Francis Glassborow

You Can Program in C++

A Programmer's Introduction

John Wiley & Sons, Ltd

You Can Program in C++

A Programmer's Introduction

Francis Glassborow

You Can Program in C++

A Programmer's Introduction

John Wiley & Sons, Ltd

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Glassborow, Francis.
You can program in C++ : a programmer's introduction / Francis Glassborow.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-470-01468-4 (pbk. : alk. paper)
ISBN-10: 0-470-01468-7 (pbk. : alk. paper)
1. C++ (Computer program language) I. Title.

QA76.73.C153G59 2006
005.13′3 – dc22

2005026864

British Library Cataloguing in Publication Data

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

ISBN-13: 978-0-470-01468-4 (PB)
ISBN-10: 0-470-01468-7 (PB)

Typeset in 10/11 JoannaMT by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Antony Rowe Ltd, Chippenham, Wiltshire
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Dedication

This book is dedicated to the numerous people who have helped me to master the art of writing simple programs in C++. I appreciate their gentle correction of programming so that it has reached a standard that I feel able to share with others.

Contents

Preface

My previous book, *You Can Do It!*, was written for the complete newcomer to programming. I made no assumptions about the reader's prior knowledge and skills other than that they were capable of using a Microsoft Windows–based machine at the general level of accessing the Internet. It should not matter to such people what language is used for their practical experience of programming. I chose C++ because I felt certain that it was well up to the task, as long as I used a carefully chosen subset and augmented the Standard Library with a library of my own design that would support writing programs that newcomers would find interesting. The priority of that book was learning sound programming.

This book is intended for a very different readership: you should already be comfortable with the basics of programming. Exactly how you have acquired those basics will result in different expectations and problems with learning C++. One of the delights of C++ is its ability to handle the programming paradigms of most of the principal language groups. If your first language is Lisp and you are fluent in expressing problems in that language, then C++ is going to cause you a lot of mental readjustment, but most other languages will provide a good basis for moving to C++, as long as you have an open mind about how the solutions to problems should be expressed in source code.

I do not intend to provide comprehensive coverage of the whole of C++: it is far too big a language to do that. I am not going to attempt to show you all the ways in which C++ can be used: C++ is far too rich a language to attempt that in a single book. Indeed, I doubt that any single author knows enough to provide adequate coverage of all the ways C++ can be used.

My aim is to provide my readers with a sound introduction to a reasonably large working subset of C++. Along the way, I will demonstrate how C++ can be used to handle a variety of programming problems.

You will get as much from this book as you put into reading it, or, more correctly, studying it. I do not believe in trivial, make-work exercises. You should be able to provide yourself with those without any help from me. That means that the exercises in this book, along with the experiments and actively trying the code in the body of the text is part and parcel of reading this book successfully.

If you want to try C++ and have a basic knowledge of programming fundamentals, this book was written for you. I hope you enjoy the journey and feel motivated at the end to continue onwards, because C++ is the most challenging programming language available. It does not seek to constrain what you can do or how you do it. That is one of the ways in which it differs from all the other popular computer-programming languages.

If you can master C++ you will be mistress of programming and able, should the need arise, to adapt to other languages quickly.

Francis Glassborow
July 29th, 2005

Acknowledgements

Like any book, many people have contributed to this one in addition to the named author. Some, such as the staff of Wiley & Sons Ltd, do so because it is their job to do so. However I appreciate the extra effort that each of these people have put in that distinguishes them from the mere time-server or wage slave.

Then there are the countless numbers of people who have added their ideas, sometimes unwittingly, to mine. These include many members of ACCU as well as all those experts who attend meetings concerned with Standardizing C++. Bjarne Stroustrup must take pride of place among these because he is not only the original creator of C++ but has also spent many hours over the last fifteen years patiently helping me to understand his creation.

However there are three people without whom this book would not be what it is. Garry Lancaster was the original implementor of the Playpen Library for Windows. Jean-Marc Bouquet adapted Garry's implementation for machines that use X11 (initially Linux, but also other Unix and MacOS X machines). Finally, Mick Brooks who has spent many hours of his valuable time checking that my text worked on a Linux machine. Each of those people contributed their efforts free, to the benefit of the C++ community.

Elsewhere, I give credit to those responsible for the two IDEs that are shipped on the CD for this book. However, I hope you will show your appreciation for the work of Parinya Thipchart `(thipchart@gmail.com)` who is the designer and implementor of MinGW Developer Studio. jGrasp, G++ and MinGW are examples of the excellent tools that are made available free by their developers. MinGW Developer Studio is the work of a single person and is distributed as user-supported software. In simple terms, it means that Parinya trusts users to make a contribution based on their use of the product and their personal circumstances.

Introduction

If you are reading this, you are most likely to be trying to decide whether this is the book that will help you learn C++. To make your choice, you need to know what this book sets out to do and how it is different from the many other alternatives that litter booksellers' shelves. I am not going to claim that this book is uniquely better than all the alternatives, but I do claim that it is distinctly different as well as being technically accurate.

The first question is: ''Can you already program in some computing language?'' If your answer to that question is no, then this book is almost certainly not the one for you to start with. You will need some other book such as my own *You Can Do It!* [Glassborow 2004]. However, if you already know about such things as loops, decision statements, and functions, then read on.

I make as few assumptions as I can about what you know about programming. In particular, I do not assume that you know anything about the C programming language. Indeed, if you already know C, be prepared to learn to do things differently. Though the name 'C++' is supposed to suggest the next step beyond C, and almost all of C is also C++, good C is often not good C++. The languages share a common core, but the differences lead to very different programming styles.

I will introduce you to programming in C++ using all the modern idioms and tools that have evolved over the past two decades. For that reason, this book introduces C++ by using high-level features and only covers low-level features as and when they become necessary for further progress.

The purpose of this book is to give you a firm grasp of the foundations of Standard C++. To gain understanding, you need to write programs. So that you can try more adventurous programs, I have supplemented the Standard C++ library with a library of my own that supplies four basic extras:

- A few simple utility functions to make it easier to write correct programs right from the start. They are written in pure Standard C++ and so are portable to all Standard C++ implementations.
- A very primitive graphics window that can be managed from a pure console program (a program that runs as pure text). This works identically on Windows- and Linux-based PCs (and also on OS X versions of the Apple Mac).
- A one-button mouse (i.e. mouse support that treats all mouse buttons as equivalent). This gives the major benefits of having a mouse without the complexity of having to deal with multiple buttons, wheels, etc.
- Direct reading of the keyboard, so that your programs can directly observe the keys you press on a keyboard. That little feature dramatically extends the kinds of program you can write.

Note that I have provided these four elements to make the process of learning C++ a more enjoyable one, during which you can write a much wider range of programs than you would otherwise have been able to. (If you want to, you could even write one of the classic console games such as Space Invaders.) I avoid

explicit use of my library throughout most of the book, but you should feel free to use it both to enhance your solutions to the provided exercises and to add other exercises for yourself.

In addition to my own library of tools, this book comes with a full C++ compiler (MinGW) and a choice of two IDEs. The first, MinGW Developer Studio, is a user-supported product. If this is a strange term to you, it means that if you like the product and wish to continue using it, you are invited and encouraged to make a donation (whatever you think it is worth to you) to the writer of the product. It is not shareware, because there is no restrictive licence limiting how much use you can legitimately make without paying a fixed registration fee. Nor is it freeware, in which you are never expected to pay anyone. (MinGW itself is an example of an outstanding freeware product.) So please show your appreciation of a well-constructed tool by going to the product's website and making a contribution.

The second choice of IDE is a more complicated but highly portable one from Auburn University called JGrasp™. It is written in Java, so you will need to acquire a suitable Java Runtime Environment. This IDE was written explicitly for the purposes of teaching and learning. It has a number of outstanding features to help newcomers explore the code they write. It is free to individuals but must not be distributed as part of a commercial product. The licence under which it has been placed on the CD is one specifically for educational books. JGrasp accepts many C++ compilers, including MinGW, as plug-ins.

These IDEs run on both Microsoft Windows and Linux systems. Indeed, JGrasp runs on any system that supports Java, but for the purposes of this book you are limited to systems for which my library has been implemented. At the time of writing, that includes all versions of Windows post 95 and Linux (and, I am told, OS X on an Apple Mac).

The main text of this book will assume that you are using MinGW Developer Studio with MinGW as the compiler. The tools that are available free and supplied on the CD are full-weight professional tools that are often used by professional programmers. You may wish to use some other toolset, but you will need to check the book's website (`http://www.spellen.org/youcandoit/`) to see whether I have supplied a copy of my library for that set of tools. Unfortunately, one of the restrictions of C++ is that library distributions are often specific to compilers and have to be recompiled if used with a different compiler.

This book is also designed for you to use as a reference. For example, when I introduce the built-in types of C++, the study text will use only a subset of the available types. At the end of that chapter, you will find a complete summary of all the built-in types and their derivatives. That provides a single point of reference close to the point where you first learn about the C++ type system. When you first study that chapter, you will probably want to skip the reference part, but at a later stage you may want to read deeper into a subject, or look something up easily.

To summarize: this book teaches you to program in Standard C++ using a modern idiomatic style. To make the process of learning more enjoyable, I have provided some extensions to the C++ library that are, at a minimum, portable between Microsoft Windows (98 and after) and Linux. The book comes with everything you need to read and study it apart from a computer and operating system – those you must provide for yourself. Apart from the extensions (graphics, sound, mouse, and direct keyboard read), everything else in this book is portable to any computer with a Standard C++ compiler installed.

Studying C++

Sometimes circumstances compel us to acquire a new skill by ourselves. It is often much harder that way and sometimes blocks our progress. If you want to learn to play a musical instrument, you need to practice by yourself, but it is also essential to spend time playing with someone else. If you do not, you will acquire bad habits that will very seriously hinder your future progress towards making music with others. To a lesser extent, the same applies in learning to program in a computer language. It will be easy to force the idioms of your first language onto the new one that you are studying (C++). Without the discipline of working with others, this weakness in your use of C++ can go unchecked. This is particularly true if your first language is a close relation of C++ (such as C, C#, or Java). It is not enough to learn the syntax of a language: you also need to learn how that syntax is intended to be used.

When I was at school, I learned the Japanese game of Go from a book. I taught a couple of fellow pupils the rules and played on a homemade board. Many years later, I met someone who had learned to play from an experienced player. Our first few games were interesting for both of us, because my understanding of tactics and strategy was so completely haphazard. There were places where my play was relatively sound and others where I would make a novice mistake. Do not let your C++ become like that.

Ideally, find both a study partner and a mentor. The former will provide an alternative view on the language; the latter will help you to develop a sound understanding. Make sure that you know how things work by checking with someone who already knows. Be wary of the self-proclaimed local expert; like my early Go knowledge, their knowledge may be very patchy. Good places to visit to read and ask questions are:

```
comp.lang.c++.moderated
alt.comp.lang.learn.c-c++
```

Those Usenet newsgroups have resident experts who know more about C++ than all but a tiny minority. There you will find genuine experts. Make use of them.

Learning C++ is an interactive exercise. You cannot learn C++ without actually writing programs. You already know that from whichever other languages you have learned, but I have repeated it here by way of emphasis. If you skip the practice work presented in this book, you will be selling yourself short. You may finish reading the book more quickly, but your C++ will be the poorer for it. This is not a book about learning C++ in some set period; you will need to take as long as it takes in order to master each bit. Sometimes it may be worth moving past a sticking point and coming back to it a little later. But if you permanently skip something because you do not understand it or you think it is not much use, it will come back to haunt you.

Using This Book

Please use this book in conjunction with a computer! Try the code and do the exercises. I hope the exercises are helpful – I did not provide them to fill an expected slot in a textbook. It is sometimes a temptation to look at an exercise and decide that you can see how to do it and move on without actually doing the work. However, practice makes perfect, and it will help your acquisition of fluency with C++. Quite often, doing an exercise will reveal subtleties that you miss at first glance.

At other times, you might get this great idea either for a new program or for developing one from the book. In such cases give in to temptation and spend some time following the idea. If you write up the problem and email it to me (initially via the submissions form on the book's website), I will post it on the site to inspire others.

You will find some source code on the CD, both that for my library and code from elsewhere. Some is good; some is not but is included because I used it as a stepping-stone. For example, the code that supports the Portable Network Graphics part of my library was written in very old C. Its greatest merit is that it works and works correctly, but it is far from the kind of clear, maintainable code I would encourage you to write. Other source code on the CD requires a great deal of knowledge of the specifics of Windows or Linux. It will be far from clear even to programmers who have a good background in the system concerned. Even my own code on the CD is a less-than-perfect example of good code. As time goes by and other experts comment, it improves, but all code is subject to, or at least should be subject to, progressive improvement. One test to apply to code is whether it is structured so that such progressive improvement is relatively easy.

To understand the code in `playpen.cpp`, you will need considerable experience of both C++ and graphical-user-interface code for the OS you are using. That is the file that had to be reworked to provide my library for Linux and other systems that support the X Window System (not to be confused with Microsoft Windows, which is something very different).

You can obtain other source code from the site for my books (`http://www.spellen.org/youcandoit/`). You should also visit that site for anything else that might have missed this book's CD. Additionally, you can contact me through that site, or directly by e-mailing `ycpcpp@robinton.demon.co.uk`.

If you choose to e-mail me directly, please make sure that you include a subject line, because I usually assume that e-mails without subjects are spam.

This book is not 'C++ in *x* time units'. You must set your own pace. If it takes you only six months to grasp all that I provide, you are doing well. However, long before you finish, you will have enough knowledge of C++ to do many useful things. On the other hand, even after you have finished, there will be much more to learn. Every one of the leading C++ experts tells me the same thing: they never stop learning and discovering new aspects of programming in general and programming with C++ in particular. Perhaps that continued process of learning and discovery is what has made them into world-class experts.

A Comment on Comments

Some people feel that source code should be extensively commented, and you will find books where the displayed source code is densely populated with comments. I believe that the best documentation of code is the code itself. Comments should be used to express those things that cannot be easily expressed in source code. An introduction to C++ needs to provide a lot of explanation, but comments are not, in my opinion, the place for that.

There are useful things that are not expressible in source code. For example, every file of source code should include both the date of its origin and the name of its author. Somewhere there needs to be documentation that describes what a piece of source code does. However, if there is a comment on every line of code, that coding style has a serious problem. A few well-placed comments can be very useful. However, if those comments are surrounded by repetitious ones that add no value, the valuable ones may be missed.

Overview of C++

I very nearly numbered this as chapter -1 on the basis that it precedes your actual learning to use C++. You can skip this chapter if you are anxious to get on with the process of learning. You can always come back to it later. However, I think that a quick read through, even the parts that address languages that you do not know will help you with your study of C++.

What is in a Name

You may already know that C++ (pronounced 'see plus plus') is so named because it was designed by Bjarne Stroustrup as a successor to an earlier (and still widely used) language called C. In C, ++ means 'increment' and, in mathematical terms, to increment means to obtain something's successor. Therefore, you could interpret the name as meaning 'the successor of C'. Like the concept of a successor in mathematics, that does not imply replacement. If you already know C, you need to recognize that C++ is a new and different language, even though much C source code will compile as C++. Usually the result of a successful compilation of C source code with a C++ compiler will be a program that behaves exactly like the one produced by a C compiler. However, that is not always true.

In the early 1980s, Bjarne Stroustrup designed an extension to C that he called 'C with classes'. If you are interested in the history of how that personal tool grew up to become the most widely used programming language in the world and one that has fired the imaginations of many people you will have to look elsewhere. (A good place to start would be with *The Design and Evolution of C++* [Stroustrup 1994].) This book is about programming in C++ as the ISO/IEC 14882:2003 Standard defines it, that is, Standard C++ as it was specified in 2003 (which is the first official standard, with various corrections that were made between 1998 and 2003).

What is in C++

C++ is one of the most widely used programming languages in the world. It is also one of the largest programming languages ever designed. Bjarne Stroustrup specified that one of the design criteria of the language is that there should be no room for a lower-level language between C++ and native machine code. Very few programmers ever use C++'s lowest level, and many do not even know that it has an `asm` keyword, which allows support for writing code in assembler.

The incorporation of C into C++ was an important design decision. On the positive side, it made it easy for C programmers to transfer to C++. Having made the transfer they could, at least in theory, incrementally

add to their C++ skills and understanding. On the negative side, it has tied C++ to a number of features of C's design that experience has shown to be, at best, problematical. It has also caused problems to many who have moved from C to C++, because they have made the transition from a C to a C++ compiler without actually making the mental transition from C to C++. They are still C programmers at heart. There is nothing wrong with that, but it does provide a roadblock to their becoming fluent C++ programmers. If you are a C programmer you may find studying modern C++ tougher than you would if your first language were something else.

At the high end of C++ we find tools that allow innovators to do metaprogramming, that is, source code that generates source code. We will not be exploring that in this book, but it is worth noting that in learning C++ you are learning a language that supports the most innovative development of programming currently around.

In between assembler support and support for metaprogramming, C++ provides tools for procedural programming, object-based programming, object-oriented programming (I will explain the difference later when you know enough C++ to appreciate the differences), and generic programming. With care, you can even do some functional programming.

Alongside the raw power of the core of the C++ language, the Standard C++ Library supports a wide range of things that programmers commonly want to do. We have learnt a great deal over the last few years, and were we to start writing a library today we might produce a substantially different one. However, what we have is better than anything provided previously in any widely used programming language. In addition, much of the library has been designed for extension: it is designed so that new components can easily be added and work correctly with standard components. On the other hand the Standard Library currently lacks many of the components that users of more recent languages such as Java, C#, and Python have come to expect.

It is both one of the strengths and one of the weaknesses of C++ that it does not dictate a methodology or paradigm. When people first learn C++ this can be a problem, because the range of choice requires understanding of the implications of those choices. If the newcomer already knows another programming language, they will naturally try to discover how to write their first language in C++ terms. They will think in their first language and try to translate into C++. Such is the range of C++ that they can often get a close approximation, but that usually does not lead to good C++.

C++ can be viewed as everyone's second language. Mastery of C++ requires that you leave behind the crutch of your first language. That is hard and you will make many mistakes along the way. However, the result will be that you are a much better programmer both in C++ and in any other programming language you already know or choose to learn later.

C++ has a wide range of operators. Most of them can be extended to include user-defined types. With the potential for redefinition comes the responsibility to use such a facility wisely. The intention was that it should be possible to add types such as complex numbers, matrices, quaternions, etc. and provide the operators that a domain specialist would expect and find intuitive. Unfortunately, some programmers take the availability of a mechanism as a challenge to find creative ways of using it. The result is that their code becomes ever more obscure.

C++ is a living language. By this I mean two things. The first is that the very best users continue to develop new idioms and other ways to use it. The entire growth of metaprogramming in C++ started one evening when a group of experts realized that the template technology of C++ (designed to support generic programming) was a Turing-complete programming language in its own right, one that was implemented at compile time. C++ was not designed for metaprogramming, so using it is often ugly, but it has enabled experts to explore the potential of metaprogramming.

The second way in which C++ is a living language is that it is subject to periodic change. Even as I write, those responsible for the definition of C++ (WG21, an ISO standards committee) are working on changes that will eventually come into effect at the end of this decade. Some of those changes are to make C++ easier to write and to learn, some are aimed at cleaning up inconsistencies, and some will be aimed at further extending the power of the language. At the time of writing it is impossible to predict exactly what will be added and what changes will be introduced. I know that providing better support for metaprogramming is one of the potential additions to C++.

I am only sharing this with you so that you can see that C++ is alive and well and continuing to develop in a controlled and measured way. What you learn from the rest of this book will be valid for a number of years, and even after the next release of the C++ Standard, almost everything you have learned will still be

true, but there may be some easier ways to achieve the same objectives. One of the guidelines for growth and development of ISO programming languages is that every effort should be made to preserve existing source code. This is in marked contrast to some non-ISO programming languages that change on much shorter cycles and often in ways that break existing code.

Different Backgrounds

In the rest of this chapter I am going to attempt to make some helpful comments based on the most likely languages that you might already have learned. Every language has strengths. Every language has its own special properties, and those using them will have learned various idioms. Some of those idioms will not easily transfer to C++.

If your first language uses a syntax that is like that of C++ (for example C, Java, or C#) you are going to have to overcome the tendency to think that what looks the same behaves the same.

In general you are going to have to fight the natural tendency to think the way that your first language works is the way that programming languages should work. There are wide differences in languages. Most computer languages have been designed by highly intelligent people. The differences are not accidental, and the differences do not make one language better than another. Sometimes we can capitalize on the way one language works to write a program in another one.

Many years ago the warden of the Oxford Schools Sailing Centre asked me to write a race-analysis program for him to use on his Acorn BBC Microcomputer (an old British desktop computer built around the 8-bit 6502 processor). The native language for that machine was a version of BASIC. Now in those days my language of choice for problem solving was Forth (check `http://www.forth.org/` if you want to know more). I spent a Saturday morning at the warden's house being fed coffee and biscuits while I wrote the program for him, superficially, in BBC BASIC. When the job was done, he looked at it and said, ''That does not look anything like BBC BASIC.'' He was spot on: the program was designed in Forth and implemented in BBC BASIC. Every program statement was a correct BBC BASIC statement, but no BBC BASIC programmer would have written that source code. Another Forth programmer would have recognized it despite the use of a different programming language.

Sometimes, as a once-off task, you can do something like that. The result was neither good Forth nor good BBC BASIC. It had only one merit: it met the client's needs and worked correctly. If a student learning BBC BASIC had produced that program, they would have lost marks from me for inappropriate use of the language. Mastering a language includes mastering its idioms.

Fundamental C++ for C++ Programmers

No, I have not lost the thread. Quite a few readers may already have done some preliminary programming with C++. Some of them will have had a good experience and a good introduction, while others will have had a less sound introduction.

My book *You Can Do It!* [Glassborow 2004] is specifically a book about programming, and all the focus of that book was on how to program. It uses C++ as the language for writing programs, but it was not written as a book about C++ and so left out a great deal of the language. (I estimate that the book covers only about 10% of C++.) While there is some overlap between that book and this one, those who acquired their basic programming skills through my first book can learn much more about C++ from this one.

There are many other books that introduce C++ as a first language. Some of these do a good job and some a rather mediocre one. You might also have learned C++ at school or from a web-based tutorial. In any of those cases, you might reasonably want a more complete introduction to C++ fundamentals. Like all those who come to this book having gained their programming basics from some other language, you will possibly need to unlearn some things and change the focus of others. If what you read in the rest of this book seems to conflict with what you learned previously, you will need to think hard. Perhaps your earlier learning led you

to too specific an interpretation of the way something worked. Perhaps the original source was plain wrong. It is even possible that this text is plain wrong (though I sincerely hope not).

One definite area of potential conflict is that the C++ I am writing about here is very different from the C++ of the early '90s, which many other books introduce. The syntax of the language has not changed (though there have been some additions), but the way that the best modern programmers use that syntax to express solutions to problems has changed out of all recognition.

Try to use any tension resulting from conflicts between what you have previously learned about C++ and what this book says. In trying to resolve those conflicts, you will come to a much better understanding of the way C++ works. While I have given a good deal of thought to the way I write and teach C++, that does not mean that I own 'the one true way'. There is no such thing. There are many bad ways to write C++ and several good ways. One hallmark of a good way is consistency. If I choose to express an idea with a different idiom it is, I hope, because there is a difference. I try not to do something one way on weekdays and a different way at weekends.

However, there is a caveat. One of the ways that programs differ is in their purpose. For example, much of the code in this book is designed to help you learn C++. That is quite different from code written as part of a large application. When you are working as a member of a team, you should conform to the presentational and coding standards of the team. However, you should also take pride in every line of code you write. I do not go for the concept of 'egoless programming'; we should be proud of our contributions, but we should not seek to make those contributions irritate by being in a different style from that of the rest of the team.

The step from good personal code to industrial-strength code is considerable. When we write code for ourselves, we can often ignore the fact that what we write will only work on the machine we are using, or that it depends on the development tools we have. On the other hand, when learning C++ you should have a clear division between what is Standard C++ and any extras. I think you will find it in this book.

Fundamental C++ for C Programmers

The traditional route to C++ was through learning C. Unfortunately, many excellent aspects of C are no longer so sound in the context of C++. In C++ we have to consider issues such as exception safety (be patient, we will get there!), polymorphism, `const` correctness, and overloading of functions and operators. These issues lead us to a very different programming style.

One small example is the issue of using pointers. Good C often relies heavily on correct and extensive use of pointers; good C++ frequently avoids pointers or uses a C++ mechanism for encapsulating pointers (called smart pointers) to control their potential for damage.

Another example is that in C++ we rarely, if ever, use a dynamic array (the kind of object that a C programmer creates with `malloc( )` and manages with `realloc( )`). C++ provides a vector container (`std::vector<>`) that automates most of the resource handling.

C++ provides real types to handle both strings of `char` and wide strings of `wchar_t` (which is a full built-in type in C++ rather than an alias for some integer type as it is in C). It also provides a convenient mechanism for creating other string types such as those we might want for Japanese.

C++ input and output mechanisms are designed to be type safe, unlike those in C, where it is the responsibility of the programmer to ensure that the type the programmer says will be provided is the type actually provided.

A C programmer learning C++ needs to focus on understanding why C++ does some things differently. Contrary to what you may read elsewhere C++ is not better than C; despite the deliberate compatibility of syntax, C++ is a distinctly different language. Unless you come to understand that, your C++ will remain C in disguise. That would be to sell both yourself and C++ short.

There is nothing in C++ that prevents you from using C dynamic arrays, arrays of `char` and `wchar_t`, or members of the `printf` and `scanf` families of I/O functions. It is just that C++ provides alternative mechanisms whose use is less demanding of programmer time.

Sometimes I may write that C++ has a better mechanism for doing something than C. Such statements should be taken in the broader context of programming rather than as an implication that C++ is better than

C. C is fundamentally a language for programming close to the metal. That makes it an excellent language for writing for small, embedded systems such as the dozens of micro-controllers that permeate modern life. C++ was designed for writing very big suites of code where millions of lines of source code is not unusual.

Fundamental C++ for Java Programmers

One of the problems you are going to have is that much of Java syntax is also valid C++ syntax but it does something slightly different. The underlying object models of the two languages are subtly (some would say radically) different.

In some ways, you are going to have more difficulty than anyone else does in learning C++. You are going to be constantly writing code that you expect to work. It will compile and then do something bizarrely unexpected. You will have to resist the natural temptation to think either that your compiler is broken or that C++ must be a broken language. Java has many strengths, and I am certainly not going to denigrate it (I do not do language wars). But learning Java as a first language causes considerable problems when you then try to learn C++.

Major issues concern such things as order of evaluation: Java strictly defines this; C++ states that it can be in any order. Another issue is the concept of references: C++ references simply do not behave the same way that Java ones do. For example, every C++ reference is required to refer to an actual object and refers to the same object throughout its lifetime.

You will also find that there are issues with constructors: superficially, C++ and Java constructors seem to do the same thing, but when we look under the hood, we find that they work differently. At the other end we have the concept of destructors to end the lifetime of an object, but C++ does not have garbage collection (though there is nothing in the language to prevent it being added as long as the programmer avoids some C++ mechanisms). The lack of garbage collection was a deliberate design decision; it can always be added but it cannot be taken away. Java source code expects a garbage collector and will be badly flawed without one. C++ written on the assumption that there is no garbage collection will usually work correctly even if there is.

All these issues and many others are going to make things difficult, because you will need to relearn your code-reading skills.

I am reminded of an episode from my school days, when I was a reasonable chess player. We had the then (1956) British Chess Champion, C.H.O'D. Alexander, visiting the school to give us some instruction and to play the school in a simultaneous match. At the end, we were discussing the subject of fairy chess (chess played with various modified rules). One variation he suggested was to simply swap the values of the pieces so that, for example, the one that looked like a bishop was actually used as a knight.

If you play chess, try it. I think you will understand the problem of things not being all that they appear to be. That is the major problem that Java programmers have when learning C++.

Fundamental C++ for C# Programmers

C# is superficially even closer to C++ (the similarity of the sharp sign to overlapping plus signs is not accidental) and so is going to add to code-reading problems.

Like the Java programmer you are going to have to learn the concept of global functions and data. That can be a hard step. Unlike the Java programmer your first language does incorporate the ideas of destructors and user-defined value types. However, the C# concept of a destructor is very different from the C++ one, so you will have to do some careful thought getting those differences clear in your mind.

C# has garbage collection and is heavily constrained by the requirement that it function under a CLI (Common Language Infrastructure) based virtual machine such as .NET or Mono. For example, types are more strictly defined; they have to be because all languages running on a CLI VM have to agree about the size and layout of the fundamental types.

Again, as for Java, C# references are not exactly the equivalent of C++ references, so be careful when I write about references—they may not be what the word leads you to expect.

In C++, the keywords `struct` and `class`, which are used to define a user type, are very close to being synonymous. So close that the only reason why good C++ programmers ever use `struct` is to emphasize that what they are writing is basically a simple C-style aggregate type. This is not the place to explain why C++ has two words without a substantive difference, though some of us think that this was one place where Bjarne Stroustrup made a less-than-ideal decision.

In C++, the difference between value types (those that C# defines as `struct`s) and object types (close to the C# concept of a `class` type) is entirely in the way the programmer designs the type. There is no semantic content in the choice of `struct` or `class` in C++.

Fundamental C++ for COBOL Programmers

C++ comes as a nasty shock to long-term COBOL programmers. I can remember sitting in the back of an introductory C++ course (I was assessing the instructor). One of the students had programmed in COBOL for 25 years. He was finding great difficulty with the concept of a pointer, because he had never needed such a mechanism in all the years that he had programmed in COBOL. The instructor was having quite a bit of difficulty in trying to get the student to see that C++ needed something that COBOL did not. At the end of the week-long course, the student assessed the course as a waste of time. In a sense, it was, for him. As it happens, I knew his manager, who told me the rest of the story. He had been sent on the course as a last attempt to break him out of a very narrow and specific view of programming. It had not worked, with the consequence that his employer could no longer use him as a programmer, and his rigid thinking made him ill-suited to other work in the company.

I relate this sad tale as a warning to others. A rigid view of how programming works makes professional development difficult and ultimately impossible. If you come to C++ from a radically different language you will need to get down to fundamentals such as decisions (`if–else`), looping (`for` and `do–while`), and functions or procedures. Those basics are shared by all languages, but the exact way they are implemented can vary considerably. Think of natural languages. English, Chinese, Arabic, and Hindi are all powerful human languages, but if you think an alphabet is fundamental to writing, you will have a great deal of difficulty with Chinese.

I was brought up in Sudan, and I can remember my mother's struggle with teaching one of our servants to read English (at his request). In Arabic the basic meaning is carried by the consonants, and the difference between verbs, nouns, adjectives, etc. is largely conveyed by the vowels. Now Abdul knew that English was different, but knowing and believing are different things. One day he sat there struggling to see the logic behind 'hat', 'hot', 'hate', 'heat', 'hut', 'hit', and 'hoot'. From his perspective, there was an obvious relationship between a hut and a hat, and you wore a hat when it was hot, and 'hate' sort of fitted, being a hot emotion. Just possibly, 'hit' could be worked in, because you wore a hat to protect yourself both from the sun's heat and from being hit on the head. However, how did 'hoot' get into the mix?

Beware of trying to force your view of a topic into some preconceived framework. Just as 'ht' does not label a fundamental concept in English, you must avoid trying to force C++ into the mould of your first language.

Fundamental C++ for Python Programmers

Python is a very interesting language. However, those that learn programming using Python must be careful to distinguish programming fundamentals from the way that Python implements them. One of the interesting features of Python is that the program structure is communicated by the program layout. Levels of indentation matter in Python, whereas they do not in C++. C++ uses braces, { and }, to block statements together. We encourage C++ programmers to use indentation to make their code more readable for human beings, but indentation has no significance to the compiler.

Another important feature of Python is something it shares with a few other languages: names are just names. A name in Python takes on the characteristics of whatever was last assigned to it. This is an immensely

powerful programming concept, but one that does not work well in the case of a compiled language. C++ source code can be interpreted (I know of one reasonable interpreter for C++) but is generally compiled to machine code for the platform on which the program will run. That is why a compiled C++ program for a Linux machine will not run on a Microsoft Windows machine.

Python gets around the problem by supplying a Python program (effectively a virtual machine) that takes Python source code as data and executes it. In this, it is a little like Java: Java programs only run in the context of special programs called Java Virtual Machines. However, Python goes further than Java by making a great deal more use of the immediacy of code.

I am not going to attempt to explain the gains and losses of each mechanism. All I can do is to warn you that they are very different; each language has its advantages and each has its costs.

Fundamental C++ for (Visual) Basic Programmers

Many years ago BASIC (computer languages were spelled in all uppercase in those days) was designed as a language for teaching programming. It has come in for a good deal of criticism from academics over the years, some of it justified and some of it not. The biggest problem was not in BASIC but in the way that it was often taught. It trapped many teachers into teaching bad programming.

Visual Basic is a Microsoft proprietary language that is a refinement and extension of the original BASIC. It is a powerful language that has been damaged by its owner's propensity for releasing incompatible dialects. One of the better features of Visual Basic is the provision of visual-programming tools that allow you to generate code for complicated components from a palette of more basic ones. One surprise to VB programmers is that Visual C++ (simply Microsoft's toolkit for C++ programming) does not supply any kind of visual-programming tools for C++. There are tools around for visual programming with C++ but they are largely of limited use, because the strengths of C++ are not with visualization but in the exact expression of computation and in data management.

C++ is much more to do with the underlying programming than it is to do with constructing applications from available components. This makes it a more powerful tool when you want to do something different, i.e. something for which there are no pre-designed components.

When you learn C++ you are going to be concerned with expressing solutions to computational problems rather than being focused on data capture and data display. When you have a firm grasp of the C++ fundamentals you will be able to use libraries of pre-built components to handle some input and output problems, but that is not the focus of C++. You may even find one of the visual-programming tools useful (though they are not cheap).

Fundamental C++ for Pascal and Delphi Programmers

Pascal was a language that set out with the grandiose intention of providing a computing language that was hard to abuse. Unfortunately, the consequence was that it created a generation of programmers with one of two bad mindsets. Highly talented programmers knew that Pascal was getting in the way and preventing them from doing things that they knew could both be done and be done safely. This led them to that form of creative programming that used to be called 'hacking'. In other words, they found ways round the well-intentioned obstructiveness of the compiler. The rest of those learning Pascal tended to have a view that if the compiler would compile it, then the code was OK. That created a mindset that what a compiler would compile was safe.

A Pascal programmer reading this book is unlikely to have such a mindset, but be warned: C++ expects users to be responsible, to understand what they are doing, and to avoid doing dangerous things. A C++

compiler will not try to second-guess you and refuse to compile code simply because the result might be dangerous.

One of the more famous problems that C++ has inherited from C is the concept of a buffer overrun (a problem that is by no means exclusive to these languages). C++ expects that when you provide storage for input you will make sure that you provide sufficient storage or add a mechanism to prevent excessive input overwriting other data. If you program carelessly you produce programs that are vulnerable to being overloaded with data. In these days of the Internet, many of us have discovered the bad consequences of such carelessness. Pascal would probably have died out many years ago were it not for the efforts of a single company, Borland. Borland created a version of Pascal with a considerable number of extensions. Those extensions provided a safe and correct way to do many of the things programmers wanted but which were prevented by Standard Pascal.

More recently Borland further enhanced Pascal with features to support object-oriented programming. That extended dialect of Pascal is called Delphi. In addition, they then enhanced their C++ development tools to support mixing of C++ and Delphi. The result is that if you are a Delphi programmer you must be very careful that you do not fall into the trap of believing that C++ is what you get when you use Borland C++ Builder (their implementation of C++) in its Delphi compatibility mode (which is very tempting because you then have the use of an extensive third-party library, written in Delphi but accessible from Borland's extended C++).

In general, Pascal programmers have to learn to trust themselves to get code correct and not trust the compiler to reject dangerous constructs.

Fundamental C++ for Functional Programmers

If you come from a background of functional programming (perhaps having learned Haskell, Scheme, or ML) you will likely be shocked by what C++ so cavalierly calls a function. In C++, not only are functions allowed to have side effects, but they usually do. Perhaps one day we will have a mechanism to tell a C++ compiler that something really is a function in the mathematical/functional-programming sense but that day is not here yet.

By default, C++ variables are indeed variable. Like other programming languages, C++ allows and even encourages assignment. If you are a functional programmer, you are first going to have to master the instinctive revulsion you may feel for such an 'ill-disciplined' language.

For the purpose of this book, you are going to have to put much that you have learned to one side. But do not discard it, because when you move on to less fundamental C++ (not in this book but perhaps the next one) you will find that many of those ideas and idioms that you are familiar with allow you to become a fluent user of some of the advanced aspects of generic programming and metaprogramming. Indeed the very best C++ programmers often deliberately choose to learn a language such as Haskell in order to enhance their C++ skills.

Fundamental C++ for Lisp and Logo Programmers

It is hard to know where to start if you have mastered Lisp. First, you must have been lucky to have been taught by someone who understood how Lisp works. I say that because far too much Lisp is taught by instructors for whom it is a third or fourth language. The problem is that they often do not think in Lisp; they know its syntax but for them it is like an English speaker writing Japanese by using a dictionary and a grammar to translate from English.

The same problem applies to those who have truly learnt Logo (usually by trial and error). If your knowledge of Logo stopped with Turtle Graphics then you may not have too big a struggle with learning C++ but if you went much further then you will need to focus on the fundamental ideas of functions, repetition

and decisions. The semantics of those are common to all computer languages but the syntax is significantly different.

Just as those that learn Lisp or Logo as a third or fourth language struggle because they keep trying to impose the structures of procedural languages onto the new language they are learning, those going the other way will have to abandon the thinking they have developed to handle list processing and imperative languages. At least you will have to suspend those ways of thought until you have mastered the fundamentals of C++.

Fundamental C++ for Object-Oriented Programmers

From time to time people describe C++ as an object-oriented programming language (OOPL). It is not; it is a language in which you can do some forms of object-oriented programming (OOP). If you want to do pure OOP, try a language such as Smalltalk. However, if you come from such a background be very careful, because the C++ form of OO is significantly different from what you will have learned. There is enough similarity to lull you into a sense of security and enough difference to make that a false one.

By default, methods (well, we call them member functions in C++) are statically bound. That is, implementation code is selected at compile time, not at execution time. C++ provides a mechanism for delaying binding until execution time (so-called dynamic binding), but the programmer has to make a positive decision that that is what they want.

In C++, not everything is an object in the sense usually meant by an OOPL. When we talk about objects in C++, we do not mean exactly what a Smalltalk programmer would mean by an object. The concept is near but not an exact duplicate.

Fundamental C++ for Every Programmer

There are many other languages that you might have learned. Fortran, Modula 2 or 3, Forth, and Prolog are just a few that I know to a greater or (usually) lesser extent. If you are old enough you might be fluent in PL/1, and if you are a mathematician you might use APL (an outstanding interpreted language for those who think mathematically—and almost impossible for the rest to grasp). You might also have learned Ada. If you really dig into the odder corners you might have learned SNOBOL. I just throw that last one in because it was a great language that did not catch on. However, I believe Andy Koenig has a set of libraries and other tools to enable SNOBOL mechanisms and idioms to be used in C++.

Whatever language or languages you already know, C++ has something to offer you, and your prior knowledge and skills have something to offer your study of C++. The most important thing is that you do not try to make C++ just another way to write the language you already know. This is particularly important if you are learning C++ because you want to broaden your job opportunities. Knowing enough C++ to get through a job interview will not help you if you then try to write Xlang in C++ clothes.

Good programmers write fluently in many computer languages; bad programmers write the same bad code in many computer languages. I hope that by the time you finish your study of C++, you are both a better programmer and a good C++ one.

CHAPTER 1

Getting Started

This chapter introduces you to the tools that I have provided on the CD for your use as you read this book. You may prefer to use some that you already have, or you may prefer to use the IDE provided on the CD. Whatever choice you make, it will be to your advantage to work through this chapter and check that each step behaves correctly (or that you can achieve the equivalent with the tools of your choice). If you use MinGW Developer Studio (which I will refer to as MDS), you will be able to check that you are doing the right thing by comparing what you see on your screen with the screenshots in this chapter. If you are using JGrasp™ from the CD, you will be able to check the alternative version of this chapter that is on the CD. If you are using some other tools, you should still work through this chapter to make sure that you understand how to use them to produce a simple program. Most importantly, you will need to check that you can use my library with your choice of tools.

Before you go any further, read the file `Read_First.txt` in the root directory of the CD. That will tell you what is on the CD and where to find the instructions for installing the software for the operating system used by your computer (Microsoft Windows or a UNIX derivative such as Linux or Mac OS X). When you have done that, or have installed some other development tools of your choice, continue from here.

It may be that you are familiar with the kind of tools used with a compiled language because one of the languages you already use has similar tools. If so, please excuse me for taking time explaining things for the benefit of other readers.

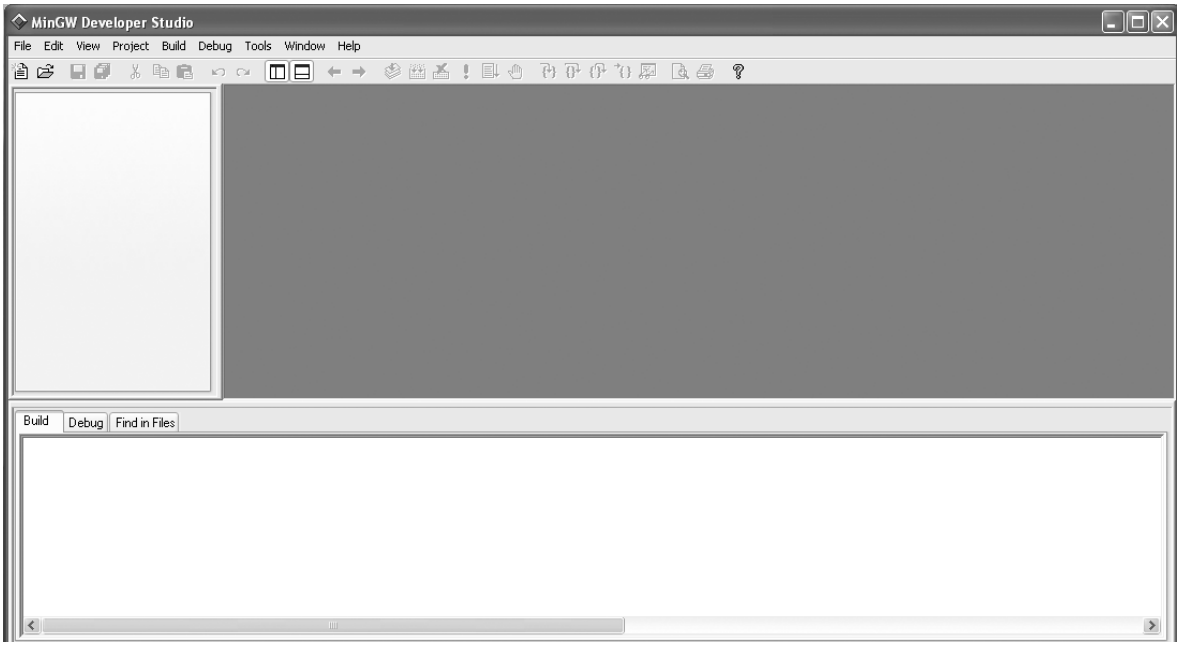
In this book, I assume that you are using an Integrated Development Environment (IDE), which is the programmer's equivalent of a carpenter's workbench. Some programmers are used to using the command line; that is fine if you understand how to write your own makefiles. If you do not (or even have no idea what those are), you would probably be better off using an IDE that will automate much of the interaction between the stages of writing, compiling, linking and debugging your code.

I am now going to walk through the process of producing two simple programs from scratch. The first program is the traditional 'Hello World'. The second one is to check that everything has been correctly set up to use my graphics library. These programs are deliberately simple so that we can focus on the process of creating a program from source code and libraries. As I go, I will add information that may be new to you if you are used to a language that is substantially different from C++.

Creating a 'Hello World' Program

The first step is to launch, or start, the IDE. I always have the IDE icon on my desktop. If you accepted that option when you installed the software from the CD, you will find this icon somewhere on your desktop:

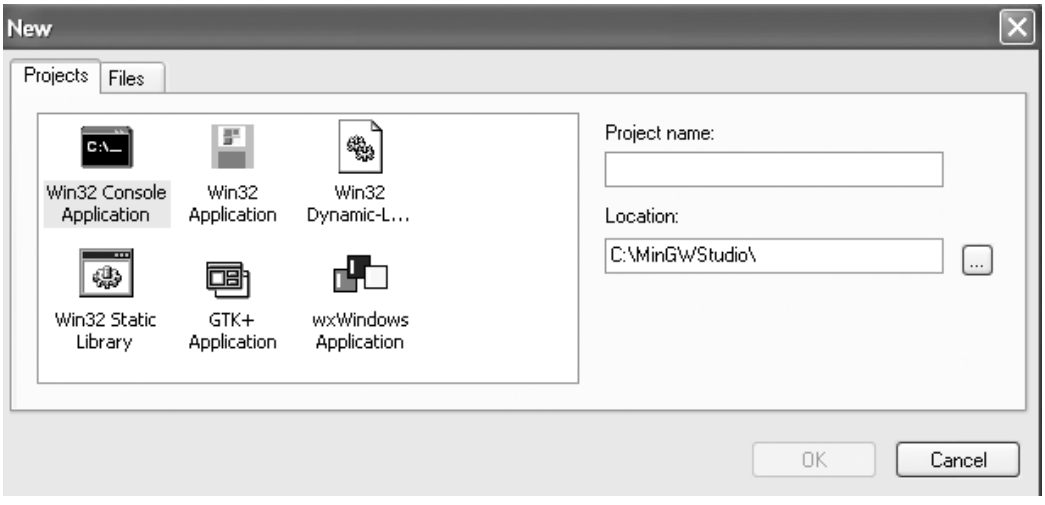
If you chose otherwise, you will have to launch MDS differently. When you have launched MDS, you should see this window on your screen:



The various panes are adjustable and we will see what each is for as we go. If you are familiar with using an IDE, you will probably recognize most of what is on the screen. If you start pulling down menus, you will notice that most of the entries are 'grayed out'. Most of the icons are grayed out too, but unless you have disabled the feature, you will get a tool-tip if you let the mouse cursor hover over an icon. These tips are minimal, but I find the inclusion of the default hotkey in the tip a useful feature.

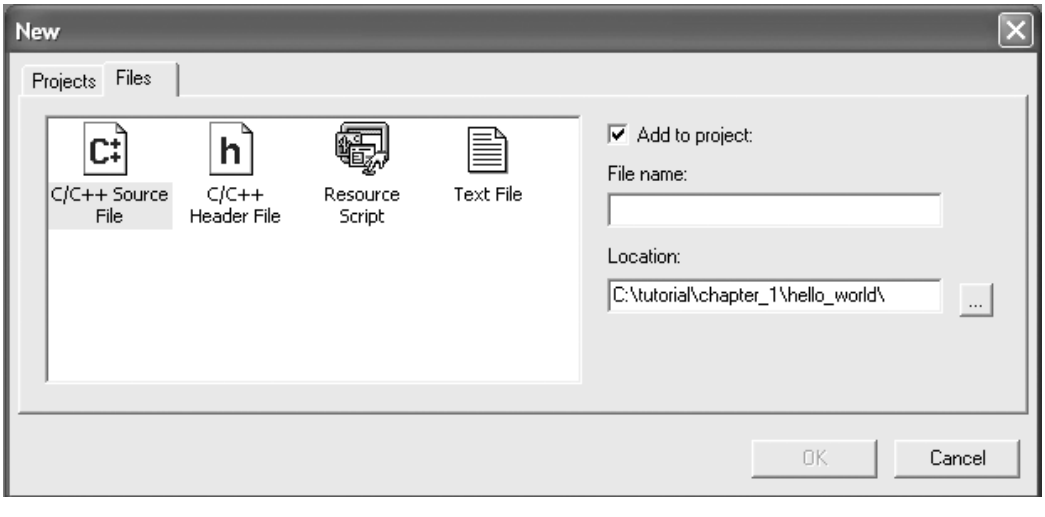
Your next step is to go to the Project menu and select New Project (Ctrl+N if you prefer to use hotkeys).

When you do this, you will see a data-capture window that looks like this:



The default project type is a console application, and that is what you will want through most of this book. You need to tell the IDE the name of the project and the location of your files for this project. You are less likely to make a mistake if you identify the location first. If you look at the lower text box, you will see a small gray button on the right. Click on it and then navigate to the appropriate subdirectory. If you installed MDS, by default that will be `C:\tutorial\chapter_1`. Now go to the upper box and give your project a name. Type in `hello_world` (note that MDS does not correctly handle file names that contain spaces) and click on the OK button.

Next you need to create a file for the source code (the correct term for what we write; the compiler turns that into object code). Select New from the File menu and you will see:



The defaults are correct (or they should be). You are creating a C/C++ source file and adding it to the project; the location of the file is the same place that you placed the project itself. All you need to do is give the file a name. Type `hellomain` into the File name box. After I have opened the FileView tree (in the left-hand pane), the top left of my work window looks like this:

Type the following into the file pane (the right-hand one, which has a '1' in the dark-gray margin zone):

```
// First program written on 21/07/04
#include <iostream>

int main( ){
  std::cout << "Hello World";
}
```

Make sure you press Enter after the closing brace, so that the file ends with a newline character.

As you type, MDS tries to give help by its use of color. When you type an unmatched bracket (parenthesis, brace, or square bracket) it first displays it in red. You can use Ctrl+B to go from one bracket to the matching one (if there is one). The MDS editor also supports code folding (the ability to hide code leaving only a header element); try clicking on the minus sign to the left of '`int main`' and you will see what I mean. It is not

very useful here, but can be helpful when you want to hide extraneous details while you focus on some other section of code.

When you have typed in the code, select Compile from the Build menu. If you have done everything correctly, you should see

```
hellomain.o - 0 error(s), 0 warning(s)
```

in the bottom pane. If you forgot to press Enter after the closing brace of the source code, you will get a warning. The compiler issues a warning because C++ files can include other files; a missing empty line at the end of a file could cause a problem because the compiler might append the next line of code to that line. If you are used to using an interpreted language such as Python, you may be wondering what you have achieved so far. A tool called a compiler has converted your text (source code) into a form (object code) that the linker can use to produce an executable program. We will look at the rest of the source code shortly, but for now I want you to produce an executable program and run it. You can do that step by step, by selecting Build from the Build menu and then Execute from the same menu; or you can just select Build and execute. The choice is entirely yours. You can even select Execute directly, and MDS will ask if you want to build the executable.

However you execute the program, you should finish up with the following console window:

MDS inserts a closing message (after your program has finished) so that the window will stay open until you are finished looking at the results. 'Terminated with return code 0' means that the program finished satisfactorily.

I want you to try several things before we go on. First, I want you to change the project settings. Select Settings from the Project menu. Now select the Compile tab and check all the warning boxes except for the last one (making all warnings into errors is usually overkill). For now, leave everything else as it is.

The settings can be separately set for a debug version and for a release version. While we are developing a program, we are usually willing to accept slower performance and a much larger program file in exchange for more help if the program encounters some problem while it is running.

However, when we are ready to share our work with others, we probably do not want to have a very big, slow program, so we produce a release version. The release version is normally much smaller and possibly appreciably faster, because the compiler has worked hard to strip out all unnecessary details.

Do not worry about the other tabs in the Project Settings dialog box. We will only be using the Compile and Link panes.

Next I would like you to take some time experimenting with the source code (try adding extra lines of output, leaving out a semicolon, and introducing other typos) until you are confident with compiling, building, and executing it. You are going to spend a lot of time with these tools, so it is worthwhile spending a little time gaining some fluency with them. That is one reason for starting with such a silly little program: it allows us to focus on the basics of our tools before we use them to write programs with more substance.

What the Code Means

Let us look briefly at the six lines that made up the 'Hello World' program.

The first line is a comment. Whenever the compiler meets a `//`, it ignores everything from there to the end of the line. Comments are for humans and sometimes for special code-analysis tools; they are not for compilers. In this case, the comment is almost redundant, and I have included it only as an example.

The second line tells the compiler that the following code may use names from the part of the Standard C++ Library concerned with streaming data in and out through the console. We call the part in the angle brackets a 'header'; it tells the compiler to get relevant information from wherever it keeps such details. Different compilers may obtain the information in different ways. In practice, a header is usually a text file in one of the compiler's subdirectories. (If you are interested, you will find the corresponding `iostream` file in `MinGWStudio\MinGW\include\c++\3.4.2`, but I doubt that it will make much sense to you yet.)

The blank line has no significance and is there purely to separate the introductory part of the source code from the rest. The next line (`int main( ){`) must exist exactly once in every program. Effectively it determines where a program starts. There are some variants that allow the provision of data at program start-up, but that is all.

The fifth line is the substance of the program. `std::cout` is the name of a console output object. In other words, it is the name we use to designate an object that represents the console on the computer where the program will run; the console is usually a window on the monitor screen. The part of the name before the double colon tells us that we are dealing with a name from the C++ Standard Library.

> *Language Note: C programmers will recognize `<<` as being the left-shift operator. In the context of an output object or destination, C++ reuses that operator as a streaming operator, to insert data into an output stream.*

The text in quotation marks tells the compiler that you want this text displayed. We call such quoted text a string literal. The final element of the statement is the semicolon. That ends the statement and is the C++ equivalent of a period in English. Try leaving it out and then attempting to compile the code; you will see the kind of error message that results.

The last line of the program is a simple closing brace to match the opening brace at the end of the line with `main` in it. In general, we refer to code between an opening and closing brace as a 'block'. In

this context, the code between the opening brace and the closing brace is the definition of this version of `main` and specifies what will happen when the program executes. We call a block that defines a function the 'function body'. So

```
{
  std::cout << "Hello World";
}
```

is the body of the function `main( )`.

Our Second Program – An Empty Playpen

From time to time, we are going to use a very special graphics window that I designed and several of my colleagues helped to implement. It is called the Playpen. (As you use it, I think you will come to appreciate the choice of name.) This is a fixed size (512 by 512) graphics window, with each pixel limited to one of 256 colors. Modern computers are usually capable of displaying many more colors than that, but I wanted something that was very portable as well as something that would allow you to learn about simple graphics systems. For now, we are going to use the Playpen with its default palette. (Later we will discover that we can choose different sets of 256 colors.)

Start up MDS and select New from the Project menu. Make sure that the location is correct (you are probably going to have to add '`chapter_1`' to what it offers you – at least that is what I have to do on my machine). Now enter '`playpen`' as the project name and press Enter.

Next, using the File menu or Ctrl+N, create a new C/C++ source file called `emptyplaypen`, and type in the following short program:

```
// written on 21/07/04
#include "playpen.h"
#include <iostream>

int main( ){
  fgw::playpen blank;
  std::cout << "Please press the 'ENTER' key";
  std::cin.get( );
}
```

Try to compile the source-code file (Ctrl+F7). It will fail with four or more error messages. The only useful one is the first, where it says it cannot find `playpen.h`. The reason for that is that we need to tell it where to look for header files. (Headers are part of the C++ Standard, but header files are a bit different: they are provided by third-party programmers and relate to third-party code in much the same way that headers relate to the Standard C++ Library.)

Choose Settings from the Project menu and select the Compile tab. In the box labeled 'Additional include directories', at the bottom of the Compile pane, enter '`C:\tutorial\fgw_headers`' (or a variant of that if you installed the CD to some other drive or directory). In other words, tell the compiler to look in a subdirectory called `fgw_headers` that is in the `tutorial` directory on drive `C`. While you are dealing with the location of this header file, check that the same warnings are selected as for the first project.