void ()> signal;

     // 3. Connect slots
     signal.connect(&mySlot);
     signal.connect(SecondSlot());

     // 4. Emit the signal
     signal();

     // 5. Disconnect first slot
     signal.disconnect(&mySlot);

     // 6. Emit signal (one slot less)
     signal();


     return 0;
}
```

A good example of where the *Boost.Signals* library can be used is when we separate GUI code from business logic code. In general, it supersedes old-style callbacks. Finally, it is possible to define so-called *Combiners* that allow developers to write event mechanisms that are tailor-made for a given domain. For more on this, see Karlsson 2006.

13.5 Input and Output

We discuss two libraries in this section. The first one allows the developer to write data to, and retrieve data from a disk while the second library supports portable file systems.

13.5.1 Boost.Serialisation

This portable library allows us to save C++ data structures to archives and to restore these structures from archives. In this case we are referring to *persistent data*, that is objects whose lifetime survives the scope in which they are defined. An archive can be a text file or an XML file, for example. The library offers support for class versioning, for STL classes and shared data, for example. Another way to define serialization is to say that it is the reversible deconstruction of an arbitrary set of C++ data structures to a sequence of bytes. We can then reconstitute an equivalent structure in another program context.

We take an example. In this case we model the parameters of the Heston model as a struct and we apply the archiving functionality to save the data to disk and then we restore the data from disk. First, we define the necessary include files:

```
#include <fstream>
#include <iostream>

// Include headers that implement an archive in simple text format
#include "boost/archive/text_oarchive.hpp"
#include "boost/archive/text_iarchive.hpp"
```

We define the persistent data structure that we introduced in chapter 5:

```
template <typename V> struct HestonStruct
{ // Defining properties of class, already discussed in chapter 5

     V T;
     V r, q;
     V kappa, theta, epsilon;
     V rho;              // Correlation

     V IC_S;             // Initial conditions S, V
     V IC_V;

     // Template member function
     template <typename Archive>
          void serialize(Archive& arc, unsigned int version)
          { // Classes must implement this function

                  arc & T; arc & r; arc & q;
                  arc & kappa; arc & theta; arc & epsilon;
                  arc & rho; arc & IC_S; arc & IC_V;
          }

};
```

We note that each persistent class must implement the `serialize()` function, as above. Finally, the code for two-way input-output is given by:

```
int main()
```

```
{
     ///////////////// Data Area ////////////////////////////
     HestonStruct<double> data;

     data.T = 1.0; data.r = 0.1; data.q = 0.0;

     data.kappa = 1.98837; data.theta = 0.1089; data.epsilon = 0.15;

     data.rho =  -0.9;

     data.IC_S = 123.4; // ATM
     data.IC_V = 0.37603 * 0.37603;


     // Create and open a character archive for output
     std::ofstream ofs("Heston.dat");

     //Save data to archive
     {
           boost::archive::text_oarchive oa(ofs);

          // Change some data
          data.r = 0.08;

          // Write class instance to archive
          oa << data;

     }

     // ... Some time later, restore the class
     HestonStruct<double> newData;
     {
          // Create and open an archive for input
          std::ifstream ifs("Heston.dat", std::ios::binary);
          boost::archive::text_iarchive ia(ifs);

          // Read class state from archive
          ia >> newData;

     // Just checking
     std::cout << "Interest rate is: "
                     << newData.r << std::endl;

     }

     return 0;
}
```

For completeness, we show the contents of the archived file:

```
22 serialization::archive 4 0 0 1 0.080000000000000002 0 1.98837 0.1089
0.14999999999999999 -0.90000000000000002 123.40000000000001 0.14139856089999997
```

The *Serialisation* library allows us to build a simple and effective data storage system.

13.5.2 Boost.Filesystem

This is a portable library for the manipulation of paths, directories and files. The developer can write code using script-like operations. For example, algorithms are provided for iterating in directories and files. Here is an example of use (more information on www.boost.org):

```
#include "boost/filesystem.hpp"
#include <iostream>


int main()
{

     boost::filesystem::path my_path( "Heston.dat" );

     return 0;
}
```

Summarising, this library allows the developer to perform portable filesystem operations (such as searching in directories and their contents, for example). Its goal is to provide scripting language facilities when C++ is the language of choice, not to replace established scripting languages such as Phyton and Perl.

13.6 Linear Algebra and uBLAS

The uBLAS library consists of a number of linear algebra operations for vectors and matrices using mathematical notation based on operator overloading. The matrices can be dense, packed or sparse. It has facilities for efficient code generation using so-called *expression templates* (see Abrahams 2005 for a discussion of this topic in particular and C++ *template metaprogramming* in general).

The library contains much of the functionality that has already been created by one of the current authors (see Duffy 2004). Furthermore, the uBLAS library supports different kinds of *banded matrices*, for example:

- Dense matrices.
- Triangular matrices.
- Symmetric matrices.
- Hermitian matrices.
- Banded matrices.
- Sparse matrices

It is also possible to define how the data is stored in memory, for example *row-major order* or *column-major order*. The choice has an impact on performance as we shall see in chapter 25. C++ uses a row-margin order regime.

Here is an example to show how to use the library by creating a dense matrix:

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
```

```
int main()
{
     using namespace boost::numeric::ublas;
     matrix<double> m (3, 3);
     for (unsigned i = 0; i < m.size1(); ++i)
     {
          for (unsigned j = 0; j < m.size2(); ++j)
          {
               m(i,j) = 3 * i + j;
          }
     }
     std::cout << m << std::endl;
     return 0;
}
```

In future applications we will use these patterned matrices as a basis for advanced numerical applications.

For example, we could use the GOF *Adapter* pattern to create specialised classes that used a variety of matrix classes, for example linear and nonlinear solvers in numerical analysis and its applications.

13.7 Date and Time

The *Date_time* library supports date and time types and the corresponding operations on them. Programming with dates and times becomes almost as simple as programming with strings and integers. The following classes are supported:

- *Date*: the primary interface for date programming.
- *Date Duration*: simple day count used for arithmetic with *Date*.
- *Date Period*: direct representation for ranges between two dates.
- *Date Iterators*: a standard mechanism for iterating through dates.

Furthermore, the library has algorithms for generating dates and schedules of dates. For example, we can represent concepts such as "the first Sunday in February", "the first Sunday after Jan 1, 2009" and so on.

The following simple example shows how to use the functionality. In this case we create a date based on a string:

```
// TestDate.cpp
//
// Testing boost Date_time (large!) library
//
// NB in version 1.34.1 you need to include the
// corresponding LIB for Date_time.
//
// (C)Datasim Education BV 2008
//

#include "boost/date_time/gregorian/gregorian.hpp"
#include <string>
```

```
using namespace std;

int main()
{

     using namespace boost::gregorian;

     // ISO 8601 extended format CCYY-MM-DD
     string s("2009-10-9"); // 10 October 2009
     date myDate(from_simple_string(s));

     // Now convert to a string
     string converted = to_simple_string(myDate);
     cout << "String: " << converted << endl; // OUTPUT is 2009-Oct-09

     return 0;
}
```

In future applications the authors intend to use boost's date and time classes and we will incrementally migrate our C++ code from the date and time classes they we created a number of years ago.

13.8 Other Libraries

We conclude this chapter with a short description of some other libraries in Boost. For more information, please consult www.boost.org and the CD accompanying the current book where we have provided C++ code examples to help the reader understand the essentials of these libraries.

13.8.1 String and Text Processing

There are four major libraries in boost for analyzing and parsing text and strings:

- *Regex*: a library for pattern-matching applications, in particular support for regular expressions in C++ and improving the robustness of input validation.
- *Spirit*: a recursive-descent parser generator framework. It makes use of meta-programming techniques and expression templates that approximate the syntax of *Extended Backus Naur Form* (EBNF). It is possible to create grammar rules in C++ and the library can be used to define command-line parsers, for example.
- *Tokenizer*: this library offers functionality for separating character sequences into tokens, for example finding data in delimited text streams. The user determines how the character sequence is delimited and the library finds the tokens as the user requests new elements.

An example of use is:

```
#include<iostream>
#include<boost/tokenizer.hpp>
#include<string>

int main()
{
     using namespace std;
```

```
    using namespace boost;


    // Default delimiter, white space
    string s = "This is,  a test";
    tokenizer<> tok(s);
    for(tokenizer<>::iterator beg=tok.begin(); beg!=tok.end();++beg)
    {
        cout << *beg << "\n";
    }

    // Delimiter is '\'
    string s2 =
    "Fld 1,\"putting quotes around fields, allows commas\",Fld 3";

    tokenizer<escaped_list_separator<char> > tok2(s2);

    for(tokenizer<escaped_list_separator<char> >::iterator
                beg=tok2.begin(); beg!=tok2.end();++beg)
    {
        cout << *beg << "\n";
    }
}
```

This is a useful library when you wish to define and use delimited ASCII files, for example.

13.8.2 Function Objects and Higher-Order Programming

We describe three libraries that allow developers to bind functions to function pointers and function objects, implement callback functions and provide support for *lambda expressions* (*unnamed functions*).

- *Bind*: this library binds arguments to anything that behaves like a function (for example, a function pointer, function object or a member function pointer) and it is an improvement on the STL binders `bind1st` and `bind2nd`. The advantage is that syntax is now uniform. This library is useful for *functional composition*. This is a mechanism to combine simple functions to build more complicated ones. Like the composition of functions in mathematics, the result of the composed function is passed to the composing one via a parameter. Because of this similarity, the syntax in program code tends to closely follow that in mathematics.
- *Function*: this is a library that implements generalized callback mechanisms. It allows us to store and invoke function objects, function pointers and member function pointers. This library works with *Boost.Bind* and *Boost.Lambda*.
- *Lambda*: a library that supports unnamed functions, especially in combination with STL algorithms. The term originates from *functional programming* and *lambda calculus*, where a lambda abstraction defines an unnamed function. The primary motivation is to provide a flexible and convenient way to define unnamed function objects for STL algorithms.

13.8.3 Concurrent and Multi-threading Programming

The *Thread* library allows us to write programs as multiple, asynchronous, independent threads-of-execution. Each thread has its own machine state including program instruction counter and

registers. We discuss parallel programming in more detail in chapters 24 and 25.

13.8.4 Interval Arithmetic

This library implements mathematical intervals that are used in numerical analysis applications. This is especially important when computations produce inexact results or when input data is imprecise. Furthermore, intervals allow us to quantify the propagation of rounding errors.

An interval is a pair of numbers and these numbers represent the infinity of all real numbers between them. We consider an interval to be closed, hence it includes its end-points. The fundamental property of interval arithmetic is the *inclusion principle*: if f is a function on a set of numbers then it can be extended to a new function defined on intervals. This new function has an interval $[a, b]$ as argument and it returns an interval, as follows:

$$\forall x \epsilon [a, b], \quad f(x) \in f([a, b])$$

The library provides arithmetic operators on intervals, such as (, -, *, /+), algebraic and piecewise-algebraic functions (such as `abs, sqrt`), transcendental and trigonometric functions (`sin, tanh, acosh` and so on) as well as the standard comparison operators such as $<$, $==$ and $>$. In this sense it is similar to the way fuzzy numbers are defined. It is possible to customise usage of the library by defining your own policies.

13.8.5 Mathematics Toolkit

This boost library contains a wealth of functionality for three interconnected mathematical functions, namely 1) univariate continuous and discrete statistical distributions (we discussed these in chapter 3), 2) special functions such as the gamma, incomplete gamma, beta and Bessel functions and 3) other functions for the calculation of infinite series, continued fractions, root finding, function minimisation and the manipulation of polynomials, to name just a few. The functions in this library can be used in their original form and they can be adapted to create higher-level functions for use in finance applications such as the Monte Carlo method.

13.8.6 Accumulators and Time-Series

Boost.Accumulators is a library for incremental statistical computation as well as a framework for incremental calculation in general. The basic concept is the *accumulator* which is a primitive computational entity that accepts data, one sample at a time and maintains some internal state. Some of the functionality includes:

- Weighted averages for the mean, covariance, skewness, moments and other statistical quantities.
- Extracting results from accumulators.
- Numerics operators sub-library.

This library could be useful when creating applications involving time series.

13.8.7 Shared Memory and Distributed Memory Processing

Boost has support for light-weight processes (threads) that execute in a shard-memory computer and processes that execute in a distributed memory configurations (the Message Passing Interface (MPI)) . Thread functionality is realised by the *Thread* library (for an introduction to threads,

see chapter 24 of the current book). In particular, the library has two major classes, namely for a thread and a collection of related threads. The thread class has functionality for thread creation and launching, joining and detaching a thread, defining synchronisation constructs (such as mutex, locks and lock functions, barriers and condition variables) and for the defining of interruption points. This is quite low-level behaviour and its proper application may be a 'bridge too far' for novice developers. An alternative is to use the (de-facto standard) OpenMP library that hides much of the detail of thread lifecycle management. We introduce OpenMP in chapter 25. *Boost.MPI*, on the other hand, is a library for message passing for high-performance parallel applications (Gropp 1997). A typical program consists of one or more processes that communicate by sending and receiving messages using *point-to-point communication* or by coordinating as a group (*collective communication*). Boost.MPI processes can be spread across many different machines, possibly having different operating systems and underlying architectures.

13.9 Summary and Conclusions

We have given an overview of the Boost library that extends and improves STL and brings C++ to a higher level. In particular, we gave a short overview of some of its libraries and some '101' examples to show how to use them. The libraries are very well written and if you know C++ templates and STL syntax you should not have much difficulty understanding and applying them.

We recommend that you look in Boost first to see if it has functionality for your applications.

This chapter is meant as a short introduction to the Boost library and we give an overview of some of it major functionality and we give some generic examples and some examples that are relevant to the Monte-Carlo method. In order to show how it can be used in an application, we have designed and implemented a Monte Carlo engine for the Heston model using as much of the relevant functionality of boost. In particular, we use smart pointers, boost data structures, function objects and multi-threading libraries to emulate the results from chapter 5 and those from Part III of this book. The source code and description of the problem can be found on the CD. Reading the code and running it is a good way to learn how boost works.

13.10 Exercises and Projects

1. (Using Arrays of Tuples, **)
   We create an array structure whole elements represent average promised yields on corporate bonds by quality rating; in particular, for each year (that we model by a `DatasimDate` class) we give the ratings such as Aaa, Aa, A and Baa. A typical record (row) is given as:

Year	Aaa	Aa	A	Baa
1976-1-1	8.43	8.75	9.09	9.75

   Answer the following questions:

   a. Design the data structure for this problem using boost tuples and STL vectors.

   b. Populate the data structure using a combination of tuple constructors, the convenience function `make_tuple()` and the vector indexing operator [].

   c. Create a function to print each record in the data structure.

2. The (*Any* library, **)
   Complete the code in section 13.3.3 by printing all the values of the types in the list of `any` types (use casting and exception handling).

3. (Which data structure to use, ***)
   In section 13.3 we discussed a number of attention points when choosing a suitable data structure in applications:

   - A1: Is data homogeneous (all of the same type) or heterogeneous?
   - A2: Are data collections of fixed or variable size?
   - A3: Is the data defined at compile-time or can be it changed at run-time?

   Determine in how far the following libraries satisfy the above three attention points: *tuple, any, variant, array.*

4. (Reegineering Finite Difference Engine of chapter 10, ****)
   The objective of this exercise is to upgrade the code and design that we introduced in section 10.2. In this case we 1) apply more flexible design than the basic one we used in section 10.2) we use a number of the specific Boost libraries to improve the maintainability, performance and functionality of the resulting software system. Which libraries would you use in this particular case?

5. (End-Of-Part II Project, *****)
   This exercise represents a medium-sized project to test your knowledge of the first 13 chapters of this book. We base the exercise on the *initial* UML diagram in Figure 13.1 and we extend it based on the requirements and desired features that we presently describe:

   - `MCEngine`: The class that offers pricing services to clients.
   - `SDE`: C++ class hierarchy that models stochastic differential equations.
   - `FDM`: C++ class hierarchy for finite difference schemes.
   - `NormalGenerator`: classes for generating normal random variates.
   - `UniformGenerator`: classes for generating uniform random variates.
   - `Builder`: Creates the objects in Figure 13.1.
   - `Client`: This is your main90 program.

   We examine this problem from four perspectives in the software lifecycle, namely 1) defining the scope of the problem, 2) object-oriented design of the problem, 3) C++ implementation and 4) evolution of the software product.

   Problem Scope We describe the kinds of derivatives products that we support:

   - R1: 1, 2 and n factor models.
   - R2: Path-dependency (for example, Asian options).
   - R3: Calculating Greeks using finite differences.
   - R4: Various random number generators.
   - R5: Different kinds of payoff functions.

   Design We discover and document the classes that realise requirements R1 to R5:

   - R6: System configuration using *Builder* and other *factory* patterns.
   - R7: Modelling SDEs as a template `SDE<V, N, D>` class.
   - R8: Using *Visitor, Strategy* and *Template Method* pattern to implement FDM.

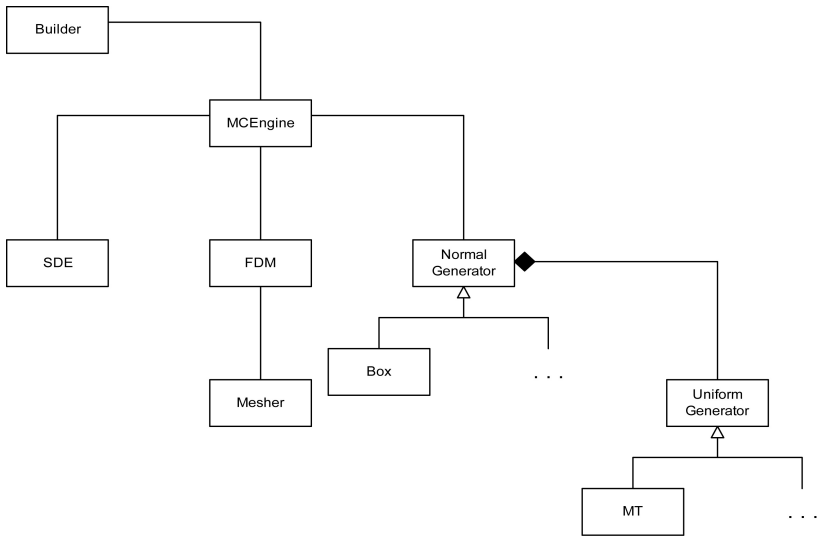

Figure 13.1: End-of-term project

- R9: Choosing between OO and policy-based design approaches.
- R10: Displaying results in Excel.

C++ Implementation We decide which particular software tools to use:

- R11: boost shared (and scoped) pointers.
- R12: `vector<T>` , `Vector<V, I>` and `VectorSpace<V,N>` classes.
- R13: Using *Property* pattern to model payoff parameters.
- R14: Multi-threading (for example, OpenMP or Boost.Thread).

Software Evolution We deliver a running system as a series of prototypes:

- R15: One-factor plain vanilla option.
- R16: Asian option.
- R17: Two-factor spread option with stochastic volatility.
- R18: Heston model.

6. Use the classes in the uBlas library to create Cholesky decomposition code.