Second Edition

C++Programming

Made Simple

Incorporating Standard C

Conor Sexton

MADE SIMPLE BOOKS

Made Simple
An imprint of Elsevier Science
Linacre House, Jordan Hill, Oxford OX2 8DP
225 Wildwood Avenue, Woburn MA 01801-2041

First published 2003

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN 0 7506 5738 3

Typeset by Elle and P.K. McBride, Southampton
Icons designed by Sarah Ward © 1994
Printed and bound in Great Britain

Contents

Preface

C++ Programming Made Simple – Second Edition is intended as an introduction to programming in the C++ language as codified by the 1998 ISO C++ Standard. It is not a reference book and does not pretend to be in any way comprehensive. The intention is to provide an accessible starting-point to people who:

- have no programming experience
- have programmed in some other high-level language
- know C or an earlier version of C++ and need an update
- in any case need a working, practical, knowledge of C++.

This book owes a good deal to its two 1997 predecessors *C Programming Made Simple* and *C++ Programming Made Simple*. At the time I wrote these, it was still customary to present C++ as an extension of C. Because C++ was then relatively new, many people were in a position of knowing C and needing an 'upgrade' to C++. This situation has now changed. The distinction between C and C++ has blurred if not disappeared: in the ISO Standard, C++ incorporates C. It can no longer be assumed that people will have a knowledge of C before approaching C++. What is called for – and, I hope, provided by this book – is an integrated coverage of C++ including the parts of C that are not obsolete.

Achieving this has involved much more than simply merging the two previous texts. First, C++ had priority: where a C idiom is replaced by a newer C++ construct, the former is no longer covered. Second, a great deal has changed in C++ in the six years since the original two *Made Simples*. The language has been 'tweaked' in a thousand details. The C++ Library and the Standard Template Library in particular are largely new. A complete revision was necessary; I hope that it is evident that this is a new book and not just a re-hash of two old ones.

This book does not try to take a rigorous approach to C++. Coverage of the language aims to be adequate for practical needs, not complete. Many of the 'dark corners' of C++ (and some not-so-dark ones) are not covered. Even with this selectiveness, the book still comes out at over 300 pages. I estimate that a completely comprehensive coverage of modern C++, including the Standard Library, would weigh in at more than 1,500 pages. The objective, then, is to get you 'up and running' with useful C++ grounded in clear, practical, program examples.

The book has 14 chapters, which I don't list here: you can see them in the Table of Contents. I think of it as having four main parts:

- In time-honoured fashion, a lightning overview of C++ essentials (Chapter 1)
- The 'C-heavy' part of the book, with C++ syntax integrated as appropriate (Chapters 2 to 7)
- Traditional C++, involving classes and inheritance (Chapters 8 to 10)
- Modern C++, including templates and the Standard Library (Chapters 11 to 14)

At those points where a topic could grow beyond the scope of a *Made Simple* book, I acknowledge the fact, and make suggestions for further reading.

I have enjoyed the various aspects of writing this book: using some material from the previous *Made Simples*; eliminating obsolete aspects of C; updating the C++ syntax and adding completely new sections. It may be optimistic of me to ask you to enjoy reading it; I hope at least that you find it useful.

Conor Sexton
Dublin

1 A quick start with C++

Background to ISO C++ language

The C++ programming language is an *object-oriented* (OO) derivative of C. It is almost true to say that C is a subset of C++. In fact, every ISO C program written in the modern idiom (specifically, with new-style function headers) and avoiding certain C++ reserved words is also a C++ program, although it is not object-oriented.

This book owes a good deal to two of my previous publications in the *Made Simple* series, *C Programming Made Simple* (0-7506-3244-5) and *C++ Programming Made Simple* (0-7506-3243-7) both published by Butterworth-Heinemann in 1997. Since that time, and, particularly, since the September 1998 ratification of the ISO C++ Standard, C has been completely subsumed by C++. C as a language no longer exists in its own right; it is therefore no longer acceptable to take what was once the conventional approach and treat C and C++ separately. Accordingly, this book presents a fully-integrated treatment of the ISO C++ language incorporating C. Parts of the original C language and library are still valid but have been replaced by superior C++ facilities. Examples of such C constructs include void parameter lists and library functions including printf, malloc and others. This book does not deal with the (obsolescent) C mechanisms, but concentrates on the facilities provided by C++. Finally, from now on in this book, the simple term 'C++' should be taken to mean 'ISO C++'

It is the aim of this first chapter to get you up and running quickly with C++. The remaining chapters go into somewhat more depth on a variety of C++ constructs and programming techniques.

C++ is for technical computer programming and is suitable for development of 'techie' software like operating systems, graphical interfaces, communications drivers and database managers. In the modern Web context, C++ is one of the three or four languages of choice for implementation of applications in the so-called *middle tier*. These applications constitute what is often called the *business logic*: the body of application code resident on an intermediate system between the (browser-based) user and the (typically database) resource at the *back end*.

The main alternative to C++ is Java. Java is the medium in which the Java 2 Enterprise Edition (J2EE) web application architecture is implemented. Java has an advantage over C++ in not having to be compiled for every type of computer on which programs written in the language must run; it is therefore more easily *portable* (movable between different systems) than C++ and particularly suitable for web applications.

At the time of writing, there are two main web application architecture 'camps', J2EE – originating from Sun Microsystems Inc. – and Microsoft's C# (pronounced 'C-sharp') and .Net combination. J2EE exclusively uses Java; C#/.Net allows use of C++ (called 'Managed C++'), C# and Visual Basic. The predecessor to the C#/.Net architecture, the Component Object Model (COM), is still widely in use and employs C++ and Visual Basic as its two primary languages.

Although Java has an advantage of portability over C++ in the web applications context, C++ is still widely present on the web, very often written in the form of COM objects or CGI (Common Gateway Interface) programs. C++ also retains the advantages of performance and flexibility over Java. Line for line, because Java is not compiled into an optimised executable form, C++ is likely to be faster in execution on a given system. In addition, C++ retains constructs (such as pointers, eliminated by Java) that allow it full access to all operating system and machine facilities, with corresponding flexibility and power.

C++ was originally developed in the early 1980s at AT&T Bell Laboratories by Dr Bjarne Stroustrup. In the almost 20 intervening years, there has been a myriad of twists and turns to the development and standardisation of the language. Mostly, these are no longer important. It's enough to refer to the standardisation process, which was started by the American National Standards Institute (ANSI) in 1990 when it formed the standardisation committee X3J16. About the same time, the International Organization for Standardization (ISO) formed its committee, ISO-WG-21, also for the purpose of standardising C++ on a worldwide basis. The efforts of the two committees were made joint from 1990 and it was at the time expected that a ratified ANSI/ISO C++ standard would be approved by 1994. In the event, there were significant additions to the scope of the work involved – most notably the addition of the C++ Standard Library including the Standard Template Library (STL) – and the Final Draft International Standard (FDIS) was not published until late 1997. The International Standard (IS) was ratified in September 1998. Its ISO title is *Information Technology – Programming Languages – C++*, with associated document number ISO/IEC 14882-1998. Though originated by ANSI, the standard is an ISO one; the documented one is distributed by national standards bodies subordinated to ISO (ANSI in the case of the United States).

ISO C++ (also called 'Standard C++') is now the single unified definition of the C++ language, and it is becoming increasingly difficult to find books and compilers that do not at least claim to conform to the Standard. The first edition of this book (*C++ Programming Made Simple*, 1997) is not ISO-Standard compliant, but is close to being so. If you know to make a few small but significant changes to the structure and syntax of programs, that book still serves as a viable presentation of the C++ language. This edition presents and explains the necessary changes, as well as a subset of the extensions (mostly in the area of the Standard Library) that are mandated by the Standard.

Some of the major characteristics of C++ are these:

- C++ provides a powerful, flexible and expressive *procedural* language (alongside an object-oriented or class-based) component grounded in the earlier C language.
- C++ implements *objects*, defined as classes, which incorporate data definitions, along with declarations and definitions of functions that operate on that

data. This *encapsulation* of data and functions in a single object is the central innovation of C++.

- *Instances* of classes may automatically be initialised and discarded using *constructors* and *destructors*. This eliminates program initialisation errors.
- The way in which C++ classes are defined enforces *data hiding*; data defined in a class is by default only available to the *member functions* of that class. External, or *client*, code that uses a class cannot tamper with the internal implementation of the class but is restricted to accessing the class by calling its member functions.
- C++ allows *overloading* of operators and functions. More than one definition of a function may be made having the same name, with the compiler identifying the appropriate definition for a given function call. Ordinary operators such as '++' and '->' can also be overloaded with additional meanings.
- C++ allows the characteristics of the class type — data and functions — to be inherited by *subclasses*, also called *derived classes*, which may in turn add further data and function definitions. This encourages reuse of existing code written in the form of shareable class libraries and consequent savings in the cost of the software development process. *Multiple inheritance* allows for derived classes to inherit characteristics from more than one base class.
- C++ allows classes to define *virtual functions*: more than one definition of a function, with the decision as to which one is selected being resolved at program run-time. This is *polymorphism*, with the run-time selection among function definitions being referred to as *late binding* or *dynamic binding*.
- *Template* classes can be defined which allow different instances of the same class to be used with data of different types but with unchanged code. This further promotes code reuse.
- ISO C++ introduces the standardised C++ Library, which includes the Standard Template Library (STL). The Standard Library incorporates and improves the old (see the first edition of this book) Stream I/O library and adds many utilities and other features. The STL provides programmer-friendly implementations of many common data structures – including list, stack, queue, string – along with the operations necessary to manipulate them. Thus, many of the 'cool' programming techniques familiar to advanced C programmers of the 1980s and early 1990s are now packaged and made invisible for you, much in the same way that the operation of a car's engine is hidden from the average owner by the bonnet (or hood, depending on where you live!).

++ facilities for object-oriented programming are characterised by classes, inheritance and virtual functions. These facilities make C++ particularly suitable for writing software to handle a multitude of related objects. A typical use of C++ is in implementing graphical user interfaces (GUIs), where many different but related objects are represented on a screen and are allowed to interact. Using the object-

oriented approach, C++ stores these objects in class hierarchies and, by means of virtual functions, provides a generic interface to those objects (e.g. draw object), which saves the programmer from having to know the detail of how the objects are manipulated. This makes it easier for the programmer to develop and maintain code, as well as rendering less likely the introduction of bugs into existing code. C++ and other object-oriented languages are also, as we have seen, central to the modern component-based architectures such as .Net and J2EE, in which reuse of proven and tested objects improves prospects for reliability of applications of ever-increasing complexity.

That's enough overview stuff. Let's write our first program!

The do-nothing program

The minimal C++ program is this:

```
main(){}
```

This is a complete C++ program. Every C++ program must consist of one or more functions. The code shown above is a program consisting exclusively of a main function. Every C++ program must have one (and only one) main function. When it is executed, the program does nothing.

A more strictly-correct C++ form of the do-nothing program is this:

```
#include <iostream>
int main(){return 0;}
```

The whole program shown is stored in a file called donowt.cpp. The .cpp part is necessary, meaning that the file contains a C++ program; 'donowt' is at your discretion. iostream is a *standard header file* that contains useful declarations for compilation and execution of the program that follows. If you are aware of the syntax of pre-ISO C++, you'll know that the name of the header file was iostream.h; this form is still usable.

iostream is an alternative to but does not replace the C standard header file cstdio. Again, in pre-ISO C++, this was called stdio.h; the 'h' has now been removed at the behest of the C++ standardisation committee and the leading 'c' added to make clear the header file's C Library lineage. iostream declares C++ library functions and facilities (see Chapters 12 and 13); cstdio does the same for Standard C Library functions such as printf. The int preceding main is the function's *return type*. It specifies that the program returns a value (in this case, zero) to the operating system when it is run. The function's parentheses are empty: the function cannot accept any parameters. When the program finishes executing (it does nothing), the return statement returns execution control to the operating system.

Here's the most-correct (by ISO criteria) version of donowt:

```
#include <iostream>
using namespace std;
int main(){}
```

The standard, or std, *namespace* is introduced. This has no practical effect as yet, but its purpose is to allow objects of the same name to be used in different contexts without a name-clash: you might want to use the standard-output object cout to mean two different things, so the first version would belong to the standard namespace, while the second would belong to another namespace. More of this later; for now (and for most of the book) we'll confine ourselves to the standard namespace.

Notice also that the return statement is gone. ISO C++ no longer requires this; you may include the line but, if you don't, the C++ system inserts an implicit one for you and ensures that control is correctly returned to the operating system.

Here is a rather complex version of the do-nothing C++ program, donowt.cpp. It uses a trivial C++ class to produce no output:

```
//  donowt.cpp - program using a simple C++
//  class to display nothing

#include <iostream>
using namespace std;

class nodisp
{
private:
public:
    void output()
    {
        return;
    }
};

int main()
{
    nodisp screen;

    screen.output();
}
```

This time, the program declares a C++ class, nodisp, of which the only member is a function, output. In the main function, we define an instance of the class:

```
nodisp screen;
```

The function nodisp::output() (the member function output of the class nodisp) when executed from main with the line screen.output(); simply returns control to the following statement. As this is the end of main, donowt.cpp stops without making any output.

Brackets and punctuation

A note on the different kinds of brackets: the names of standard header files are enclosed in angle brackets <>; function argument lists, after the function name, in parentheses (); and code blocks in curly braces {}. Statements, such as return; must be terminated with a semicolon. C++ programs are free-form – you can write the code in any format, jumbled up on one lien of text if you want. The structured layout shown in this book is not strictly necessary but is a good idea for readability and avaoidance of error.

Building and running a C program

The filename suffix for C++ programs is usually .cpp on PCs. On computers running the UNIX operating system, the C++ source code filename may end with any of several suffixes, including .c, .C, .cxx and .cpp. This book uses exclusively the suffix .cpp.

The donowt.cpp program, in any of the forms shown above, must first be converted by compiler and linker programs into executable code. For this book, I'm assuming use of the Borland C++ Builder 5 development suite. This has a vast range of supporting facilities for development of GUI and component-based objects in C++; it also has a very good command-line compiler, which is what I use in this book. If you're using a PC with the Borland C++ Builder 5 compiler and linker, you can compile donowt.cpp using this command line:

```
bcc32 donowt.cpp
```

This produces an output file called donowt.exe, which you can run at the command line, admiring the spectacular lack of results that ensues.

For some Microsoft C++ compilers, you can use 'c-ell':

```
cl donowt.cpp
```

More usually, you use the *integrated development environment* (IDE) provided by the Microsoft Visual C++ 6.0 or .Net environments.

If you're using a UNIX system, you can compile and *load* (UNIX-speak for link) the program using a number of different command-line formats, depending on the UNIX variant. I can't specify the precise command-line input for your UNIX system, so I present a few possibilities below (note that UNIX distinguishes between upper- and lower-case characters entered at the command line):

```
CC donowt.C      // UNIX System V
g++ donowt.cpp   // Linux
c++ donowt.C     // Also Linux, .cxx and .c suffixes OK too
```

Whichever command line is correct for your system (you'll have to experiment a bit), the resulting executable program is in a file called a.out (for *assembler output*, believe it or not).

Some programs can be built (compiled and linked) at the command line in the ways shown. Programs that make GUI displays, as well as programs for the modern component environments such as Microsoft's .Net, cannot in practical terms be built using the command line. It's more likely that you will use an IDE provided by Microsoft, Borland, IBM or another supplier. The IDE uses a menu-driven interface that is better for managing programs of significant size.

It's not the subject of this book to tell you how to use the IDEs of any software supplier. I assume that the information you now have will enable you to build at least simple C++ programs, and we move forward now to writing programs that actually do something.

Here's one, called message1.cpp:

```
//  message1.cpp - program to display a greeting

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello C++ World\n";
}
```

The double-slash notation is a comment: all characters following the double slash, //, on the same line are ignored. The /*.....*/ notation is also used in C++ but, for short comments, // is preferred.

The header file iostream contains class and function declarations that are #included by the *preprocessor* in the source code file and are necessary for C++ Library facilities to be used. These facilities include cout, of the class type ostream; ostream is declared in the iostream header file.

cout is an object representing the *standard output stream*. The characters to the right of the << operator are sent to cout, which causes them to be displayed on the user's terminal screen, assuming that is the standard output device. The << operator is in fact the bitwise left-shift operator *overloaded* by the C++ system to mean 'insert on a stream'. The C++ Stream I/O system is explained in Chapter 13.

You should try entering this program at your computer and building it. As an exercise, make message.cpp display two lines:

```
Ask not what your country can do for you
Ask rather what you can do for your country
```

If you were to omit the line

```
using namespace std;
```

you would get a compilation error complaining about cout being an 'undefined symbol'. This is because cout needs to be specified as being part of some namespace. This can be specified explicitly as:

```
std::cout << "Hello C++ World\n";
```

with the same result as if using namespace.... had been retained.

Here's a class-based version of the message program, message2.cpp, that produces exactly the same output – Hello C++ World – as the simpler form of the program shown above:

```
//  message2.cpp - program using a simple C++
//  class to display a greeting

#include <iostream>
using namespace std;

class message
{
private:
public:
    void greeting()
    {
        cout << "Hello C++ World\n";
    }
};

int main()
{
    message user;

    user.greeting();
}
```

You'll see much more about classes throughout this book and, in particular, in Chapter 8. For now, I'll briefly describe the message class and its contents. Everything within the enclosing curly braces following message is a member of the class message. All the members of message are declared public; they are generally accessible. message only has one public member, a function called greeting which has no return type or argument list.

In the function main, we define an instance of the message class, called user. The greeting function is called and the Hello C++ World message displayed by the function call:

```
user.greeting();
```

Use of a class in this case is overkill, but from it you should be able to understand simple characteristics of the class construct.

Enough C++ to get up and running

While message2.cpp does produce a visible result, it's not very useful. To produce more functional C++ programs, you must know a minimum set of the basic building-blocks of the C++ language. This section presents these building blocks, under a number of headings:

- Variables
- Operators
- Expressions and statements
- Functions
- Branching
- Looping
- Arrays
- Classes
- Constructors and destructors
- Overloading
- Inheritance

Variables

Variables in C++ are data objects that may change in value. A variable is given a name by means of a definition, which allocates storage space for the data and associates the storage location with the variable name.

The C++ language defines five fundamental representations of data:

boolean

integer

character

floating-point

double floating-point

Each of these is associated with a special *type specifier*:

bool	specifies a true/false value
int	specifies an integer variable
char	specifies a character variable
float	specifies a fractional-number variable
double	specifies a fractional-number variable with more decimal places

Any of the type specifiers may be qualified with the type qualifier const, which specifies that the variable must not be changed after it is initialised.

A data definition is of the following general form:

```
<type-specifier> <name>;
```

A variable name is also called an *identifier*. The following are some examples of simple data definitions in C++:

```
int       apples;          // integer variable
char      c;               // character value eg: 'b'
float     balance;         // bank balance
const     double x = 5;    // high-precision variable
                           // value fixed when set
bool      cplusplus = TRUE;
```

Operators

C++ has a full set of arithmetic, relational and logical operators. The binary arithmetic operators in C++ are:

+	addition	-	subtraction
*	multiplication	/	division
%	modulus		

There is no operator for exponentiation; in line with general C++ practice, this is implemented as a special function in an external library.

Both + and - may be used as unary operators, as in the cases of -5 and +8. There is no difference between +8 and 8.

The modulus operator, %, provides a useful remainder facility:

```
17%4 // gives 1, the remainder after division
```

The assignment operator, =, assigns a value to a memory location associated with a variable name. For example:

```
a = 7;
pi = 3.1415927;
```

Relational operators in C++ are:

<	less than	>	greater than
>=	greater than or equal to	<=	less than or equal to
!=	not equal		
==	test for equality		

Care is needed in use of the equality test ==. A beginning programmer will at least once make the mistake of using a single = as an equality test; experienced programmers do it all the time!

Writing

```
x = 5;
```

assigns the value 5 to the memory location associated with the name x.