The Boost C++ Libraries Overview and Applicability to Computational Finance: Part I

Daniel J. Duffy
Datasim Education BV, e-mail: dduffy@datasim.nl

1 Introduction and Objectives

In this article we give an overview of the Boost C++ libraries (at the moment of writing there are approximately 100 specific libraries in the Boost suite) and what they have to offer. In particular, we focus our attention on the functionality that we think is of value in computational finance applications. This article functions as a roadmap and can be used as an aid to help the reader first of all determine if Boost has implemented the functionality that she is interested in and second how to learn and apply the appropriate library once it has been found.

One of the criticisms of C++ in the past is that is has little support for extended functionality in the form of libraries. It was often necessary for developers to create their own dedicated data structures and related functionality as necessary precursor to writing applications (for example, Duffy 2004). With the emergence of Boost we are now able to concentrate on the application while much of the lower-level *plumbing* can be delegated to one or more Boost libraries.

This article is a general overview of the Boost C++ libraries and it is meant to create awareness of the library.

2 History of Boost

Boost is a suite of free peer-reviewed portable C++ source libraries. The goal of these libraries is that they be useful, compatible with STL, and used in a wide range of applications. These libraries increase programmer productivity and mitigate reinvention of the software wheel. Long-term maintenance costs are drastically reduced because Boost is developed and supported by teams of professional developers. Finally, Boost runs on most operating systems, including UNIX and Windows.

Boost has its own license and it encourages both commercial and non-commercial use. The authors of Boost see the license as being more business-friendly than other license agreements. For more information on this and other technical issues (e.g., software downloads and installation) related to Boost, we refer the reader to www.boost.org.

3 Library Classification

There are approximately 100 Boost libraries. Each library realises some concern in software development. These concerns are related to a number of software categories, as shown in Figures 1 and 2. In general, a given library addresses one or more concerns and hence can be classified in one or more categories. For convenience, we place each library in one category. We further partition the categories into context-dependent groups that are (in our opinion) useful in computational finance (the libraries in Figure 1) and those libraries that are nonetheless important but which developers may not use on a regular basis (the libraries in Figure 2).

The next subsections provide a high-level overview of each library in Figures 1 and 2 and describe each library in such a way that the reader can determine if the library is useful for the job at hand. Having determined that the library is useful, the developer can then examine prototypical examples of use and discussions on the general applicability of the library. For more details on how to use these libraries, see Demming 2010, Demming 2011, and the online documentation on www.boost.org.

The list of libraries in Figures 1 and 2 does not includes all the libraries in Boost. The Boost website (www.boost.org) provides a complete and up-to-date list.

3.1 Essential Libraries

We now refer to Figure 1. The *Math category* contains libraries for a number of topics in applied and numerical mathematics, including:

- Special Functions, for example Bessel, gamma, factorial and error functions;
- Orthogonal polynomials;
- Approximately 30 univariate statistical distributions;
- Matrix library for a range of matrix and vector structures and associated operations.
- Random number generators;
- Support for Interval Analysis and interval arithmetic;
- Classes to represent integers and rational numbers;
- Functions to compute greatest common divisor and least common multiple of two numbers;
- Non-linear solvers (for example, the Newton–Raphson method).

The libraries in the *Data Structures category* extend the range of capabilities in C++ and they help us to directly use advanced data structures in C++

Figure 1: Essential libraries.

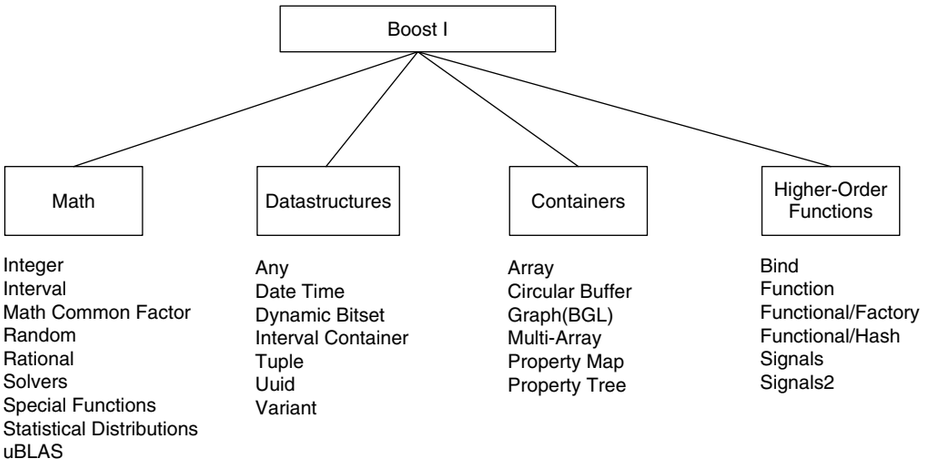
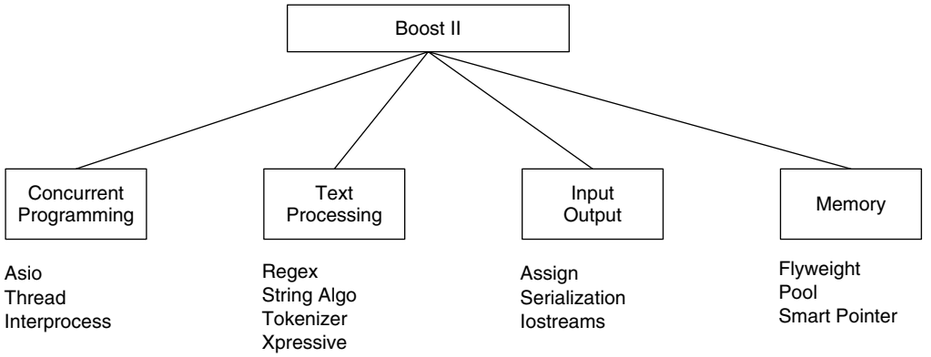


Figure 2: Supporting libraries.



code. Some of these data structures are fundamental improvements on and replacements for existing structures in C and C++ while others are more advanced and have many applications:

- Any: this class models heterogeneous data; it is a type-safe generalisation of `void*` in C.
- Date and Time: contains a set of generic modules that support a wide range of operations on date and time classes. This library is very useful in applications that need dates and times.
- Dynamic BitSet: this class represents a set of bits. The number of bits in instances of this class is dynamic and can be set in the constructor. This is in contrast to `std::bitset<N>` in which the number of bits is fixed at compile-time.
- Interval Container: this library has functionality for defining interval sets and maps. It has many applications to scheduling problems.
- Tuple: this class groups a fixed number of objects of different types into one logical whole. It is a sequence of values and it can be seen as a generalisation of `std::pair`.
- Uuid: this is a class that models a universally unique identifier and it can be used in applications in which we wish to assign unique 'tag' numbers to certain objects.
- Variant: a type-safe class that can hold objects of different types and sizes at different times. It is similar to the (unsafe) C *union* but is type-safe.

These data structures are of general applicability and their use promotes code robustness and readability.

The libraries in the *Containers* categories are suitable for advanced mathematical applications in which structured data needs to be processed. In the past developers tended to create these data structures themselves or used proprietary libraries:

- Array: this is an STL-compliant container for fixed-sized (that is, at compile time) vectors.
- Circular buffer: an STL-compliant container that never overflows. It is typically used in producer-consumer applications.

- Graph (BGL): this library contains a massive amount of functionality for graphs, graph operations and graph algorithms. It has many applications.
- Multi-Array: this library offers functionality for modelling N-dimensional arrays. It can be used in applications in which the concept of n-dimensional geometry plays a role.
- Property Map: Concepts that define interfaces that map key objects to value objects. This library plays an important and supporting role when developing BGL applications.
- Property Tree: this is a tree data structure that is suitable for storing configuration data.

There are many applications in computational finance of the libraries in the *Containers* category that we shall discuss later.

Finally, the libraries in the *Higher-Order Functions* category represent major improvements and extensions to how functions are defined and used in C++ and STL. Furthermore, their use allows us to apply modern design methods to create flexible and extendible software systems:

- Bind: this is a generalisation of the function bind in STL. It supports arbitrary function objects, function pointers and member function pointers. It also supports function composition and the binding of function arguments.
- Function: this very important library implements function object wrappers for deferred calls or callbacks.
- Functional/Factory: this library contains function object templates for dynamic and static object creation. It is the Boost equivalent of the Gamma (GOF) Factory Method pattern. One of the disadvantages of OOP patterns is the amount of boilerplate code that needs to be written. Using Functional/Factory we can achieve the same ends with less code and without having to define extra classes.
- Functional/Hash: a hash function that can be extended to hash user-defined types.
- Signals: this is a library that implements event-notification patterns, such as the GOF *Observer*, *Mediator* and *Chain of Responsibility*. The difference however, is that Signals is based on a delegates mechanism (as with C#) rather than using inheritance as is the case with GOF.
- Signals2: This is the thread-safe version of Signals and it implements signals and slot callback mechanisms.

There are many ways to use these libraries in applications, both on a standalone basis and in combination with each other and with libraries for data types and containers.

3.2 Supporting Libraries

The libraries that we discussed in the previous section are of direct relevance to quant developers in our opinion, in particular libraries for mathematical and higher-order functions are useful. Most of the productivity gains will be realised by using these libraries.

In this section we discuss a number of libraries which certainly do have applications but may be needed on an incidental basis only. We refer to Figure 2 and we classify the libraries into four main categories for concurrent and network programming, text and string processing, input-output and memory management.

The libraries that deal with multi-threading and network programming allow developers to create efficient and scalable software systems. The most important libraries are:

- Asio: a portable library for network programming (including sockets) with support for TCP/UDP protocols, IP addressing and name resolution.
- Thread: a library that enables the creation of efficient applications on multi-processor shared memory computers. This library allows the creation of *lightweight processes* or *threads*.
- Interprocess: a library that supports shared memory, memory mapped files, process-shared mutexes, condition variables, containers and allocators. This library allows communication between *heavyweight processes*.

In some cases developers may need to create applications that process text and string data in some way. There are many situations in which text needs to created, processed and exported to different formats. The libraries in the *Text Processing* category have similarities with activities in compiler theory, lexical analysis and parsing:

- Regex: a library that supports the creation of regular expressions, regular expression matching, searching for strings in a regular expression and replacing matches of a regular expression in a character sequence. In general, this library is used for static (that is, defined at compile-time) regular expressions.
- String Algo: STL has little support for string manipulation. The String Algorithm library fills this gap. It has support for a wide range of text and string manipulation functions.
- Tokenizer: this small library allows developers to break a string or other sequence of characters into so-called *tokens*. Examples of tokens are keywords, identifiers and punctuation symbols.
- Xpressive: a library that supports lexical analysis and the creation of regular expression objects. Both static (compile-time) and dynamic (run-time) regular expressions can be created. Nested expressions and semantic actions are also supported. A *semantic action* is a C++ function that is called whenever the parser successfully recognises a portion of the input.

For completeness, we should mention the Boost Spirit library that is an object-oriented, recursive-descent parser and output generation library for C++. It allows you to write grammars and format descriptions using a format similar to Extended Backus Naur Form (EBNF) directly in C++. This looks like a promising library for computational finance applications, for example creating payoff definition languages in C++.

There are a number of useful libraries to realise input-output:

- Assign: the ability to initialise data in STL containers using comma-separated lists of data. We see this as a *convenience library* and it speeds up development when testing and debugging code. It also makes the code more readable in general.
- Serialization: this library has functionality to save objects and data to *persistent storage* and to reconstruct the original objects from a persistent representation. It supports the serialisation of STL containers and

other complex objects. Furthermore, the library supports XML and binary formats.
- Iostreams: this library provides a framework for defining streams, stream buffers and i/o filters.

Finally, Boost has several libraries (the first two of which are rather specialised) that allow developers to control how memory is created and managed, for example:

- Flyweight: this library is an implementation of the GOF *Flyweight* design pattern to manage large numbers of highly redundant objects.
- Pool: pool allocation is a memory allocation scheme that is very fast, but limited in its usage. Using pools gives you more control over how memory is used in a program.
- Smart Pointer: smart pointers are objects which store pointers to dynamically allocated (heap) objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. There are six smart pointer class templates in this library.

We have now completed our short overview of a number of major libraries in Boost. We now need to determine if we can use them in applications and if so how to align them to current developer work practices and what the consequences are.

4 Applicability of Boost to Computational Finance

In this section we motivate why using the Boost libraries promotes code reliability and programmer productivity. Furthermore, we also discuss some of the other advantages such as run-time code efficiency as well as code portability and maintainability. In general, the advantages of adopting Boost should outweight the choice of creating (and maintaining) home-grown software libraries. To this end, we determine which components in a software system to create and which components to outsource to Boost. In a sense this is the approach taken in the past by Fortran developers who developed modular applications by using Fortran libraries such as NAG and IMSL as supporting tools. The advantage of this approach is that quant developers can concentrate on the job at hand while using modular libraries as black boxes. We would like to engender the same work practices in C++.

We now describe how to decide on which Boost libraries in a particular context. There are a number of dimensions to this problem and we enumerate them in order to separate concerns. To this end, we use the following steps as a general guideline:

- S1: what is the problem we wish to solve? For example, we may be interested in creating a Monte Carlo engine, a Finite Difference solver or a network application.
- S2: which libraries and library categories from Boost (see Figures 1 and 2) will be most useful in the current application?
- S3: determine how to use the chosen libraries from step S2 in new and/or existing applications.
- S4: design and implement the application using the Boost libraries.

The main effort in this project consists in executing steps S3 and S4. Steps S3 and S4 are concerned with software design and we can deploy Design Patterns (see GOF, 1995) in combination with object-oriented, generic and modular programming models. These issues will be discussed in Part II of this article when we discuss how to design and implement the following applications using C++ in combination with Boost, STL and generic design patterns:

- One-Factor and Two-Factor Alternating Direction Explicit (ADE) method for European and American option pricing.
- A generic Monte Carlo Framework for option pricing that generalises and extends the object-oriented engines in Duffy 2009.
- Creating matrix solvers (such as LU decomposition and the Conjugate Gradient Method (CGM)) using uBLAS library as substrate.
- Using the noncentral ChiSquared distribution to compute CIR (Cox-Ingersoll-Ross) bond prices.

We also discuss the advantages of using the Boost libraries. Some code examples can be found on www.datasimfinancial.com.

Conclusion

In this first part of a two-part series of articles on the Boost C++ libraries and their applications to computational finance, we gave an overview of what we think are the most important and useful libraries that quant developers can use in their applications. We also discussed the process that describes the steps to be taken when actually using the Boost libraries. In Part II we discuss this process in some detail when we discuss a number of relevant code applications.

Daniel J. Duffy works for Datasim Education. His main activities are software design and practise and the application of modern numerical methods to option pricing applications. He has a PhD in numerical analysis from Trinity College, Dublin.

REFERENCES

Demming, R. and Duffy, D.J. 2010. Introduction *to the Boost C++ Libraries Volume I – Foundations*. Datasim Press: Amsterdam.

Demming, R. and Duffy, D.J. 2011. *Introduction to the Boost C++ Libraries Volume II – Advanced Libraries*. Datasim Press: Amsterdam.

Duffy, D.J. and Kienitz, J. 2009. *Monte Carlo frameworks Building Customisable High performance C++ Applications* Wiley: Chichester.

GOF: Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* Boston: Addison-Wesley: Boston.

Josuttis, N. M. 1999. *The C++ Standard Library.* Addison-Wesley: Boston.

Josuttis, N. M. and Vandervoorde, D. 2003. C++ *Templates.* Addison-Wesley: Boston.