Part I

C++ Essential Skills

1
Introduction to C++ and Quantitative Finance

1.1 INTRODUCTION AND OBJECTIVES

In this chapter we give an overview of the C++ programming language, its relationship with Quantitative Finance (QF) and why C++ is suitable for complex applications in this domain. In particular, we discuss the various programming paradigms and how each paradigm is suited to software projects in QF. Furthermore, we shall describe how these paradigms can be dovetailed to help us build efficient and robust code. Last, but not least, our objective is to create software that is also easy to understand and to maintain. This is an extremely important requirement because a large C++ application consisting of a network of tightly coupled objects can be difficult to maintain at best, and a nightmare at worst. In this day and age the focus is on developing flexible frameworks that can be easily and quickly customised to changing requirements in the market place. To this end, we are convinced that C++ can realise these requirements if and only if we design our software systems in a correct and proper way.

You can skip this chapter if you wish to start as soon as possible on this C++ journey. Then you should go immediately to chapter two. Before doing so, however, we would strongly advise you to read section 1.3 (Programming Paradigms). This book complements my book on C++ for financial instrument pricing (Duffy, 2004) because the latter book assumes previous C++ knowledge and the current book takes a more leisurely pace by discussing each topic in detail.

If we compare this book to an opera, then this chapter would correspond to the overture. It sets the tone by providing some background information on C++ and its relevance and applicability to Quantitative Finance.

1.2 A SHORT HISTORY OF C++

The object-oriented way of thinking and programming (we call this a *paradigm*) is almost fifty years old and it has its origins in the programming language Simula that was developed in Norway. Simula was the first language to support the concept of a class as we know it in its current form.

C++ has its origins in the early 1980's when its inventor, Dr Bjarne Stroustrup (Stroustrup, 1997) was working at AT&T. The original name for the language was 'C with classes' because the language was developed as an object-oriented extension to the programming language C while still remaining compatible with it. This very fact may be a reason why C++ has weathered the storm: the legacy C code that organisations maintained could be upgraded to C++. C++ is compatible with C and was called a 'better C' in those early days.

The late 1980's can be seen as the period when C++ came out of the laboratories and began to manifest itself in mainstream applications. The first C++ compiler (actually, precompiler because C++ code was compiled to C code) was from a company called Glockenspiel in Dublin in 1988 and it was in this period that the current author started to work with C++.

The early 1990's saw a remarkable growth in interest in the object-oriented (OO) paradigm in general and in C++ in particular. As with many new technologies promises were made that were not met. For example, it was believed in some circles that OO would solve all software ails and that a new industry would emerge in which application builders would purchase reusable class libraries from companies that could be described as 'class library builders'. The most important applications in this period were in the following domains: simulation (Computer Aided Design (CAD), Computer Graphics), telecommunications and real-time applications (for example, medical devices and process control). It was during this period that the current author worked on an early version of a pricing and risk management system in C++.

At the moment of writing we can conclude that the object-oriented paradigm is (justifiably) accepted as a necessary precondition for success in software development. However, it is not sufficient in the sense that blind adherence to it will not automatically lead to good results. First, there are other software paradigms that complement and even compete with the object-oriented paradigm and second the paradigm can be taken to extremes, as we have seen in the past. We discuss these problems and risks in this chapter and we provide some guidelines on how to turn our object-oriented projects into success stories.

1.3 C++, A MULTI-PARADIGM LANGUAGE

One of the features of C++ is that it supports many kinds of programming paradigms, unlike some languages that are 'pure' object-oriented languages (in the sense that every piece of software must be an object or a class). Instead, we can write applications that are a mixture of different programming styles. Whether this is a wise thing to do is debatable but that is not the issue at the moment. In general, the author does not believe that a single paradigm is flexible enough to encompass every possible kind of application and in general some parts of an application can be written in an object-oriented fashion while other parts can and should be written using a modular approach, reminiscent of Fortran, C and Cobol.

1.3.1 Object-oriented paradigm

This paradigm is based on the concept of a class. Classes have their origins in philosophy, logic and cognitive psychology (Eysenck and Keane, 2000). In particular, the theory of concepts has been an important influence on the development of the object paradigm. There are a number of theories, one of which is the *defining attribute view.* This view was developed and elaborated by the German logician Frege (Frege, 1952). Frege maintained that a concept can be characterised by a set of defining attributes or semantic features. He distinguishes between a concept's intension and extension. The *intension* of a concept consists of the set of attributes that determine what it is to be a member of the concept. This idea is similar to a class in *class-based object-oriented languages*. The *extension* of a concept is the set of entities that are members of the concept. This idea corresponds to class instances or objects. Some features of the defining attribute view are:

- The meaning of a concept is captured by its defining attributes
- Attributes are atomic building blocks for concepts
- Attributes are necessary and sufficient for defining members of a concept
- There is no doubt about whether an entity is in the concept; there are clear-cut boundaries between members and non-members of the concept

- All members of the concept are equally representative of the concept; we cannot say that one member is more typical of the concept than another member
- When concepts are organised in a hierarchy the defining attributes of the more specific concept (for example, a sparrow) include all the attributes of the superordinate concept (in this case, bird).

These features are implemented in many class-based object-oriented languages such as C++, Java and C#. In this case we first define a class consisting of data and functions and we then create objects or so-called instances of the class by initialising the data in the class. Looking back in hindsight (which is always easy), the author concludes that these assumptions are too restrictive for certain types of applications. There are other object-oriented languages where there is no class concept. Instead, if we wish to create an object we must clone or copy it from an existing *prototypical* object. The Self language is one example of a so-called classless object-oriented language.

Let us take a simple example. In this case we wish to model one-factor plain options (in other words we can only exercise at the maturity date T). An option can be a call option or a put option. When we model this as a class we must discover its attributes and the messages to which instances (objects) of the class respond to. The attributes are:

- The risk-free interest rate: r
- The volatility of the relative price change: σ
- The strike price: K
- The time to expiration (in years): T
- The cost-of-carry: b

These attributes are just names and when we create instances of the class we must assign values to them, for example (Haug, 1998):

- Volatility $\sigma = 0.15$
- Strike Price $K = 490$
- Time to expiry $T = 0.25$ (3 months)
- Risk-free interest rate $r = 0.08$
- Cost-of-carry $b = 0.03$

We thus see that the object is concrete while its corresponding class is abstract. Having defined the object's data we may speculate on the kinds of information we wish to extract from the object. Since this is a context-sensitive question we would expect different answers from various stakeholder groups such as:

- Traders
- Quantitative analysts
- Risk managers
- IT personnel

Each group has its own requirements and features that they would like to have. For example, a common set of requirements might be:

- Calculate the option price
- Calculate an option's sensitivities (for hedging applications)
- The ability to support constant, time-dependent and stochastic volatility models
- Export option-related information to a spreadsheet, for example Excel

These features will be implemented by one or more so-called member functions. In order to reduce the scope we concentrate on the pricing and hedging functionality. For example, the price for a one-factor plain call or put option is known analytically:

```
double CallPrice()
{
      double tmp = sig * sqrt(T);
      double d1 = ( log(U/K) + (b+ (sig*sig)*0.5 ) * T )/ tmp;
      double d2 = d1 - tmp;

      return (U * exp((b-r)*T) * N(d1)) - (K * exp(-r * T)* N(d2));
}
```

and

```
double PutPrice()
{
      double tmp = sig * sqrt(T);

      double d1 = ( log(U/K) + (b+ (sig*sig)*0.5 ) * T )/ tmp;
      double d2 = d1 - tmp;
      return (K * exp(-r * T)* N(-d2)) - (U * exp((b-r)*T) * N(-d1));
}
```

In this code we use the variable `U` to denote the underlying variable.

1.3.2 Generic programming

This is a paradigm that can be a competitor of the object-oriented paradigm and it can also be used in conjunction with the latter paradigm.

When we design a software entity using the generic paradigm we try to stop thinking about hard-wired data types and so on. We then design the software using generic underlying types. When we wish to work with specific data types we *instantiate* or clone the software entity by replacing the generic types by these specific types. The compiler takes care of these replacement issues and it checks that the specific data types satisfy the interface requirements demanded by the generic type.

Let us take a simple example. Suppose that we wish to define a function that calculates the maximum of two numbers. In C++ we realise this using a *template function*:

```
template <class Numeric>
      Numeric Max(const Numeric& x, const Numeric& y);
```

This template function in C++ accepts two parameters of a generic type and then calculates their maximum. The code for the function is easy to read if you have programmed in any high-level language:

```
template <class Numeric>
Numeric Max(const Numeric& x, const Numeric& y)
{
```

```
      if (x > y)
            return x;
      return y;
}
```

The only difference with normal programming practice in this case is that we need to give the compiler a hint that we are working with generic data types and not with specific ones. An example of use is:

```
long dA = 12334; long dB = 2;
cout << "\n\nMax and min of two numbers: " << endl;
cout << "Max value is: " << Max<long>(dA, dB) << endl;
```

Concluding, when we work in this way we write the software once and reuse it many times. We have applied the generic paradigm to quantitative finance applications in Duffy (2004).

1.3.3 Procedural, modular and functional programming

The programming language Fortran (Formula Translation) has been the most successful language of all time for scientific, mathematical and engineering applications. It is ideally suited to problems involving data structures such as vectors and matrices and the corresponding algorithms that use these data structures. Hundreds of libraries have been built to help Fortran programmers, for example:

- Numerical linear algebra
- Initial value problems
- Ordinary and partial differential equations
- And many more . . .

Fortran achieves this level of reusability by the use of subroutines and modules. A module is a function that produces output from input. It is not a member function of a class and hence we do not need to create an object in order to use it. Object-oriented purists may frown on this approach but my answer would be: not everything is, or needs to be an object.

We have applied the modular paradigm to quantitative finance applications in Duffy (2004).

1.4 C++ AND QUANTITATIVE FINANCE: WHAT'S THE RELATIONSHIP?

C++ has become very popular in Quantitative Finance and its importance will grow rather than diminish in the coming years (in my humble opinion). It may not be the most elegant and usable language out there but – all things being equal – it is the most flexible and adaptable language. It is an ISO standard, which means your C++ code will also work in 20 years time!

I could say much more, but for good or bad there is no way we can ignore C++. An important point is that potential employers wish to see employees with C++ experience. Accepting this fact, and the fact that so many people wish to learn the language (and learn it well) I have decided to write this book. I hope that it will help you in your career.

1.5 WHAT IS SOFTWARE QUALITY?

The ISO 9126 standard (see Kitchenham and Pfleeger, 1996) is a description of a set of characteristics that measures the quality of software products. It consists of six orthogonal

quality characteristics that describe how good a product is. We discuss them because they are very useful in all phases of the software development lifecycle (in particular, business modelling) and not just in the more solution-dependent stages such as design, coding and maintenance. In fact, many managers think in term of these characteristics, albeit implicitly. Furthermore, each characteristic has several sub-characteristics.

The six characteristics are:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Functionality refers to the capability of a system (in fact, the software that implements the system) to satisfy user needs. These needs may be explicitly stated but they can also be implicit. This characteristic has five sub-characteristics:

- *Suitability*: this has to do with functions for specified tasks and their appropriateness for their tasks
- *Accuracy*: this has to do with the problem of producing correct and agreed results or the agreed effect
- *Interoperability*: this has to do with the ability to interact with other systems. An important precondition is that the systems are predefined
- *Compliance*: this sub-characteristic refers to whether the system adheres to standards and conventions such as regulations, domain-related standards and the law
- *Security*: this has to do with the ability of the system to prevent unauthorised access, whether it be deliberate or accidental

Reliability is concerned with how a system maintains a given level of performance over some given period of time. We must also state the conditions under which the system performs

This characteristic has three sub-characteristics:

- *Maturity*: has to do with the frequency of failure in the system. Most failures are caused by so-called faults
- *Fault tolerance*: refers to the ability of the system to maintain a specified level of performance. We must specify the duration of time in which that level is to be maintained. Disturbances compromise this level of performance. These disturbances are caused by software faults and bad interfaces, for example
- *Recoverability*: this refers to the capability to re-establish previous levels of performance. For example, we could consider the time and effort it takes to recover information and data after a system crash

Usability refers to the effort that is needed in order to 'use' an application or system. Of course, there are many kinds of users of a system and each one has a definition of usability. For example, there are both direct and indirect users of the system. It is important to define what developers, managers and users of the software mean by usability.

This characteristic has three sub-characteristics:

- *Understandability*: the effort needed to recognize logical concepts and their applicability
- *Learnability*: the effort needed to learn the application, for example how often the user manual is consulted
- *Operability*: the effort for operation and operational control, for example backup and file management

Efficiency refers to the level of performance and the amount of resources needed to achieve the performance.

This characteristic has two sub-characteristics:

- *Time behaviour*: this is related to response and processing times
- *Resource behaviour*: has to do with the amount of resources needed to perform functions. This sub-characteristic is also concerned with how long the resources are held while performing the functions

Maintainability refers to the effort needed to make specified modifications. These modifications may include corrections, improvements or adaptation. In general, modifications are caused by changes in the environment and by changes to requirements and functionality.

This characteristic has four sub-characteristics:

- *Analysability*: the effort needed for diagnosis or deficiency detection. We wish to detect the causes of failure in this case and to identify parts of the system requiring modification
- *Changeability*: this is related to the effort that is needed for modification, fault removal or environmental change
- *Stability*: the risk of unexpected effect of modification. This is the sub-characteristic that gives managers and project leaders nightmares. Traditional object-oriented software projects tend to suffer from this problem because of their inherent bottom-up approach, aggravated by overuse of the C++ inheritance mechanism. The end-result is a tightly coupled set of object networks that *can* (and usually) does lead to huge maintenance problems
- *Testability*: the effort that is needed to validate the modified software or the effort that is needed to test it

Portability refers to the ability of software in a system to be transferred from one environment to another environment. This includes organisational, hardware and software environments.

This characteristic has four sub-characteristics:

- *Adaptability*: the opportunity for adaptation of software to different specified environments. This implies that no other actions should be applied or changes made
- *Installability*: the effort needed to install software in a specified environment
- *Conformance*: does software adhere to standards or conventions?
- *Replaceability*: the opportunity and effort of using software in place of other software in the same environment. This sub-characteristic may also include attributes of both installability and adaptability

1.6 SUMMARY AND CONCLUSIONS

In this chapter we have given an overview of a number of programming *paradigms* and how they are supported in C++. Furthermore, we gave a short history of C++ and its applications

during the last 25 years. Finally, we gave an introduction to the ISO 9126 standard that describes the quality of *software products*. Just like my car or washing machine, we wish to create software applications that are extendible and easy to maintain and of course, fast. We realise the first two requirements by improving design and programming skills while the third requirement can be realised by a clever synergy between software and hardware.

1.7 EXERCISES

1. Which programming language(s) are you using at the moment? In how far does it support the following paradigms:
   - Object oriented programming
   - Generic programming (as is seen with C++ templates)
   - Procedural and modular programming

   Are there things you would like to do with the language that are not possible?

2. In how far do the following languages support the above three paradigms: Java, VBA, C#, C, Cobol, Fortran, APL, PL/I, Maple, Matlab, Smalltalk, VB.NET?

3. Which ISO 9126 characteristics are important for the following kinds of software projects:
   (a) A throwaway prototype application to test if a new pricing model is accurate
   (b) A COM Addin (written in C++) that will be used on the trading floor
   (c) A large system using the finite difference method that will be updated, extended and improvement over a period of years

4. What are the three most important ISO 9126 characteristics in general in your opinion?