Wiley Finance Series

Financial Instrument Pricing Using C++

Second Edition

DANIEL J. DUFFY

WILEY

Financial Instrument Pricing Using C++ 2e

Founded in 1807, John Wiley & Sons is the oldest independent publishing company in the United States. With offices in North America, Europe, Australia and Asia, Wiley is globally committed to developing and marketing print and electronic products and services for our customers' professional and personal knowledge and understanding.

The Wiley Finance series contains books written specifically for finance and investment professionals as well as sophisticated individual investors and their financial advisors. Book topics range from portfolio management to e-commerce, risk management, financial engineering, valuation and financial instrument analysis, as well as much more.

For a list of available titles, visit our website at www.WileyFinance.com.

Financial Instrument Pricing Using C++ 2e

DANIEL J. DUFFY

WILEY

This edition first published 2018

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

Library of Congress Cataloging-in-Publication Data
Names: Duffy, Daniel J., author.
Title: Financial instrument pricing using C++ / Daniel J. Duffy.
Description: Second Edition. | Hoboken : Wiley, [2018] | Series: Wiley finance series | Revised and updated edition of the author's Financial instrument pricing using C++, c2004. | Includes bibliographical references and index. |
Identifiers: LCCN 2018017672 (print) | LCCN 2018019643 (ebook) | ISBN 9781119170495 (Adobe PDF) | ISBN 9781119170488 (ePub) | ISBN 9780470971192 (hardcover) | ISBN 9781119170495 (ePDF) | ISBN 9781119170518 (Obook)
Subjects: LCSH: Investments–Mathematical models. | Financial engineering. | C++ (Computer program language)
Classification: LCC HG4515.2 (ebook) | LCC HG4515.2 .D85 2018 (print) | DDC 332.60285/5133–dc23
LC record available at https://lccn.loc.gov/2018017672

A catalogue record for this book is available from the British Library.

ISBN 978-0-470-97119-2 (hardback) ISBN 978-1-119-17049-5 (ePDF)
ISBN 978-1-119-17048-8 (ePub) ISBN 978-1-119-17051-8 (Obook)

10 9 8 7 6 5 4 3 2 1

Cover design: Wiley
Cover images: © whiteMocca/Shutterstock

Set in 10/12pt Times by Aptara Inc., New Delhi, India
Printed in Great Britain by TJ International Ltd, Padstow, Cornwall, UK

Contents

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++*, also written by the author (Duffy, 2004B). 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 partial differential equation (PDE)/finite difference method (FDM), Monte Carlo and lattice 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 1990s 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 programming 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 automatically 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, determining 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 20 to 29 are devoted to the finite difference method 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 multiparadigm 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. Multithreaded Monte Carlo simulation.

Appendix 1: Multiprecision 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. They should be read sequentially. The same advice holds for Chapters 28 to 30 and 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 Modelling 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>;
```