Siddhartha Rao

SEVENTH EDITION

Covers C++11

Sams Teach Yourself

C++ in One Hour a Day

SAMS

Siddhartha Rao

Sams Teach Yourself C++ in One Hour a Day

Seventh Edition

SAMS
800 East 96th Street, Indianapolis, Indiana 46240

Sams Teach Yourself C++ in One Hour a Day, Seventh Edition

ISBN-13: 978-0-672-33567-9
ISBN-10: 0-672-33567-0

The Library of Congress Cataloging-in-Publication Data is on file.

Printed in the United States of America

Third Printing April 2013

Trademarks

Warning and Disclaimer

Bulk Sales

Acquisitions Editor
Mark Taber

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Charlotte Kughen

Indexer
Tim Wright

Proofreader
Megan Wade

Technical Editor
Jon Upchurch

Publishing Coordinator
Vanessa Evans

Designer
Gary Adair

Compositor
Studio Galou, LLC

Contents at a Glance

Appendixes

Table of Contents

PART V: Advanced C++ Concepts

About the Author

Siddhartha Rao is a technologist at SAP AG, the world's leading supplier of enterprise software. As the head of SAP Product Security India, his primary responsibilities include hiring expert talent in the area of product security as well as defining development best practices that keeps SAP software globally competitive. Awarded Most Valuable Professional by Microsoft for Visual Studio–Visual C++, he is convinced that C++11 will help you program faster, simpler, and more efficient C++ applications.

Siddhartha also loves traveling and discovering new cultures given an opportunity to. For instance, parts of this book have been composed facing the Atlantic Ocean at a quaint village called Plogoff in Brittany, France—one of the four countries this book was authored in. He looks forward to your feedback on this global effort!

Dedication

This book is dedicated to my lovely parents and my wonderful sister for being there when I have needed them the most.

Acknowledgments

I am deeply indebted to my friends who cooked and baked for me while I burned the midnight oil working on this project. I am grateful to the editorial staff for their very professional engagement and the wonderful job in getting this book to your shelf!

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone number or email address.

E-mail: feedback@samspublishing.com

Mail: Reader Feedback
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

2011 was a special year for C++. With the ratification of the new standard, C++11 empowers you to write better code using new keywords and constructs that increase your programming efficiency. This book helps you learn C++11 in tiny steps. It has been thoughtfully divided into lessons that teach you the fundamentals of this object-oriented programming language from a practical point of view. Depending on your proficiency level, you will be able to master C++11 one hour at a time.

Learning C++ by doing is the best way—so try the rich variety of code samples in this book hands-on and help yourself improve your programming proficiency. These code snippets have been tested using the latest versions of the available compilers at the time of writing, namely the Microsoft Visual C++ 2010 compiler for C++ and GNU's C++ compiler version 4.6, which both offer a rich coverage of C++11 features.

Who Should Read This Book?

The book starts with the very basics of C++. All that is needed is a desire to learn this language and curiosity to understand how stuff works. An existing knowledge of C++ programming can be an advantage but is not a prerequisite. This is also a book you might like to refer to if you already know C++ but want to learn additions that have been made to the language in C++11. If you are a professional programmer, Part III, "Learning the Standard Template Library (STL)," is bound to help you create better, more practical C++11 applications.

Organization of This Book

Depending on your current proficiency levels with C++, you can choose the section you would like to start with. This book has been organized into five parts:

- Part I, "The Basics," gets you started with writing simple C++ applications. In doing so, it introduces you to the keywords that you most frequently see in C++ code of a variable without compromising on type safety.
- Part II, "Fundamentals of Object-Oriented C++ Programming," teaches you the concept of classes. You learn how C++ supports the important object-oriented programming principles of encapsulation, abstraction, inheritance, and polymorphism.

Lesson 9, "Classes and Objects," teaches you the new C++11 concept of move constructor followed by the move assignment operator in Lesson 12, "Operator Types and Operator Overloading." These performance features help reduce unwanted and unnecessary copy steps, boosting the performance of your application. Lesson 14, "An Introduction to Macros and Templates," is your stepping stone into writing powerful generic C++ code.

- Part III, "Learning the Standard Template Library (STL)," helps you write efficient and practical C++ code using the STL `string` class and containers. You learn how `std::string` makes simple string concatenation operations safe and easy and how you don't need to use C-style `char*` strings anymore. You will be able to use STL dynamic arrays and linked lists instead of programming your own.
- Part IV, "More STL," focuses on algorithms. You learn to use `sort` on containers such as `vector` via iterators. In this part, you find out how C++11 keyword `auto` has made a significant reduction to the length of your iterator declarations. Lesson 22, "C++11 Lambda Expressions," presents a powerful new feature that results in significant code reduction when you use STL algorithms.
- Part V, "Advanced C++ Concepts," explains language capabilities such as smart pointers and exception-handling, which are not a must in a C++ application but help make a significant contribution toward increasing its stability and quality. This part ends with a note on best practices in writing good C++11 applications.

Conventions Used in This Book

Within the lessons, you find the following elements that provide additional information:

NOTE

> These boxes provide additional information related to material you read.

C++11

These boxes highlight features new to C++11. You may need to use the newer versions of the available compilers to use these language capabilities.

CAUTION

> These boxes alert your attention to problems or side effects that can occur in special situations.

TIP

> These boxes give you best practices in writing your C++ programs.

DO	DON'T
DO use the "Do/Don't" boxes to find a quick summary of a fundamental principle in a lesson.	**DON'T** overlook the useful information offered in these boxes.

This book uses different typefaces to differentiate between code and plain English. Throughout the lessons, code, commands, and programming-related terms appear in a computer `typeface`.

Sample Code for this Book

The code samples in this book are available online for download from the publisher's website.

This page intentionally left blank

LESSON 2

The Anatomy of a C++ Program

C++ programs consist of classes, functions, variables, and other component parts. Most of this book is devoted to explaining these parts in depth, but to get a sense of how a program fits together, you must see a complete working program.

In this lesson, you learn

- The parts of a C++ program
- How the parts work together
- What a function is and what it does
- Basic input and output operations

Part of the Hello World Program

Your first C++ program in Lesson 1, “Getting Started,” did nothing more than write a simple “Hello World” statement to the screen. Yet this program contains some of the most important and basic building blocks of a C++ program. You use Listing 2.1 as a starting point to analyze components all C++ programs contain.

LISTING 2.1 `HelloWorldAnalysis.cpp`: Analyze a C++ Program

```
 1:  // Preprocessor directive that includes header iostream
 2: #include <iostream>
 3:
 4: // Start of your program: function block main()
 5: int main()
 6: {
 7:    /* Write to the screen */
 8:    std::cout << "Hello World" << std::endl;
 9:
10:    // Return a value to the OS
11:    return 0;
12: }
```

This C++ program can be broadly classified into two parts: the preprocessor directives that start with a `#` and the main body of the program that starts with `int main()`.

NOTE

> Lines 1, 4, 7, and 10, which start with a `//` or with a `/*`, are comments and are ignored by the compiler. These comments are for humans to read.
>
> Comments are discussed in greater detail in the next section.

Preprocessor Directive `#include`

As the name suggests, a *preprocessor* is a tool that runs before the actual compilation starts. Preprocessor directives are commands to the preprocessor and always start with a pound sign `#`. In Line 2 of Listing 2.1, *`#include <filename>`* tells the preprocessor to take the contents of the file (`iostream`, in this case) and include them at the line where the directive is made. `iostream` is a standard header file that is included because it contains the definition of `std::cout` used in Line 8 that prints “Hello World” on the screen. In other words, the compiler was able to compile Line 8 that contains `std::cout` because we instructed the preprocessor to include the definition of `std::cout` in Line 2.

NOTE

In professionally programmed C++ applications, not all includes are only standard headers. Complex applications are typically programmed in multiple files wherein some need to include others. So, if an artifact declared in FileA needs to be used in FileB, you need to include the former in the latter. You usually do that by putting the following `include` statement in FileA:

```
#include "...relative path to FileB\FileB"
```

We use quotes in this case and not angle brackets in including a self-created header. <> brackets are typically used when including standard headers.

The Body of Your Program `main()`

Following the preprocessor directive(s) is the body of the program characterized by the function `main()`. The execution of a C++ program always starts here. It is a standardized convention that function `main()` is declared with an `int` preceding it. `int` is the return value type of the function `main()`.

NOTE

In many C++ applications, you find a variant of the `main()` function that looks like this:

```
int main (int argc, char* argv[])
```

This is also standard compliant and acceptable as `main` returns `int`. The contents of the parenthesis are "arguments" supplied to the program. This program possibly allows the user to start it with command-line arguments, such as

```
program.exe /DoSomethingSpecific
```

`/DoSomethingSpecific` is the argument for that program passed by the OS as a parameter to it, to be handled within `main`.

Let's discuss Line 8 that fulfills the actual purpose of this program!

```
std::cout << "Hello World" << std::endl;
```

`cout` ("console-out", also pronounced see-out) is the statement that writes "Hello World" to the screen. `cout` is a stream defined in the standard namespace (hence, `std::cout`), and what you are doing in this line is putting the text `"Hello World"` into this stream by using the stream insertion operator <<. `std::endl` is used to end a line, and inserting it into a stream is akin to inserting a carriage return. Note that the stream insertion operator is used every time a new entity needs to be inserted into the stream.

The good thing about streams in C++ is that similar stream semantics used with another stream type result in a different operation being performed with the same text—for example, insertion into a file instead of a console. Thus, working with streams gets intuitive, and when you are used to one stream (such as `cout` that writes text to the console), you find it easy to work with others (such as `fstream` that helps write text files to the disk).

Streams are discussed in greater detail in Lesson 27, "Using Streams for Input and Output."

NOTE

The actual text, including the quotes `"Hello World"`, is called a *string literal*.

Returning a Value

Functions in C++ need to return a value unless explicitly specified otherwise. `main()` is a function, too, and always returns an integer. This value is returned to the operating system (OS) and, depending on the nature of your application, can be very useful as most OSes provide for an ability to query on the return value of an application that has terminated naturally. In many cases, one application is launched by another and the parent application (that launches) wants to know if the child application (that was launched) has completed its task successfully. The programmer can use the return value of `main()` to convey a success or error state to the parent application.

NOTE

Conventionally programmers return 0 in the event of success or -1 in the event of error. However, the return value is an integer, and the programmer has the flexibility to convey many different states of success or failure using the available range of integer return values.

CAUTION

C++ is case-sensitive. So, expect compilation to fail if you write `Int` instead of `int`, `Void` instead of `void`, and `Std::Cout` instead of `std::cout`.

The Concept of Namespaces

The reason you used `std::cout` in the program and not only `cout` is that the artifact (`cout`) that you want to invoke is in the standard (`std`) namespace.

So, what exactly are namespaces?

Assume that you didn't use the namespace qualifier in invoking `cout` and assume that `cout` existed in two locations known to the compiler—which one should the compiler invoke? This causes a conflict and the compilation fails, of course. This is where namespaces get useful. Namespaces are names given to parts of code that help in reducing the potential for a naming conflict. By invoking `std::cout`, you are telling the compiler to use that one unique `cout` that is available in the `std` namespace.

NOTE

> You use the `std` (pronounced "standard") namespace to invoke functions, streams, and utilities that have been ratified by the ISO Standards Committee and are hence declared within it.

Many programmers find it tedious to repeatedly add the `std` namespace specifier to their code when using `cout` and other such features contained in the same. The `using namespace` declaration as demonstrated in Listing 2.2 will help you avoid this repetition.

LISTING 2.2 The `using namespace` Declaration

```
// Pre-processor directive
#include <iostream>

// Start of your program
int main()
{
   // Tell the compiler what namespace to search in
   using namespace std;

   /* Write to the screen using std::cout */
   cout << "Hello World" << endl;

   // Return a value to the OS
   return 0;
}
```

Analysis ▼

Note Line 8. By telling the compiler that you are using the namespace `std`, you don't need to explicitly mention the namespace on Line 11 when using `std::cout` or `std::endl`.

A more restrictive variant of Listing 2.2 is shown in Listing 2.3 where you do not include a namespace in its entirety. You only include those artifacts that you wish to use.

LISTING 2.3 Another Demonstration of the `using` Keyword

```
 1: // Pre-processor directive
 2: #include <iostream>
 3:
 4: // Start of your program
 5: int main()
 6: {
 7:    using std::cout;
 8:    using std::endl;
 9:
10:    /* Write to the screen using cout */
11:    cout << "Hello World" << endl;
12:
13:    // Return a value to the OS
14:    return 0;
15: }
```

Analysis ▼

Line 8 in Listing 2.2 has now been replaced by Lines 7 and 8 in Listing 2.3. The difference between `using namespace std` and `using std::cout` is that the former allows all artifacts in the `std` namespace to be used without explicitly needing to specify the namespace qualifier `std::`. With the latter, the convenience of not needing to disambiguate the namespace explicitly is restricted to only `std::cout` and `std::endl`.

Comments in C++ Code

Lines 1, 4, 10 and 13 in Listing 2.3 contain text in a spoken language (English, in this case) yet do not interfere with the ability of the program to compile. They also do not alter the output of the program. Such lines are called *comments*. Comments are ignored by the compiler and are popularly used by programmers to explain their code—hence, they are written in human- (or geek-) readable language.

C++ supports comments in two styles:

- `//` indicates that the line is a comment. For example:

  ```
  // This is a comment
  ```

- `/*` followed by `*/` indicates the contained text is a comment, even if it spans multiple lines:

  ```
  /* This is a comment
  and it spans two lines */
  ```

2

NOTE

It might seem strange that a programmer needs to explain his own code, but the bigger a program gets or the larger the number of programmers working on a particular module gets, the more important it is to write code that can be easily understood. It is important to explain what is being done and why it is being done in that particular manner using well-written comments.

DO	DON'T
Do add comments explaining the working of complicated algorithms and complex parts of your program. **Do** compose comments in a style that fellow programmers can understand.	**Don't** use comments to explain or repeat the obvious. **Don't** forget that adding comments will not justify writing obscure code. **Don't** forget that when code is modified, comments might need to be updated, too.

Functions in C++

Functions in C++ are the same as functions in C. Functions are artifacts that enable you to divide the content of your application into functional units that can be invoked in a sequence of your choosing. A function, when called (that is, invoked), typically returns a value to the calling function. The most famous function is, of course, `main()`. It is recognized by the compiler as the starting point of your C++ application and has to return an `int` (i.e., an integer).

You as a programmer have the choice and usually the need to compose your own functions. Listing 2.4 is a simple application that uses a function to display statements on the screen using `std::cout` with various parameters.

LISTING 2.4 Declaring, Defining, and Calling a Function That Demonstrates Some Capabilities of `std::cout`

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // Function declaration
 5: int DemoConsoleOutput();
 6:
 7: int main()
 8: {
 9:    // Call i.e. invoke the function
10:    DemoConsoleOutput();
11:
12:    return 0;
13: }
14:
15: // Function definition
16: int DemoConsoleOutput()
17: {
18:    cout << "This is a simple string literal" << endl;
19:    cout << "Writing number five: " << 5 << endl;
20:    cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
21:    cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
22:    cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
23:
24:    return 0;
25: }
```

Output ▶

```
This is a simple string literal
Writing number five: 5
Performing division 10 / 5 = 2
Pi when approximated is 22 / 7 = 3
Pi more accurately is 22 / 7 = 3.14286
```

Analysis ▶

Lines 5, 10, and 15 through 25 are those of interest. Line 5 is called a *function declaration*, which basically tells the compiler that you want to create a function called `DemoConsoleOutput()` that returns an `int` (integer). It is because of this *declaration* that the compiler agrees to compile Line 10, assuming that the *definition* (that is, the implementation of the function) comes up, which it does in Lines 15 through 25.

This function actually displays the various capabilities of `cout`. Note how it not only prints text the same way as it displayed "Hello World" in previous examples, but also the result of simple arithmetic computations. Lines 21 and 22 both attempt to display the result of pi (22 / 7), but the latter is more accurate simply because by diving 22.0 by 7, you tell the compiler to treat the result as a real number (a `float` in C++ terms) and not as an integer.

Note that your function is stipulated to return an integer and returns 0. As it did not perform any decision-making, there was no need to return any other value. Similarly, `main()` returns 0, too. Given that `main()` has delegated all its activity to the function `DemoConsoleOutput()`, you would be wiser to use the return value of the function in returning from `main()` as seen in Listing 2.5.

LISTING 2.5 Using the Return Value of a Function

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // Function declaration and definition
 5: int DemoConsoleOutput()
 6: {
 7:    cout << "This is a simple string literal" << endl;
 8:    cout << "Writing number five: " << 5 << endl;
 9:    cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
10:    cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
11:    cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
12:
13:    return 0;
14: }
15:
16: int main()
17: {
18:    // Function call with return used to exit
19:    return DemoConsoleOutput();
20: }
```

Analysis ▶

The output of this application is the same as the output of the previous listing. Yet, there are slight changes in the way it is programmed. For one, as you have defined (i.e., implemented) the function before `main()` at Line 5, you don’t need an extra declaration of the same. Modern C++ compilers take it as a function declaration and definition in one. `main()` is a bit shorter, too. Line 19 invokes the function `DemoConsoleOutput()` and simultaneously returns the return value of the function from the application.

> **NOTE**
>
> In cases such as this where a function is not required to make a decision or return success or failure status, you can declare a function of return type `void`:
>
> ```
> void DemoConsoleOutput()
> ```
>
> This function cannot return a value, and the execution of a function that returns `void` cannot be used to make a decision.

Functions can take parameters, can be recursive, can contain multiple return statements, can be overloaded, can be expanded in-line by the compiler, and lots more. These concepts are introduced in greater detail in Lesson 7, "Organizing Code with Functions."

Basic Input Using `std::cin` and Output Using `std::cout`

Your computer enables you to interact with applications running on it in various forms and allows these applications to interact with you in many forms, too. You can interact with applications using the keyboard or the mouse. You can have information displayed on the screen as text, displayed in the form of complex graphics, printed on paper using a printer, or simply saved to the file system for later usage. This section discusses the very simplest form of input and output in C++—using the console to write and read information.

You use `std::cout` (pronounced "standard see-out") to write simple text data to the console and use `std::cin` ("standard see-in") to read text and numbers (entered using the keyboard) from the console. In fact, in displaying "Hello World" on the screen, you have already encountered `cout`, as seen in Listing 2.1:

```
8:    std::cout << "Hello World" << std::endl;
```

The statement shows `cout` followed by the insertion operator << (that helps insert data into the output stream), followed by the string literal "Hello World" to be inserted, followed by a new line in the form of `std::endl` (pronounced "standard end-line").

The usage of `cin` is simple, too, and as `cin` is used for input, it is accompanied by the variable you want to be storing the input data in:

```
std::cin >> Variable;
```

Thus, `cin` is followed by the extraction operator >> (extracts data from the input stream), which is followed by the variable where the data needs to be stored. If the user input needs to be stored in two variables, each containing data separated by a space, then you can do so using one statement:

```
std::cin >> Variable1 >> Variable2;
```

Note that `cin` can be used for text as well as numeric inputs from the user, as shown in Listing 2.6.

LISTING 2.6 Use `cin` and `cout` to Display Number and Text Input by User

```
 1: #include <iostream>
 2: #include <string>
 3: using namespace std;
 4:
 5: int main()
 6: {
 7:    // Declare a variable to store an integer
 8:    int InputNumber;
 9:
10:    cout << "Enter an integer: ";
11:
12:    // store integer given user input
13:    cin >> InputNumber;
14:
15:    // The same with text i.e. string data
16:    cout << "Enter your name: ";
17:    string InputName;
18:    cin >> InputName;
19:
20:    cout << InputName << " entered " << InputNumber << endl;
21:
22:    return 0;
23: }
```

Output ▶

```
Enter an integer: 2011
Enter your name: Siddhartha
Siddhartha entered 2011
```

Analysis ▶

Line 8 shows how a variable of name `InputNumber` is declared to store data of type `int`. The user is requested to enter a number using `cout` in Line 10, and the entered number is stored in the integer variable using `cin` in Line 13. The same exercise is repeated with storing the user's name, which of course cannot be held in an integer but in a different type called `string` as seen in Lines 17 and 18. The reason you included `<string>` in Line 2 was to use type `string` later inside `main()`. Finally in Line 20, a `cout` statement is used to display the entered name with the number and an intermediate text to produce the output `Siddhartha entered 2011`.

This is a very simple example of how basic input and output work in C++. Don't worry if the concept of variables is not clear to you as it is explained in good detail in the following Lesson 3, "Using Variables, Declaring Constants."

Summary

This lesson introduced the basic parts of a simple C++ program. You understood what `main()` is, got an introduction to namespaces, and learned the basics of console input and output. You are able to use a lot of these in every program you write.

Q&A

Q What does `#include` do?

A This is a directive to the preprocessor that runs when you call your compiler. This specific directive causes the contents of the file named in `<>` after `#include` to be inserted at that line as if it were typed at that location in your source code.

Q What is the difference between `//` comments and `/*` comments?

A The double-slash comments (`//`) expire at the end of the line. Slash-star (`/*`) comments are in effect until there is a closing comment mark (`*/`). The double-slash comments are also referred to as *single-line comments*, and the slash-star comments are often referred to as *multiline comments*. Remember, not even the end of the function terminates a slash-star comment; you must put in the closing comment mark or you will receive a compile-time error.

Q When do you need to program command-line arguments?

A To allow the user to alter the behavior of a program. For example, the command `ls` in Linux or `dir` in Windows enables you to see the contents within the current directory or folder. To view files in another directory, you would specify the path of the same using command-line arguments, as seen in `ls /` or `dir \`.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to answer the quiz and exercise questions before checking the answers in Appendix D, and be certain you understand the answers before continuing to the next lesson.

Quiz

1. What is the problem in declaring `Int main()`?
2. Can comments be longer than one line?

Exercises

1. **BUG BUSTERS:** Enter this program and compile it. Why does it fail? How can you fix it?

```
#include <iostream>
void main()
{
     std::Cout << Is there a bug here?";
}
```

2. Fix the bug in Exercise 1 and recompile, link, and run it.
3. Modify Listing 2.4 to demonstrate subtraction (using –) and multiplication (using *).

This page intentionally left blank

Index

B

D

G-H

I

J-K

L

Q-R

S

T

W

X-Y-Z