CHAPTER 1

A Tour of C++ and Environs

riverrun, past Eve and Adam's, from swerve of shore to bend of bay, brings us by a commodius vicus of recirculation back to Howth Castle and Environs

—(Joyce 1939).

1.1 INTRODUCTION AND OBJECTIVES

This book is the second edition of *Financial Instrument Pricing using C++* (2004), also written by the author. The most important reason for writing this hands-on book is to reflect the many changes and improvements to the C++ language, in particular due to the announcement of the new standard C++11 (and to a lesser extent C++14 and C++17). It feels like a new language compared to C++03 and in a sense it is. First, C++11 improves and extends the syntax of C++03. Second, it has become a programming language that supports the functional programming model in addition to the object-oriented and generic programming models.

We apply modern C++ to design and implement applications in computational finance, in particular option pricing problems using PDE/FDM, Monte Carlo and lattices models. We show the benefits of using C++11 compared to similar solutions in C++03. The resulting code tends to be more maintainable and extendible especially if the software system has been properly designed. We recommend spending some time on designing the software system before jumping into code and to this end we include a *defined process* to take a problem description, design the problem and then implement it in such a way that it results in a product that satisfies the requirements and that is delivered on time and within budget.

This book is a detailed exposition of the language features in C++ , how to use these features and how to design applications in computational finance. We discuss modern numerical methods to price plain and American options and the book is written in a hands-on, step-by-step fashion.

1.2 WHAT IS C++ ?

C++ is a general-purpose systems programming language that was originally designed as an extension to the C programming language. Its original name was "C with classes" and its object-oriented roots can be traced to the programming language *Simula* which was one of the

first object-oriented languages. C++ was standardised by the *International Organization for Standardization* (ISO) in 1998 (called the C++03 standard) and C++14 is the standard at the moment of writing. It can be seen as a minor extension to C++11, which is a major update to the language.

C++ was designed primarily for applications in which performance, efficiency and flexibility play a vital role. In this sense it is a systems programming language and early applications in the 1990's were in telecommunications, embedded systems, medical devices and Computer Aided Design (CAD) as well as first-generation option pricing risk management systems in computational finance. The rise in popularity continued well into the late 1990s as major vendors such as Microsoft, Sun and IBM began to endorse object-oriented technology in general and C++ in particular. It was also in this period that the Java programming language appeared which in time became a competitor to C++ .

C++ remains one of the most important programming languages at the moment of writing. It is evolving to support new hardware such as multicore processors, GPUs (graphics processing units) and heterogeneous computing environments. It also has a number of mathematical libraries that are useful in computational finance applications.

1.3 C++ AS A MULTIPARADIGM PROGRAMMING LANGUAGE

We give an overview of the programming paradigms that C++ supports. In general, a *programming paradigm* is a way to classify programming languages according to the style of computer programming. Features of various programming languages determine which programming paradigms they belong to. C++ is a multiparadigm programming language because it supports the following styles:

- *Procedural*: organises code around functions as typically seen in programs written in C, FORTRAN and COBOL. The style is based on structured programming in which a function or program is decomposed into simpler functions.
- *Object-oriented*: organises code around classes. A *class* is an abstract entity that encapsulates functions and data into a logical unit. We instantiate a class to produce *objects*. Furthermore, classes can be grouped into hierarchies. It is probably safe to say that this style is the most popular one in the C++ community.
- *Generic/template*: templates are a feature of C++ that allow functions and classes to operate with generic types. A function or class can then work on different data types.
- *Functional*: treats computation as the evaluation of mathematical functions. It is a declarative programming paradigm; this means that programming is done with expressions and declarations instead of statements. The output value of a function depends only on its input arguments.

The generic programing style is becoming more important and pronounced in C++ , possibly at the expense of the traditional object-oriented model which is based on class hierarchies and subtype (dynamic) polymorphism. Template code tends to perform better at run-time while many errors are caught at compile-time in contrast to object-oriented code where the errors tend to be caught by the linker or even at run-time.

The most recent style that C++ has (some) support for is *functional programming*. This style predates both structured and object-oriented programming. Functional programming has

its origins in *lambda calculus*, a formal system developed by Alonzo Church in the 1930s to investigate computability, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus. C++ supports the notion of lambda functions. A *lambda function* in C++ is an unnamed function but it has all the characteristics of a normal function. Here is an example of defining a *stored lambda function* (which we can define in-place in code) and we then call it as a normal function:

```
// TestLambda101.cpp
//
// Simple example of a lambda function
//
// (C) Datasim Education BV 2018
//
//

#include <iostream>
#include <string>

int main()
{
    // Captured variable
    std::string cVar("Hello");

    // Stored lambda function, with captured variable
    auto hello = [&cVar](const std::string& s)
    { // Return type automatially deduced

       std::cout << cVar << " " << s << '\n';
    };

    // Call the stored lambda function
    hello(std::string("C"));
    hello(std::string("C++"));

    return 0;
}
```

In this case we see that the lambda function has a formal input string argument and it uses a *captured variable* `cVar`. Lambda functions are simple but powerful and we shall show how they can be used in computational finance.

C++11 is a major improvement on C++03 and it has a number of features that facilitate the design of software systems based on a combination of *Structured Analysis* and object-oriented technology. In general, we have a *defined process* to decompose a system into loosely coupled subsystems (Duffy 2004). We then implement each subsystem in C++11. We discuss this process in detail in this book.

1.4 THE STRUCTURE AND CONTENTS OF THIS BOOK: OVERVIEW

This book examines C++ from a number of perspectives. In this sense it differs from other C++ literature because it discusses the full software lifecycle, starting with the problem description and eventually producing a working C++ program. In this book the topics are based on numerical analysis and its applications to computational finance (in particular, option pricing). In order to design and implement maintainable and efficient software systems we discuss each of the following *building blocks* in detail:

- A1: The new and improved syntax and language features in C++ .
- A2: Integrating object-oriented, generic and functional programming styles in C++ code.
- A3: Replacing and upgrading the traditional *Gang-of-Four* software design patterns to fit into a multiparadigm design methodology.
- A4: Analysing and designing large and complex software systems using a combination of top-down system decomposition and bottom-up object assembly.
- A5: When writing applications, determine how much of the features in A1, A2, A3 and A4 to use.

The chapters can be categorised into those that deal with modern C++ syntax and language features, those that focus on system design and finally those chapters that discuss applications. In general, the first ten chapters introduce new language features. Chapters 11 to 19 focus on using C++ to create numerical libraries, visualisation software in Excel and lattice option pricing code. Chapters twenty to twenty-nine are devoted to the finite difference method (FDM) on the one hand and to multithreading and parallel processing on the other hand. The last three chapters of the book deal with Monte Carlo methods. For easy reference, we give a one-line summary of each chapter in the book:

2: Smart pointers, move semantics, r-value references.

3: All kinds of function types; lambda functions, `std::bind`, functional programming fundamentals.

4: Advanced templates, variadic templates, `decltype`, template metaprogramming.

5: Tuples A-Z and their applications.

6: Type traits and compile-time introspection of template types.

7: Fundamental C++ syntax improvements.

8: IEEE 754 standard: operations on floating-point types.

9: A defined process to decompose systems into software components.

10: Useful data types: static and dynamic bitsets, fractions, date and time, fixed-sized arrays, matrices, matrix solvers.

11: Fundamental software design and data structures for lattice models.

12: Option pricing with lattice models. Both plain and early exercise cases are considered.

13: Essential numerical linear algebra and cubic spline interpolation.

14: A C++ package to visualise data in Excel (for example, a matrix or array of option prices from a finite difference solver). This package also allows us to use Excel for simple data storage.

15: Univariate statistical distributions in C++ and Boost. We also discuss some applications.

16: The different ways to compute the Bivariate Cumulative Normal (BVN) distribution accurately and efficiently using the Genz algorithm and by solving a hyperbolic PDE. Applications to computing the analytic solution of two-factor asset option pricing problems are given.

17: STL algorithms A-Z. Part I.

18: STL algorithms A-Z. Part II.

19: The solution of nonlinear equations and optimisation. The scope is restricted to the univariate case.

20: A mathematical background to convection-diffusion-reaction and Black Scholes PDEs.

21: A Software Framework for the Black-Scholes PDE using the finite difference method.

22: Extending the functionality of the framework in chapter 21; computing option sensitivities; an analysis of traditional software design patterns. We also discuss opportunities to upgrade software patterns to their multi paradigm extensions.

23: Path-dependent option problems using the finite difference method.

24: Ordinary differential equations (ODEs); theory and numerical approximations.

25: The Method of Lines (MOL) for PDEs.

26: Random number generation; some numerical linear algebra solvers.

27: Interoperability between ISO C++ and the Microsoft .NET Software Framework.

28: C++ Concurrency: Threads.

29: C++ Concurrency: Task.

30: Introduction to Parallel Patterns Library (PPL).

31: Single-threaded Monte Carlo simulation.

32: Multi-threaded Monte Carlo simulation.

Appendix 1: Multi-precision data types in C++ .

Appendix 2: Computing Implied Volatility.

This is quite a list of topics. The first ten chapters are essential reading as they lay the foundation for the rest of the book. In particular, chapters 2, 3, 4, 5, 7 and 8 introduce the most important syntax and language features. Chapters 11 to 19 are more or less independent of each other and we recommend that you read chapter 9 before embarking on chapters 11, 12 and 19. Chapters 17 and 18 discuss STL algorithms in great detail. Chapters 20 to 25 are devoted to PDEs and their numerical approximation using the finite difference method (FDM). They should be read sequentially. The same advice holds for chapters 28 to 30 and for chapters 31 to 32.

We have put some effort into creating exercises for each chapter. Reading them and understanding their intent is crucial in our opinion. Even better, actually programming these exercises is proof that you really understand the material.

1.5 A TOUR OF C++11: BLACK SCHOLES AND ENVIRONS

Since this is a hands-on book we introduce a simple and relevant example to show some of the new features in C++ . It is a kind of preview or *trailer*. In particular, we discuss the Black-Scholes option pricing formula and its sensitivities. We focus on the analytical solutions for stock options, futures contracts, futures options and currency options (see Haug 2007). The approach that we take in this section is similar to how mathematicians solve problems. We quote the famous mathematician Paul Halmos:

> …the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of generality is, in essence, the same as a small and concrete special case.

We now describe a mini-system that mirrors many of the design techniques and C++ language features that we will discuss in the other 31 chapters of this book. Of course, it goes without saying that we could implement this problem in a few lines of C++ code but the point of the exercise is to trace the system lifecycle from beginning to end by doing justice to each stage in the software process, no matter how small these stages are.

We use the following data type:

```
using value_type = double;
```

1.5.1 System Architecture

This is the first stage in which we scope the problem ("what are we trying to solve?") by defining the system scope and decomposing the system into loosely-coupled subsystems each of which has a single major responsibility (Duffy 2004). The subsystems cooperate to satisfy the system's core process which is to compute plain call and put option prices and their sensitivities. The architecture is based on a *dataflow metaphor* in which each subsystem processes input data and produces output data. Data is transferred between subsystems using a *plug and socket architecture* (Leavens and Sitarman 2000). In general, a system delivers a certain *service* to other systems. A service has a type and it can be connected to the service of another system if the other service is of *dual type*. We sometimes say that a service is a *plug* and the dual service is called a *socket*.

We represent the architectural model for this problem by the *UML* (Unified Modeling Language) *component diagram* in Figure 1.1. Each system does one job well and it interfaces with other systems by means of plugs and sockets. We first define the data that is exchanged between systems:

```
// Option data {K, T, r, sig/v} from Input system
template <typename T>
    using OptionData = std::tuple<T, T, T, T>;

// Return type of Algorithm system
// We compute V, delta and gamma
template <typename T>
    using ComputedData = std::tuple<T, T, T>;
```

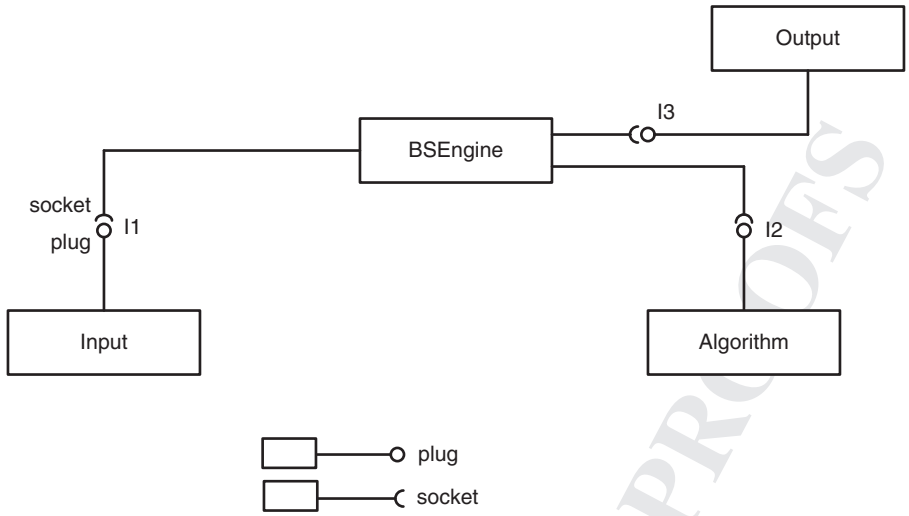

FIGURE 1.1 Context Diagram

We also define the interface to compute option price and sensitivities based on option data. To this end, we use *type-safe function pointers*:

```
// The abstract interface to compute V, delta and gamma
template <typename T> using IAlgorithm
      = std::function<ComputedData<T> (const OptionData<T>&
      optData, const T& S)>;
```

Having defined the data structures we now need to design the classes in Figure 1.1 that use them. To this end, we describe how to design these classes.

1.5.2 Detailed Design

In this case we apply the *policy-based design* idiom (Alexandrescu 2001) to model the classes in Figure 1.1. We are not necessarily endorsing this design as being the best one in general but it does show some of the important design features that we wish to highlight. We model the input and output systems as template parameters of a template class. We use *private inheritance* and *template-template parameters* to model this class that we call `SUD` (*System Under Discussion*) while we use a *signature-based approach* to model the algorithms to compute option prices and sensitivities:

```
template <typename T, template <typename T> class Source,
            template <typename T> class Sink>

class SUD : private Source<T>, private Sink<T>
{ // System under discussion, in this case for Black Scholes equa-
tion
```

```
private:
    // Define 'provides'/'requires' interfaces of satellite systems
    using Source<T>::getData;                    // Get input
    using Sink<T>::SendData;                     // Produce output
    using Sink<T>::end;                          // End of program

    // Conversion
    IAlgorithm<T> convert;
public:
    SUD(const IAlgorithm<T>& conversion): convert(conversion) {}
    void run(const T& S)
    {
       // The main process in the application
       OptionData<T> t1 = getData();             // Source
       ComputedData<T> t2 = convert(t1, S);      // Processing
       SendData(t2);                             // Sink

       end();                                    // Notification to
                                                    Sink
    }
};
```

Thus, this class inherits from its source and sink classes and it is composed of an algorithm.

The member function `run()` ties in the participating systems to produce the desired output. We should have a clear idea of the data flow in the system.

1.5.3 Libraries and Algorithms

Examining Figure 1.1 we see that the `Algorithm` subsystem computes option prices from option data. It has the same signature as the interface `IAlgorithm` and this means that we can configure these algorithms with any other *callable object* (for example, a free function, function object or lambda function) that has the same signature as `IAlgorithm`. This means that we do not need to create class hierarchies to achieve this level of flexibility. Furthermore, we can switch algorithms at run-time more easily than with traditional object-oriented technology.

C++ has support for a number of mathematical functions that are useful in computational finance. In this section we introduce the *error function* that allows us to compute the *univariate cumulative normal distribution*:

```
// Normal variates etc.
double n(double x)
{

    const double A = 1.0 / std::sqrt(2.0 * 3.14159265358979323846);
    return A * std::exp(-x*x*0.5);
}

// C++11 supports the error function
```

```
auto cndN = [](double x)
       { return 0.5 * (1.0 - std::erf(-x / std::sqrt(2.0))); };

double N(double x)
{ // The approximation to the cumulative normal distribution

    return cndN(x);
}
```

We now use these functions to compute the analytical solution of plain call and put option prices and their sensitivities. The aggregated value is placed in a *tuple* which is a new data type in C++11:

```
// Option Pricing; give price+delta+gamma
template <typename V>
    ComputedData<V> CallValues(const OptionData<V>& optData,
const V& S)
{
    // Extract data
    V K = std::get<0>(optData); V T = std::get<1>(optData);
    V r = std::get<2>(optData); V v = std::get<3>(optData);
    V b = r; // Stock option

    // Common functionality
    V tmp = v * std::sqrt(T);
    V d1 = (std::log(S / K) + (b + (v*v)*0.5) * T) / tmp;
    V d2 = d1 - tmp;

    V t1 = std::exp((b - r)*T); V t2 = std::exp(-r * T);
    V Nd1 = N(d1); V Nd2 = N(d2);

    V price = (S * t1 * Nd1) - (K * t2* Nd2);
    V delta = t1*Nd1;
    V gamma = (n(d1) * t1) / (S * tmp);

    return std::make_tuple(price, delta, gamma);
}


// Option Pricing; give price+delta+gamma
template <typename V>
    ComputedData<V> PutValues(const OptionData<V>& optData,
const V& S)
{
    // Extract data
    V K = std::get<0>(optData); V T = std::get<1>(optData);
```

```
    V r = std::get<2>(optData); V v = std::get<3>(optData);
    V b = r; // Stock option

    // Common functionality
    V tmp = v * std::sqrt(T);
    V d1 = (std::log(S / K) + (b + (v*v)*0.5) * T) / tmp;
    V d2 = d1 - tmp;

    V t1 = std::exp((b - r)*T); V t2 = std::exp(-r * T);
    V Nmd2 = N(-d2); V Nmd1 = N(-d1);

    V price = (K * t2 * Nmd2) - (S * t1* Nmd1);
    V delta = t1*(Nmd1 - 1.0);
    V gamma = (n(d1) * t1) / (S * tmp);

    return std::make_tuple(price, delta, gamma);
}
```

We see how useful tuples are as return types of functions. This is a more efficient solution than creating a separate function for each of an option's price, delta and gamma functions.

1.5.4 Configuration and Execution

We now discuss how to configure the objects and interfaces in Figure 1.1. This usually takes place by either creating the needed objects directly in the body of `main()` or by outsourcing this process to *creational design patterns* and *builders* (see GOF 1995). In general, we need to choose what we want. As an example, we consider the following hard-coded *source* and *sink* classes:

```
template <typename T> class Input
{
public:

   static OptionData<T> getData ()
   { // Function object

      T K = 65.0; T expiration = 0.25;
      T r = 0.08; T v = 0.3;
      OptionData<T> optData(K, expiration, r, v);

      return optData;
   }
};

template <typename T> class Output
{
```

```
    public:

       void SendData (const ComputedData<T>& tup) const
       {
          ThreadSafePrint(tup);
       }

       void end() const
       {
          std::cout << "end" << std::endl;
       }

    };
```

Multiple threads can write to the console in a non-deterministic way. For this reason we create a lock on the console if and when we port the single threaded code to a multithreaded program. The corresponding *thread-safe code* is:

```
template <typename T>
    void ThreadSafePrint(const ComputedData<T>& tup)
{ // Function to avoid garbled output on tbe console

    std::mutex my_mutex;
    std::lock_guard<std::mutex> guard(my_mutex);
    std::cout << "(" << std::get<0>(tup) << "," << std::get<1>(tup)
              << "," << std::get<2>(tup) << ")\n";
}
```

We create classes to model calls and puts as follows:

```
template <typename T> class Processing
{
public:

    ComputedData<T> convert(const OptionData<T>& optData,
                            const T& S) const
    {
       return CallValues(optData, S);
    }

    ComputedData<T> operator () (const OptionData<T>& optData,
                                 const T& S) const
    {
       return CallValues(optData, S);
    }
```

```
};

template <typename T> class ProcessingII
{
public:

    ComputedData<T> convert(const OptionData<T>& optData,
                            const T& S) const
    {
       return PutValues(optData, S);
    }

    ComputedData<T> operator () (const OptionData<T>& optData,
                                 const T& S) const
    {
       return PutValues(optData, S);
    }

};
```

Having created the objects that we need we are now in a position to run the application:

```
Processing<value_type> converter;

// Calls
SUD<value_type, Input, Output> callPricer(converter);
value_type S = 60.0;
callPricer.run(S);

// Puts
ProcessingII<value_type> converter2;
SUD<value_type, Input, Output> putPricer(converter2);
value_type S2 = 60.0;
putPricer.run(S2);
```

The output in this case is:

```
(2.13337,0.372483,0.0420428)
end
(5.84628,-0.372483,0.0420428)
end
```

1.6 PARALLEL PROGRAMMING IN C++ AND PARALLEL C++ LIBRARIES

The *C++ Concurrency* library supports both multithreading and multitasking, that is creating programs that are executed by multiple independent threads of control. Multithreading is *preemptive* in the sense that the scheduler allocates a fixed amount of time (a *quantum*) to each thread after which time the thread reverts to *sleep or* to *wait-to-join* mode. The library also supports the creation of programs and algorithms by decomposing them into components that can potentially run in parallel with little interaction between them. In general terms, *potential or exploitable concurrency* involves our being able to structure code to permit a problem's subproblems to run on multiple processors. Each subproblem is implemented by a task. A *task* (Quinn 2004) is a program in local memory in combination with a collection of I/O ports. Tasks send data to other tasks through their output ports and they receive data from other tasks through their input ports.

We take a simple example. In this case we parallelise the code in section 1.5.4. Both of the following solutions are special cases of a more general *fork-join idiom* in which a single (main) thread creates two child threads. Each child thread executes code independently of the other child threads. Since the shared data is read-only in this special case then there is no danger of non-deterministic behaviour. For both solutions the main thread or task must wait on its children to complete before it can proceed.

We now describe the solution using *C++ Concurrency*. We first encapsulate the algorithms in stored lambda functions:

```
// Parallel execution
auto fn1 = [&converter](value_type S)
{
    SUD<value_type, Input, Output> callPricer(converter);
    callPricer.run(S);
};

auto fn2 = [&converter2](value_type S)
{
    ProcessingII<value_type> converter2;
    SUD<value_type, Input, Output> putPricer(converter2);
    putPricer.run(S);
};
```

The stock value is:

```
value_type stock = 60.0;
```

The solution using C++ threads is:

```
// Threads
std::thread t1(fn1, stock);
```

```
std::thread t2(fn2, stock);

// Wait on threads to complete
t1.join(); t2.join();
```

For the task-based solution we use C++ *asynchronous futures*:

```
// Asynchronous Tasks
std::future<void> task1(std::async(fn1, stock));
std::future<void> task2(std::async(fn2, stock));

// Wait on threads to complete
task1.wait(); task2.wait();

// Get results from tasks
task1.get(); task2.get();
```

Our final example is to show how to parallelise this code using the *OpenMP* library (Chapman, Jost and Van der Pas 2008). We do not discuss this library in this book but we recommend it as a good way of learning how to write multithreaded applications before moving to *C++ Concurrency*. In this case we create an array of threads and we execute them using *loop-level parallelism*:

```
// OMP solution
std::vector <std::function<void(value_type)>> tGroupFunctions
            = { fn1, fn2 };

value_type stock = 60.0;

#pragma omp parallel for
for (std::size_t i = 0; i < tGroupFunctions.size(); ++i)
{
    tGroupFunctions[i](stock);
}
```

The output produced from this code is:

```
(2.13337,0.372483,0.0420428)
end
(5.84628,-0.372483,0.0420428)
end
(2.13337,0.372483,0.0420428)
```

```
end
(5.84628,-0.372483,0.0420428)
end
(2.13337,0.372483,0.0420428)
end
(5.84628,-0.372483,0.0420428)
end
```

In general, writing parallel applications using tasks is easier than using threads because tasks hide many of the tricky *synchronisation* and *notification* use cases when using threads and it is possible to apply the system decomposition technique (Duffy 2004) to help create *task dependency graphs* that we then implement in C++ .

1.7 WRITING C++ APPLICATIONS; WHERE AND HOW TO START?

This book centres around the modern multiparadigm language features in C++11 and C++14. We discuss most of the essential language features in the first ten chapters. Later chapters introduce a number of topics that build on these first ten chapters. These are shown in Figure 1.2 in the form of a *concept map*. The main goal of this book is to develop tools to show how to design and implement software systems for computational finance applications. Based on this remark we show in Figure 1.2 how we will achieve the goal by displaying the main concepts and their relationship with C++ . In general, we use standard C++ in most of the examples unless otherwise mentioned. The book is self-contained in the sense that we prefer to use standard libraries (such as those in C++ as well as *Boost* and *Quantlib*) rather than proprietary libraries. The code for the numerical methods in the book is self-contained and has been developed by the author.

We give a short description of the concepts in Figure 1.2 and their relationship with the applications in this book:

- A: We develop code for well-known numerical methods such as numerical differentiation, interpolation, numerical quadrature, matrix algebra, finding the zeroes of nonlinear equations and optimisation problems.
- B: We use the *Boost* C++ libraries when we need functionality that is not (yet) in C++ . Much of the new functionality in C++ had its roots in *Boost*. In general, the *Boost* libraries tend to be reasonably well-documented. We recommend the *Boost Math Toolkit* for numerical applications. We also use the *Boost odeint* library to numerically solve systems of ordinary differential equations (ODEs).
- C: This book is unique in our opinion because it introduces a *defined process* to analyse, design and implement any kind of software system in a step-by-step fashion. The process is based on the author's experience as requirements analyst and software architect in several application domains (see Duffy 2004 where these domains have been documented). The process is a fusion of *Structured Analysis* (De Marco 1978) and the object-oriented paradigm. We apply the process to creating *design blueprints* for Finite Difference (FDM) and Monte Carlo applications.

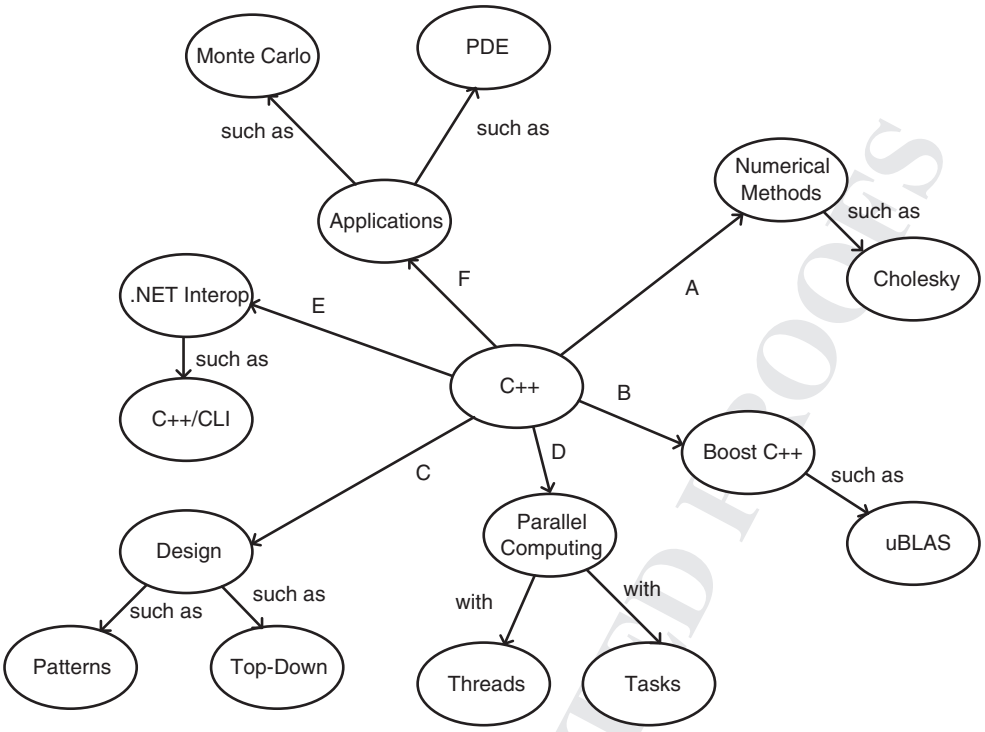

FIGURE 1.2 C++ and Environs

- D: Modern laptop and desktop computers have multiple processors on board. This opens the door to developing multithreaded and parallel code to increase the *speedup* of programs. C++ offers support in the *C++ Concurrency* library.
- E: We decided to include a chapter on the .NET language C++/CLI that bridges the (native/ISO) C++ and .NET worlds. C++ is and remains a *systems programming language* which makes it suitable for certain kinds of applications. The C++/CLI language then allows us to create C++ applications that can call .NET functionality on the one hand while it is possible to create .NET wrappers for native C++ classes and call them from C# code. This approach promotes code reusability and helps C# developers because they do not have to learn C++. All they need is to wrap C++ code in .NET wrappers.
- F: In this book, we design option pricing software using lattice methods, Monte Carlo and Partial Differential Equation (PDE)/FDM methods. We propose a range of numerical methods, design patterns and design styles. Furthermore, we use *C++ Concurrency* to parallelise the corresponding algorithms.

1.8 FOR WHOM IS THIS BOOK INTENDED?

C++ is probably one of the most difficult programming languages to master. The learning curve is much steeper than that of other object-oriented languages such as C# and Java, for example.

The only real way to learn C++ is to program in C++ and it is for this reason we say that you need to have a number of years of solid C++ experience writing code and applications. **In other words, this is not a beginner's book to learn C++.**

The primary focus of this book is to design and implement maintainable and extendable applications and it is suitable for front-office and middle-office quant developers who use C++ in their daily work. The book is also useful for software architects and project managers who wish to understand and manage software projects.

This book is also useful for MSc students in finance. We would hope that the step-by-step approach will help them structure their theses.

The first ten chapters can be read by a range of C++ developers because the topics are application-independent and to our knowledge this is the first book on the new features in C++11. Chapters 20 to 25 could also be of interest to mathematical physicists.

1.9 NEXT GENERATION DESIGN AND DESIGN PATTERNS IN C++

The approach taken in this book is special in the sense that we find it important to have an idea of the software system that we wish to build before developing the code to implement it. This is in the interest of developer productivity. We need *design blueprints* that describe the system at a level higher than raw C++ code. Computer Science is not yet an engineering discipline and there are few *standardised* design processes and standards to help developers analyse, design and implement C++ applications. One possible exception is the famous *software design patterns* that were first published in GOF 1995 although when used on their own they do not ensure that the software system will be stable.

Design patterns have become very popular in the last 20 years as witnessed by the number of books devoted to them for object-oriented languages such as Java, C#, C++ and others. Neither the structure nor the number of patterns have changed much in the last 20 years as most the literature seems to imitate the 23 patterns in GOF 1995. In a sense, these patterns were invented when C++ was still in its infancy and when it only supported the traditional object-oriented technology based on subtype polymorphism (`virtual` functions) and class hierarchies. Our basic premise is that the GOF patterns represent knowledge that has not adapted to improvements in software, hardware and development methods.

The GOF design patterns are based on the object model which means that the patterns are implemented using objects, classes and class hierarchies in combination with subtype polymorphism. This means that an abstract requirement regarding the flexibility of a software design must be turned into an explicit data model by introducing a *proxy* for a non-computable concept. This process is called *reification* and it allows any aspect of a programming language to be expressed in the language itself.

It is possible to *upgrade* the GOF patterns in a number of ways:

- S1: Keep and use the patterns in their current form without change.
- S2: Improve the patterns by using new improved C++ functionality such as shared pointers and the syntax that we discussed in the first 10 chapters of this book.
- S3: Reengineer those patterns that can be implemented more easily and correctly using the generic and functional programming models, for example.

- S4: Do not (yet) use design patterns but instead postpone their use in the design trajectory for as long as possible. Instead, we use the system decomposition techniques of chapter 9 and we hope to achieve the same (and improved) levels of flexibility as with GOF patterns but then by other means, specifically by defined standardised interfaces between components.

1.10 SOME USEFUL GUIDELINES AND DEVELOPER FOLKLORE

We conclude this chapter with some guidelines on the estimation, planning and management of software projects. The size of a project can range from a one-person software endeavour lasting three months to a 30-person application with a lifetime of five years, for example. Some of the principles underlying our design approach can be summarised by the steps that György Pólya described when solving a mathematical problem (Pólya 1990):

1. First, you have to *understand the problem*.
2. After understanding, then *make a plan*.
3. *Carry out the plan*.
4. *Look back* at your work. How could it be better?

We see these steps as being applicable to the software development process in general and to the creation of software systems for computational finance in particular. Getting each step right saves time and money. In short, we take the following tactic regarding software projects: *get it working, then get it right and only then get it optimised* (in that order). In the current context we translate these steps into a defined software process (as explained in chapter 9) in order to avoid some scary outcomes, for example:

> "A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems."

We use the following general principles when developing software systems:

1. Understand the problem as soon as possible. Can you explain the problem to non-developers?
2. Can you develop a software prototype in a few days?
3. Scope the problem by identifying the boundaries of the software system.
4. Decompose the system into loosely-coupled subsystems.
5. Use a suitable combination of the object-oriented, generic and functional programming styles.

Each developer has her own way to design software systems. There is no *silver bullet* (Brooks 1995).

1.11 ABOUT THE AUTHOR

Daniel J. Duffy is the author of both the first and second editions of *Financial Instrument Pricing using C++* . He started his company Datasim in 1987 to promote C++ as a new object-oriented language for developing applications. He played the roles of developer, architect and requirements analyst to help clients design and analyse software systems in areas such as Computer Aided Design (CAD), process control and hardware-software systems, logistics, holography (optical technology) and computational finance. He used a combination of top-down functional decomposition and bottom-up object-oriented programming techniques to create stable and extendible applications (for a discussion, see Duffy 2004 where we have grouped applications into domain categories). He also worked on engineering applications in oil and gas and semiconductor industries using a range of numerical methods (for example, the finite element method (FEM)).

Daniel Duffy has BA, MSc and PhD degrees in pure and applied mathematics (from Trinity College, The University of Dublin) and has been active in promoting partial differential equation (PDE) and finite difference methods (FDM) for applications in computational finance. He was responsible for the introduction of the Fractional Step (Soviet Splitting) method and the Alternating Direction Explicit (ADE) method in computational finance.

He is the originator of two popular C++ online courses on www.quantnet.com in cooperation with Quantnet LLC and Baruch College (CUNY), NYC. He also trains quant developers around the world. He can be contacted dduffy@datasim.nl. The official Datasim site is www.datasimfinancial.com.

1.12 THE SOURCE CODE AND GETTING THE SOURCE CODE

The C++ code in this book is based on the C++11 standard (and later versions). The only exception is the code in chapter 14 in which we introduce the Excel Driver library and chapter 27 where we discuss interfacing between C++ and Microsoft's .NET Framework. Furthermore, the code that is presented in each chapter is machine-readable and has been tested beforehand. Our development environment is Visual Studio C++ which supports all the C++ functionality that we present in this book. We have not tested the code using other compilers but we would not expect major issues. It is the responsibility of the reader to know how to install the compiler, *Boost C++* libraries and *Quantlib*.

Regarding copyright, legitimate owners of the book are entitled to the source code which can be used for personal use provided you do not remove the copyright notice in the source code `(C)Datasim Education BV 2018`.

For further queries concerning training and support, please contact me directly at dduffy@datasim.nl,