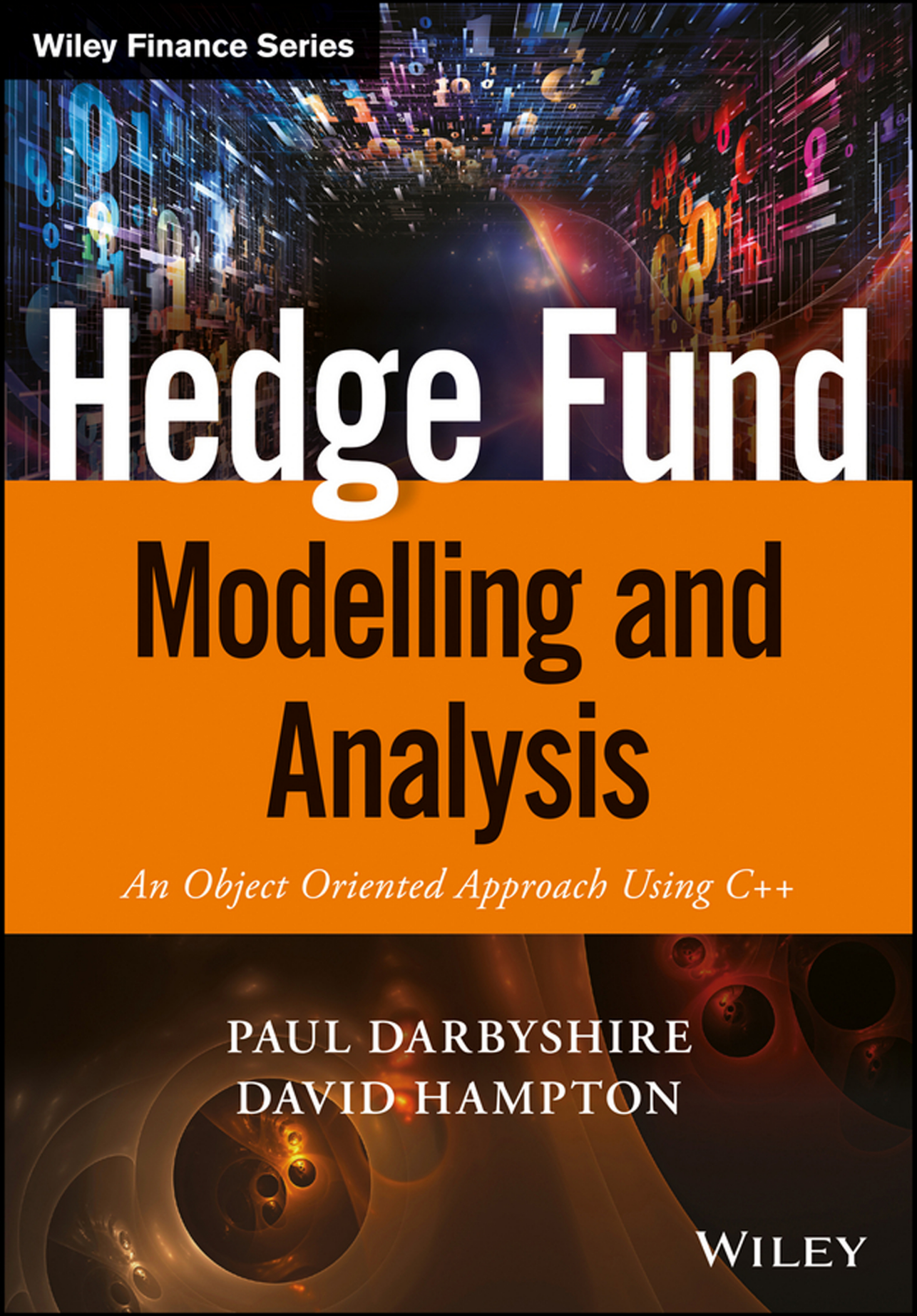


Wiley Finance Series



Hedge Fund Modelling and Analysis

An Object Oriented Approach Using C++



PAUL DARBYSHIRE
DAVID HAMPTON

WILEY

Hedge Fund Modelling and Analysis

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.

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.

Hedge Fund Modelling and Analysis

An Object Oriented Approach Using C++

PAUL DARBYSHIRE
DAVID HAMPTON

WILEY

This edition first published 2017
© 2017 Paul Darbyshire and David Hampton

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.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the publisher is not engaged in rendering professional services and neither the publisher nor the author shall be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Library of Congress Cataloging-in-Publication Data is available

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

ISBN 978-1-118-87957-3 (hbk) ISBN 978-1-118-87955-9 (ebk)
ISBN 978-1-118-87956-6 (ebk) ISBN 978-1-118-87954-2 (ebk)

Cover Design: Wiley

Cover Images: Top Image: ©iStock.com/agsandrew

Bottom Image: ©iStock.com/Storman

Set in 11/13pt Times by Aptara Inc., New Delhi, India

Printed in Great Britain by TJ International Ltd, Padstow, Cornwall, UK

Mum and Dad,

Whose love and support encourages me to achieve success.

– P.D.

For Marie-Christine, Juliette and Antoine.

– D.H.

Contents

Preface	xi
CHAPTER 1	
Essential C++	1
1.1 A Brief History of C and C++	1
1.2 A Basic C++ Program	2
1.3 Variables	4
1.3.1 Characters and Strings	5
1.3.2 Variable Declarations	8
1.3.3 Type Casting	9
1.3.4 Variable Scope	10
1.3.5 Constants	11
1.4 Operators	12
1.4.1 The Assignment Operator	12
1.4.2 Arithmetic Operators	14
1.4.3 Relational Operators	15
1.4.4 Logical Operators	16
1.4.5 Conditional Operator	17
1.5 Input and Output	18
1.6 Control Structures	21
1.6.1 Branching	21
1.6.2 Looping	25
1.6.3 The for Loop	25
1.6.4 The while Loop	27
1.6.5 The do ... while Loop	29
1.7 Arrays	30
1.8 Vectors	31
1.9 Functions	33
1.9.1 Call-by-Value vs. Call-by-Reference	36
1.9.2 Overloading Functions	39
1.10 Object Oriented Programming	41
1.10.1 Classes and Abstract Data Types	42

1.10.2	Encapsulation and Interfaces	43
1.10.3	Inheritance and Overriding Functions	44
1.10.4	Polymorphism	45
1.10.5	An Example of a Class	46
1.10.6	Getter and Setter Methods	49
1.10.7	Constructors and Destructors	52
1.10.8	A More Detailed Class Example	55
1.10.9	Implementing Inheritance	61
1.10.10	Operator Overloading	64

CHAPTER 2

The Hedge Fund Industry **71**

2.1	What are Hedge Funds?	71
2.2	The Structure of a Hedge Fund	74
2.2.1	Fund Administrators	74
2.2.2	Prime Brokers	75
2.2.3	Custodian, Auditors and Legal	76
2.3	The Global Hedge Fund Industry	77
2.3.1	North America	79
2.3.2	Europe	80
2.3.3	Asia	81
2.4	Specialist Investment Techniques	82
2.4.1	Short Selling	82
2.4.2	Leverage	83
2.4.3	Liquidity	84
2.5	Recent Developments for Hedge Funds	85
2.5.1	UCITS Hedge Funds	85
2.5.2	The European Passport	88
2.5.3	Restrictions on Short Selling	88

CHAPTER 3

Hedge Fund Data Sources **91**

3.1	Hedge Fund Databases	91
3.2	Major Hedge Fund Indices	92
3.2.1	Non-Investable and Investable Indices	92
3.2.2	Dow Jones Credit Suisse Hedge Fund Indices (www.hedgeindex.com)	94
3.2.3	Hedge Fund Research (www.hedgefundresearch.com)	100
3.2.4	FTSE Hedge (www.ftse.com)	102
3.2.5	Greenwich Alternative Investments (www.greenwichai.com)	104
3.2.6	Morningstar Alternative Investment Center (www.morningstar.com/advisor/alternative-investments.htm)	108

3.2.7	EDHEC Risk and Asset Management Research Centre (www.edhec-risk.com)	112
3.3	Database and Index Biases	113
3.3.1	Survivorship Bias	113
3.3.2	Instant History Bias	115
3.4	Benchmarking	115
3.4.1	Tracking Error	116

CHAPTER 4

	Statistical Analysis	119
4.1	The Stats Class	119
4.2	The Utils Class	120
4.3	The Import Class	123
4.4	Basic Performance Plots	127
4.4.1	Value Added Index	127
4.4.2	Histograms	130
4.5	Probability Distributions	131
4.5.1	Populations and Samples	132
4.6	Probability Density Function	133
4.7	Cumulative Distribution Function	134
4.8	The Normal Distribution	134
4.8.1	Standard Normal Distribution	136
4.9	Visual Tests for Normality	136
4.9.1	Inspection	136
4.9.2	Normal Probability Plot	137
4.10	Moments of a Distribution	138
4.10.1	Mean and Standard Deviation	138
4.10.2	Skew	141
4.10.3	Kurtosis	142
4.11	Covariance and Correlation	146
4.12	Linear Regression	158
4.12.1	Coefficient of Determination	163
4.12.2	Residual Plots	167

CHAPTER 5

	Performance Measurement	173
5.1	The PMetrics Class	173
5.2	The Intuition Behind Risk-Adjusted Returns	174
5.2.1	Risk-Adjusted Returns	182
5.3	Absolute Risk-Adjusted Return Metrics	184
5.4	The Sharpe Ratio	187
5.5	Market Models	191

5.5.1	The Information Ratio	192
5.5.2	The Treynor Ratio	197
5.5.3	Jensen's Alpha	203
5.5.4	M-Squared	205
5.6	The Minimum Acceptable Return	207
5.6.1	The Sortino Ratio	207
5.6.2	The Omega Ratio	211
CHAPTER 6		
Mean-Variance Optimisation		213
6.1	The Optimise Class	213
6.2	Mean-Variance Analysis	214
6.2.1	Portfolio Return and Variance	214
6.2.2	The Mean-Variance Optimisation Problem	229
6.2.3	The Global Minimum Variance Portfolio	244
6.2.4	Short Sale Constraints	246
CHAPTER 7		
Market Risk Management		247
7.1	The RMetrics Class	247
7.2	Value-at-Risk	248
7.3	Traditional VaR Methods	251
7.3.1	Historical Simulation	251
7.3.2	Parametric Method	254
7.3.3	Monte-Carlo Simulation	261
7.4	Modified VaR	263
7.5	Expected Shortfall	266
7.6	Extreme Value Theory	271
7.6.1	Block Maxima	272
7.6.2	Peaks Over Threshold	272
References		277
Index		279

Preface

This book is a practical introduction to modelling and analysing hedge funds using the C++ programming language. The structure of the book is as follows. Chapter 1 gives an overview of the C++ syntax in enough detail to approach the material covered in the technical chapters. Chapter 1 also introduces the concept of object oriented programming which allow us to build large and complex programs that can be broken down into smaller self-contained reusable code units known as classes. We will develop a series of classes throughout the book to tackle many of the problems encountered. Please note that this book is not intended to be an exhaustive exploration of C++ to solve problems in modelling and analysing hedge fund data. In addition, C++ is used to facilitate the solution of such problems through object oriented programming methods and various details highlighted as and when necessary.

Chapters 2 and 3 give an update of the current state of the global hedge fund industry and a detailed look at the primary data sources available to hedge fund managers and analysts. With this fundamental knowledge in place, Chapters 4–7 cover the more quantitative and theoretical material needed to effectively analyse a series of hedge fund returns and extract the relevant information required in order to make critical investment decisions.

C++ SOURCE CODE

Throughout the book there are numerous C++ source boxes (e.g., Source 2.4) typically listing the `AClass.h`, `AClass.cpp`, and `main.cpp` files and a console window showing the results of the class implementation. For example, an extract from the `Optimise` class is shown in Source P.1.

SOURCE P.1: A SAMPLE C++ SOURCE CODE

```
// Optimise.h
#pragma once;
```

```

#include "Matrix.h"
#include "Stats.h"

class Optimise: public Stats
{
public:
    Optimise() {}
    virtual ~Optimise() {}

    // Member function declarations
    Matrix PRet(const V2DD& v); // PRet()
    Matrix PVar(const V2DD& v); // PVar()
private:
    // Member variable declarations
    Matrix m_matrix; // An instance of the Matrix class
};

// Optimise.cpp
#include "Optimise.h"

Matrix Optimise::PRet(const V2DD& v)
{
    UINT n = v[0].size()-1;

    // Declare wT and R matrices
    Matrix wT = Matrix(1, n);
    Matrix R = Matrix(n, 1);

    // Transpose weights
    for (UINT i=1; i<=n; i++)
        wT(1, i) = 1 / (DBL)n; // Equal weights

    // Mean returns
    V1DD r = Mean(v, 12);

    // R matrix
    for (UINT i=1; i<=n; i++)
        R(i, 1) = r[i-1];

    return wT * R;
}

Matrix Optimise::PVar(const V2DD& v)
{

```

```
UINT n = v[0].size()-1;

// Declare w, wT and VCV matrices
Matrix w = Matrix(n, 1);
Matrix wT = Matrix(1, n);
Matrix VCV = Matrix(n, n);

    // Initialise portfolio weights
for (int i=1; i<=n; i++)
    w(i, 1) = 1 / (DBL)n; // Equal weights

// Transpose weights
for (int i=1; i<=n; i++)
    wT(1, i) = 1 / (DBL)n; // Equal weights

// Covariance matrix
V1DD cov = Cov(v);

// VCV matrix
int k = 0; // Covariance offset
for (UINT i=1; i<=n; i++)
{
    for (UINT j=1; j<=n; j++)
    {
        VCV(i, j) = cov[j+k-1];
        if(i == j)
            VCV(i, j) *= 12; // Annualise variance
    }
    k+=10;
}

return wT * VCV * w;
}

// ...
// main.cpp
// ...

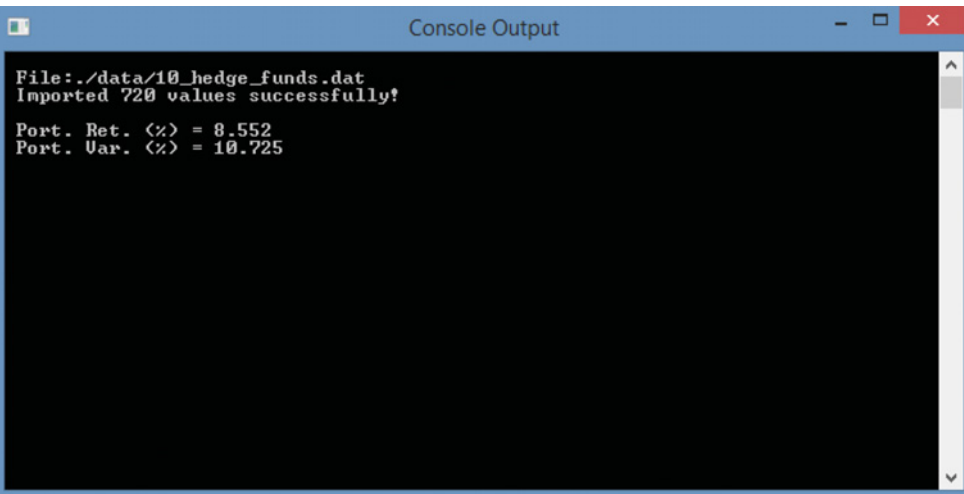
// Create class instances
Import thfs;
Optimise optimise;

// Declare and call GetData()
V2DD data = thfs.GetData("./data/10_hedge_funds.dat");
```

```
// Declare and call PRet() and PVar() member function
Matrix pret = optimise.PRet(data);
Matrix pvar = optimise.PVar(data);

// Output results
cout << "\n Port. Ret. (%) = ";
pret.Print();
cout << " Port. Var. (%) = ";
pvar.Print();

// ...
```



```
File:../data/10_hedge_funds.dat
Imported 720 values successfully!
Port. Ret. (%) = 8.552
Port. Var. (%) = 10.725
```

Comment blocks, such as:

```
// ...
// main.cpp
// ...
```

are used to omit parts of the source code (above and below) when new code is added to existing definitions or implementations. As we progress through the book we will gradually reduce unnecessary overuse of comments (//) within source listings once we feel confident we have clearly defined such routines and concepts in previous listings.

Please note that we do not give any warranty for completeness, nor do we guarantee that the code is error free. Any damage or loss incurred in the application of the C++ source code, algorithms and classes discussed in the book are entirely the

TABLE P.1 10 Hypothetical Hedge Funds

Hedge Fund	Abbreviation
Commodity Trading Advisor	CTA1, CTA2, CTA3
Long Short Equity	LS1, LS2, LS3
Global Macro	GM1, GM2
Market Neutral	MN1, MN2

reader's responsibility. If you notice any errors in the C++ source code, algorithms or classes, or you wish to submit some new method as a C++ function, algorithm, class, model or some improvement of the method illustrated in the book, you are very welcome.

HYPOTHETICAL HEDGE FUND DATA

Throughout the book there is constant reference to many monthly hedge fund return series. The 10 hedge funds are all *hypothetical* and have been simulated by the authors as a unique data set for demonstration purposes only. The techniques and models used in the book can therefore be tested on the hypothetical data before being applied to real-life situations by the reader. The hypothetical data is nonetheless close to what would be expected in reality. The 10 funds are a mixture of several major hedge fund strategies i.e. *Commodity Trading Advisor* (CTA), *Long/Short Equity* (LS), *Global Macro* (GM) and *Market Neutral* (MN) strategies as described in Table P.1.

All data files used throughout the book are identified in italics e.g. *10_hedge_funds.dat*.

BOOK WEBSITE

The official website for the book is located at: www.darbyshirehampton.com

The website provides free downloads to all of the hypothetical data, C++ programs and classes, as well as many other useful resources.

The authors can be contacted on any matter relating to the book, or in a professional capacity, at the following email addresses:

Paul Darbyshire: pd@darbyshirehampton.com

David Hampton: dh@darbyshirehampton.com

Essential C++

This chapter covers the fundamental requirements necessary to allow the reader to get up and running building quantitative models using the C++ programming language. This introduction is in no way intended to be an in-depth treatment of the C++ programming language but more an overview of the basics required to build your own efficient and adaptable programs. Once the key concepts have been developed, object-oriented principles are introduced and many of the advantages of building quantitative systems using such programming approaches are outlined. It is assumed that the reader will have some prerequisite knowledge of a low-level programming language and the necessary computation skills to effectively grasp and apply the material presented here.

1.1 A BRIEF HISTORY OF C AND C++

C is a *procedural*¹ programming language developed at Bell Laboratories between 1969 and 1973 for the UNIX operating system. Early versions of C were known as K&R C after the publication of the book *The C Programming Language* written by Brian Kernighan and Dennis Ritchie in 1978. However, as the language developed and became more standardised, a version known as ANSI² C became more prominent. Although C is no longer the choice of many developers, there is still a huge amount of *legacy* software coded in it that is actively maintained. Indeed, C has greatly influenced other programming languages, in particular C++ which began purely as an extension of C.

¹ *Procedural* programming is a form of *imperative programming* in which a program is built from one or more procedures i.e. subroutines or functions.

² Founded in 1918, the *American National Standards Institute* (ANSI) is a private, non-profit membership organisation that facilitates the development of *American National Standards* (ANS) by accrediting the procedures of the *Standards Developing Organizations* (SDOs). These groups work cooperatively to develop voluntary national consensus standards.

Often described as a *superset* of the C language, C++ uses an entirely different set of programming concepts designed around the *Object-Oriented Programming* (OOP) paradigm. Solving a computer problem with OOP involves the design of so-called *classes* that are abstractions of physical objects containing the state, members, capabilities and methods of the object. C++ was initially developed by Bjarne Stroustrup in 1979 whilst at Bell Laboratories as an enhancement to C; originally known as C with Classes. The language was renamed C++ in the early 80s and by 1998, C++ was standardised as ANSI/ISO³ C++. During this time several new features were added to the language, including virtual functions, operator overloading, multiple inheritance and exception handling. The ANSI/ISO standard is based on two main components: the *core language* and the *C++ Standard Library* that incorporates the C Standard Library with a number of modifications optimised for use with the C++ language. The C++ Standard Library also includes most of the *Standard Template Library* (STL); a set of tools, such as *containers* and *iterators* that provide array-like functionality, as well as *algorithms* designed specifically for sorting and searching tasks. C++11 is the most recent *complete* overhaul of the C++ programming language approved by ANSI/ISO on 12 August 2011, replacing C++03, and superseded by C++14 on 18 August 2014. The naming convention follows the tradition of naming language versions by the year of the specification's publication, although it was formerly known as C++0x to take into account many publication delays. C++14 is the informal name for the most recent revision of the C++ ANSI/ISO standard, intended to be a small extension over C++11, featuring mainly bug fixes and small syntax improvements.

1.2 A BASIC C++ PROGRAM

Without doubt the best method of learning a programming language is to actually start by writing and analysing programs. Source 1.1 implements a basic C++ program that simply outputs a string of text, once the program has been compiled and executed, to the console window. Although the program looks very simple it nevertheless contains many of the fundamental components that every C++ program generally requires.

SOURCE 1.1: A BASIC C++ PROGRAM

```
// main.cpp
#include <windows.h>
#include <iostream>
```

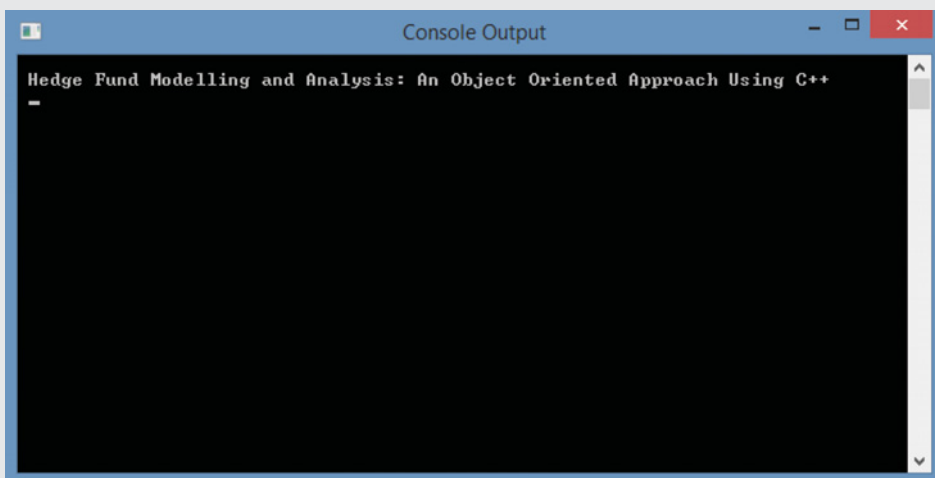
³ The *International Organisation for Standardisation* (ISO) is an international standard-setting body made up of representatives from a range of *National Standards Organisations* (NSOs).

```
using std::cout;
using std::cin;

int main()
{
    SetConsoleTitle(L"Console Output"); // Set title of console
    window

    cout << "\n " << "Hedge Fund Modelling and Analysis: An Object
    Oriented Approach Using C++";

    cin.get(); // Pause console window
    return 0; // Return null integer and exit
}
```



Statements beginning with a hash symbol (#) indicate *directives* to the *preprocessor* that initialise when the compiler is first invoked, in this case, to inform the compiler that certain functions from the C++ Standard Library must be included. `#include <windows.h>` gives the program access to certain functions in the library, such as `SetConsoleTitle()` whilst `#include <iostream>` enables console input and output (I/O). Typical objects in the `iostream` library include `cin` and `cout` which are explicitly included through the `using` statement at the top of the program. Writing `using std::cout` at the top of the program avoids the need to keep retyping `std` through the *scope resolution operator* (`::`) every time `cout` is used. For example, if we had not specified `using std::cout` we would have to explicitly write `std` in front of each usage throughout the program, that is:

```
std::cout << "\n " << "Hedge Fund Modelling and Analysis: An Object  
Oriented Approach Using C++";  
std::cin.get();
```

Although in this case there are only two occasions where we need `std`, you can imagine how this could quickly clog up code for very large programs. Note also that all C++ statements must end with a semi-colon (;).

A commonly identified problem with the C language is the issue of running out of names for definitions and functions when programs reach very large sizes eventually resulting in name clashes. Standard C++ has a mechanism to prevent such a clash through the use of the `namespace` keyword. Each set of C++ definitions in a library or program is *wrapped* into a namespace, and if some other definition has an identical name, but is in a different namespace, then there is *no* conflict. All Standard C++ libraries are wrapped in a single namespace called `std` and invoked with the `using` keyword:

```
using namespace std;
```

Whether to use `using namespace std` or explicitly state their use through `using std::cout`, for example, is purely a preference of programming style. The main reason we do not invoke `using namespace std` in our programs is that this leaves us the opportunity of defining our own namespaces if we wish and it is generally good practice to have only one namespace invocation in each program.

The `main()` function is the point at which all C++ programs start their execution even if there are several other functions declared in the same program. For this reason, it is an essential requirement that all C++ programs have a `main()` function within the body at some point in the program. Once the text is output to the console window, `cin.get()` is used to cause the program to pause so that the user can read the output and then close and exit the window by pressing any key. Technically, in C or C++ the `main()` function must return a value because it is declared as `int` i.e. the main function should return an integer data type. The `int` value that `main()` returns is usually the value that will be passed back to the operating system; in this case it is 0 i.e. `return 0` which indicates that the program ran successfully. It is not necessary to state `return 0` explicitly, because the compiler invokes this automatically when `main()` terminates, but it is good practice to include a return type for all functions (including `main()`).

1.3 VARIABLES

A variable is a name associated with a portion of memory used to *store* and *manipulate* the data associated with that variable. The compiler sets aside a specific amount of memory space to store the data *assigned* to the variable and associates the variable name with that memory *address*. As the name implies, variables can be changed within a program as and when required. When new data is assigned to the same variable, the old data is *overwritten* and restored in the same memory address. The data stored in a

TABLE 1.1 Reserved C++ keywords

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

variable is only *temporary* and only exists as long as the variable itself exists (defined by the *scope* of the variable). If the data stored in a variable is required beyond its existence then it must be written to a *permanent* storage device, such as a disk or file.

A variable name can be any length and composed of lower and upper case letters, numbers and the underscore (`_`) character, but keep in mind that variables are *case-sensitive*. In practice, a programmer will usually develop their own variable naming convention but bear in mind that C++ reserves certain keywords for variable names so try not to clash with these. Table 1.1 shows a list of reserved C++ keywords.

There are several built-in *data types* provided by C++ along with specific *type modifiers* to further quantify the data. A complete list of all the data types and their associated modifiers are described in Table 1.2.

In Table 1.2, other than `char` (which has a size of exactly one byte), none of the fundamental types has a standard size (only a minimum size, at most). This does not mean that these types are of an undetermined size, but that there is no *standard size* across all compilers and machines; each compiler implementation can specify the sizes that best fit the architecture where the program is going to be executing. This rather generic size specification of data types allows the C++ language a lot of flexibility in adapting to work optimally on all kinds of platforms, both present and future.

1.3.1 Characters and Strings

When using the `char` data type, we use single quotes, for example:

```
char Stock = 'MSFT';
```

Certain characters, such as single (`'`) and double (`"`) quotes have special meaning in C++ and have to be treated with care. In addition, C++ reserves special characters for formatting text and other processing tasks known as *character escape sequences* (or *backslash character constants*) as shown in Table 1.3.

A more versatile data type than `char` is `string` which can be a combination of characters, numbers, spaces and symbols of any length. C++ does not have a built-in data type to hold strings instead it is defined in the C++ Standard Library through the inclusion of the header file `<string>`. An example of using `string` variables is shown in Source 1.2.

TABLE 1.2 Common C++ data types

Name	Description	Size (Bytes)	Range
char	Character	1	-128 to 127
unsigned char		1	0 to 255 (ASCII characters)
signed char		1	-128 to 127 (ASCII characters)
int	Integer number	4	-2,147,483,648 to 2,147,483,647
unsigned int		4	0 to 4,294,967,295
signed int		4	-2,147,483,648 to 2,147,483,647
short int		2	-32,768 to 32,767
unsigned short int		2	0 to 65,535
signed short int		2	-32,768 to 32,767
long int		4	Same as int
unsigned long int		4	Same as unsigned int
signed long int		4	Same as signed int
float	Floating point number	4	3.4E-38 to 3.4E+38
double	Double precision floating point number	8	1.7E-308 to 1.7E+308
long double		10	3.4E-4932 to 1.1E+4932
bool	Boolean value	1	True or False
string	As required		Any length
wchar_t	Wide character	2	0 to 65,535

TABLE 1.3 Character escape sequences

Sequence	Output
\n	New line
\t	Tab
\b	Back space
\?	Question mark
\f	Page feed
\a	Alert (beep)
\\	Backslash
\'	Single quote
\"	Double quote

SOURCE 1.2: STRING VARIABLES

```
// main.cpp
#include <windows.h>
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::string;

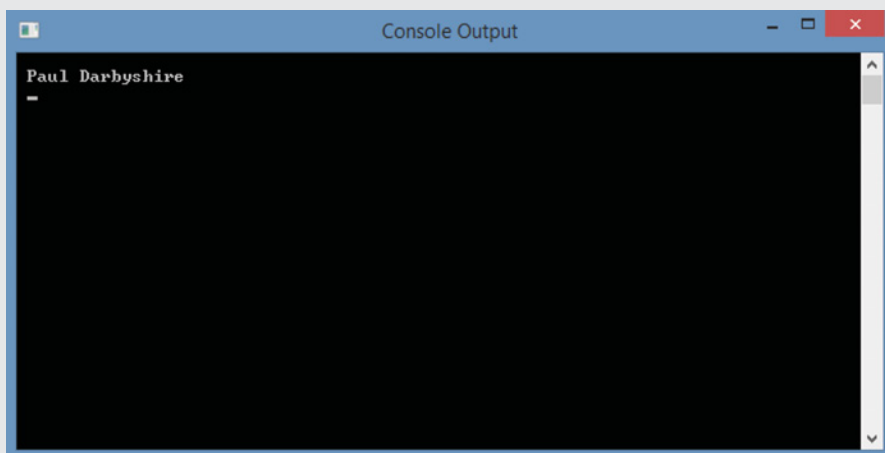
int main()
{
    SetConsoleTitle(L"Console Output"); // Set title of console
    window

    //declare two string variables
    string strFirstName = "Paul";
    string strLastName = "Darbyshire";

    //concatenate the two strings
    string strFullName = strFirstName + " " + strLastName;

    cout << "\n " << strFullName;

    cin.get(); // Pause console window
    return 0; // Return null integer and exit
}
```



In Source 1.2, two `string` variables are declared and initialised and then joined together to form another string. The `(+)` symbol is used for joining (or *concatenating*) two variables together, and in this context the `(+)` symbol is often referred to as the *concatenation operator*.

1.3.2 Variable Declarations

Before a variable can be used in a program it must first be *declared* as shown in Source 1.3. Declaring the variable and its data type allows the compiler to set aside the appropriate amount of memory for storage and subsequent manipulation.

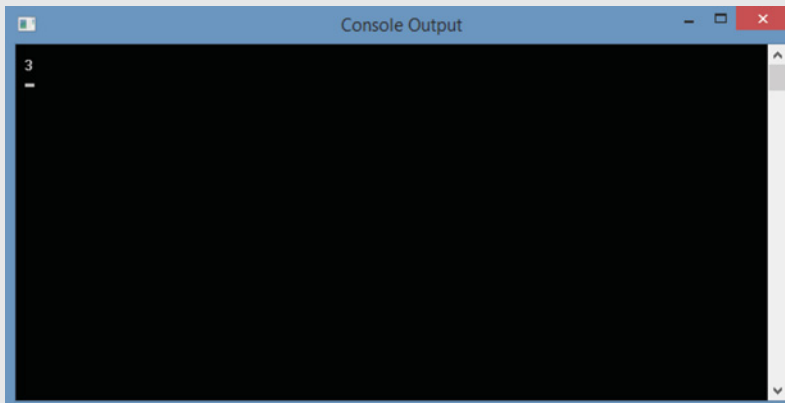
SOURCE 1.3: DECLARING VARIABLES

```
// ...

// Declare variables
int x, y;
int result;
// Assign values
x = 4;
y = 2;
x = x + 1;
//Do something
result = x - y;

cout << "\n " << result << "\n ";

// ...
```



It is possible to declare more than one variable of the *same* type in the same declaration statement. It is also possible to assign initial values to variables whilst they are being declared through the process of *initialisation*, for example:

```
int x, y = 4, z = 3;
```

There is another useful method of initialising a variable known as *constructor initialisation*:

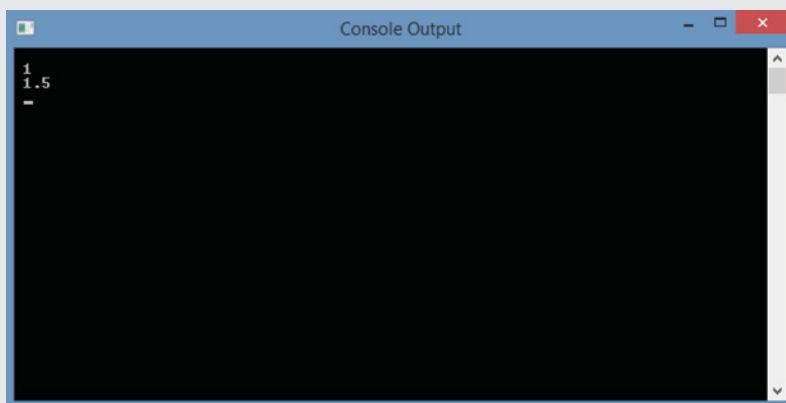
```
int x(0);
```

1.3.3 Type Casting

One way to force an expression to produce a result that is of a different type to the variables declared in the expression is to use a construct called `cast` (i.e. *type casting*). Source 1.4 shows an example of declaring two variables as `int` and dividing them to produce an `int` and `double` division through type casting.

SOURCE 1.4: TYPE CASTING

```
// ...  
  
int a = 6, b = 4;  
  
cout << "\n " << a/b << "\n"; // Integer division  
cout << " " << (double)a/b << "\n "; // Type casting to double  
division  
  
// ...
```



Note that type casting will not change the type of the variables from integer only the type of the result to double.

1.3.4 Variable Scope

A variable can have either *global* (i.e. *public*) or *local* (i.e. *private*) scope depending on where it is declared within the program. Any variables declared with global scope should be prefixed with the keyword `const`. An example is shown in Source 1.5.

SOURCE 1.5: VARIABLE SCOPE

```
// ...

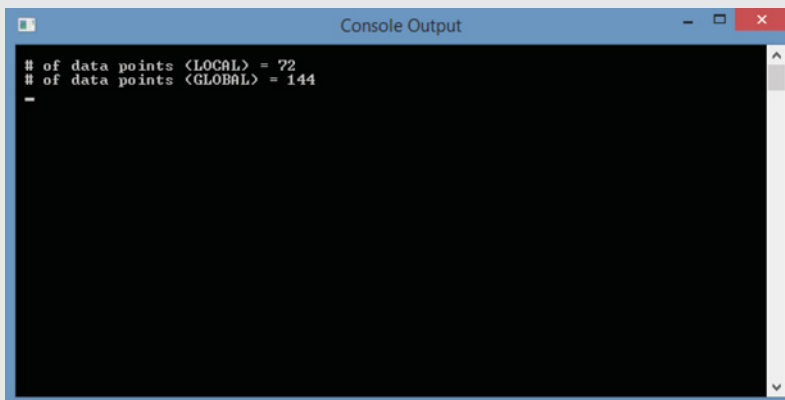
// GLOBAL variable
int globalN = 144;

int main()
{
    SetConsoleTitle(L"Console Output"); // Set title of console
    window

    // LOCAL variable
    int localN = 72;

    cout << "\n " << "# of data points (LOCAL) = " << localN;
    cout << "\n " << "# of data points (GLOBAL) = " << globalN;

    cin.get(); // Pause console window
    return 0; // Return null integer and exit
}
```



In Source 1.5, you can see that the variable `globalN` has been declared globally and initialised to the value 144. Global variables can be accessed from anywhere in the program once they have been declared. Local variables, on the other hand, such as `localN` can only be used within the block enclosed by the braces (`{}`) in which it is declared.

1.3.5 Constants

Constants are fixed values assigned to variables that cannot be changed once they have been declared and initialised. We have already used *literal constants* when a variable was declared and initialised in Source 1.2:

```
string FirstName = "Paul";
```

Or, as in Source 1.5:

```
int localN = 72;
```

With *symbolic constants* the `const` keyword is used in front of the declaration and initialisation, for example:

```
const double Volatility = 0.18;
```

Enumerated constants are an alternative way of creating a series of integer constants. Suppose you wanted to assign an integer value of 0 to 6 to the days of the week starting at Sunday. This could be achieved using a list of symbolic constants written as:

```
const int Sun = 0;  
const int Mon = 1;  
const int Tue = 2; etc.
```

However, with enumeration it is possible to write:

```
enum WeekDays  
{  
    Sun,  
    Mon,  
    Tue,  
    Wed,  
    Thu,  
    Fri,  
    Sat  
};
```

If each week day is not explicitly initialised, they are *automatically* assigned the values 0, 1, 2, 3, etc., starting with the variable `Sun`. Note that the default value starts at 0 and not 1. Alternatively, it is possible to initialise one or more of the variables to any integer value, for example:

```
enum WeekDays
{
    Sun = 10,
    Mon,
    Tue,
    Wed = 6,
    Thu,
    Fri,
    Sat
};
```

Variables that are not explicitly initialised are given initial values counting upwards from the preceding initialised variable i.e. `Sun = 10`, `Mon = 11`, `Tue = 6`, `Wed = 7`, `Thu = 8` and so on.

1.4 OPERATORS

Operators are used to perform a specific operation on a set of operands in an expression. Operators can be of two types:

Unary – take only one argument and

Binary – take two arguments.

1.4.1 The Assignment Operator

The *assignment operator* simply assigns a value to a variable, for example:

```
x = 4;
```

The statement above assigns to the variable `x` the value 4. Note that the assignment operator always reads from *right* -> *left*, and never the other way around. The following statement is valid in C++:

```
x = y = z = 3;
```

In this statement, the value 3 is assigned to all three variables `x`, `y` and `z`. Expressions that are evaluated within the assignment operator, such as:

```
x = x + 1;
```