Rogers Cadenhead
Jesse Liberty

SIXTH EDITION
Updated for C++14

Sams Teach Yourself C++ in 24 Hours

SAMS

Contents at a Glance

Rogers Cadenhead
Jesse Liberty

Sams Teach Yourself C++ in 24 Hours

SIXTH EDITION

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself C++ in 24 Hours

ISBN-13: 978-0-672-33746-8
ISBN-10: 0-672-33746-0

Library of Congress Control Number: 2016945006

First Printing August 2016

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or programs accompanying it.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact

governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact

intlcs@pearson.com.

Editor
Mark Taber

Project Editor
Lori Lyons

Project Manager
Prashanthi Nadipalli

Copy Editor
Christopher Morris

Technical Editor
Jon Upchurch

Contents at a Glance

Part VI: Special Topics

Part VII: Appendixes

Table of Contents

Part VI: Special Topics

Part VII: Appendixes

About the Authors

Rogers Cadenhead is a writer, computer programmer, and web developer who has written more than 25 books on Internet-related topics, including *Sams Teach Yourself Java in 21 Days* and *Absolute Beginner's Guide to Minecraft Mods Programming*. He publishes the Drudge Retort and other websites that receive more than 22 million visits a year. This book's official website is at http://cplusplus.cadenhead.org.

Jesse Liberty is the author of numerous books on software development, including best-selling titles on C++ and .NET. He is the president of Liberty Associates, Inc. (www.libertyassociates.com), where he provides custom programming, consulting, and training.

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.

We welcome your comments. You can email or write 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 better.

Please note that we cannot help you with technical problems related to the topic of *this book.*

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
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

Congratulations! By reading this sentence, you are already 20 seconds closer to learning C++, one of the most important programming languages in the world.

If you continue for another 23 hours, 59 minutes, and 40 seconds, you will master the fundamentals of the C++ programming language. Twenty-four one-hour lessons cover important features such as managing I/O, creating loops and arrays, using object-oriented programming with templates, and creating C++ programs.

All of this has been organized into well-structured, easy-to-follow lessons. There are working projects that you create—complete with output and an analysis of the code—to illustrate the topics of the hour. Syntax examples are clearly marked for handy reference.

To help you become more proficient, each hour ends with a set of common questions and answers.

Who Should Read This Book?

You don't need any previous experience in programming to learn C++ with this book. It starts with the basics and teaches you both the language and the concepts involved with programming C++. Whether you are just beginning or already have some experience programming, you will find that this book makes learning C++ fast and easy.

Should I Learn C First?

No, you don't need to learn C first. C++ is a much more powerful and versatile language that was created by Bjarne Stroustrup as a successor to C. Learning C first can lead you into some programming habits that are more error-prone than what you'll do in C++. This book does not assume that readers are familiar with C.

Why Should I Learn C++?

You could be learning a lot of other languages, but C++ is valuable to learn because it has stood the test of time and continues to be a popular choice for modern programming.

Despite being created in 1979, C++ is still being used for professional software today because of the power and flexibility of the language. There's even a new version of the language, called C++14, that makes it even more useful.

Because other languages such as Java were inspired by C++, learning the language can provide you insight into them, as well. Mastering C++ gives you portable skills that you can use on just about any platform on the market today, from desktop computers to Linux servers, mobile devices, videogame consoles, and mainframes.

What If I Don't Want This Book?

I'm sorry you feel that way, but these things happen sometimes. Please reshelve this book with the front cover facing outward on an endcap with access to a lot of the store's foot traffic.

Conventions Used in This Book

This book contains special elements as described here.

NOTE

These boxes provide additional information to the material you just read.

CAUTION

These boxes focus your attention on problems or side effects that can occur in specific situations.

TIP

These boxes give you tips and highlight information that can make your C++ programming more efficient and effective.

When you see this symbol, you know that what you see next will show the output from a code listing/example.

This book uses various typefaces:

- To help you distinguish C++ code from regular English, actual C++ code is typeset in a special `monospace` font.
- Placeholders—words or characters temporarily used to represent the real words or characters you would type in code—are typeset in *`italic monospace`*.
- New or important terms are typeset in *italic*.
- In the listings in this book, each real code line is numbered. If you see an unnumbered line in a listing, you'll know that the unnumbered line is really a continuation of the preceding numbered code line (some code lines are too long for the width of the book). In this case, you should type the two lines as one; do not divide them.

This page intentionally left blank

HOUR 2

Organizing the Parts of a Program

What You'll Learn in This Hour:

- Why to use C++
- How C++ programs are organized
- How comments make programs easier to understand
- What functions can accomplish

Although it recently turned 37, the C++ programming language has aged a lot better than some other things that came out in the late 1970s. Unlike disco, oil embargoes, shag carpet, and avocado-colored refrigerators, C++ is still in vogue today. It remains a world-class programming language.

The reason for its surprising longevity is that C++ makes it possible to create fast executing programs with a small amount of code that can run on a variety of computing environments. Today's C++ programming tools enable the creation of complex and powerful applications in commercial, business, and open source development.

Reasons to Use C++

During the seven decades of the computing age, computer programming languages have undergone a dramatic evolution. C++ is considered to be an evolutional improvement of a language called C that was introduced in 1972.

The earliest programmers worked with the most primitive computer instructions: machine language. These instructions were represented by long strings of ones and zeroes. Assemblers were devised that could map machine instructions to human-readable and manageable commands such as `ADD` and `MOV`.

The instructions that make up a computer program are called its *source code*.

In time, higher-level languages were introduced such as BASIC and COBOL. These languages made it possible for programmers to begin to craft programs using language closer to actual

words and sentences, such as `Let Average = .366`. These instructions were translated back into machine language by tools that were called either *interpreters* or *compilers*.

An interpreter-based language translates a program as it reads each line, acting on each instruction.

A compiler-based language translates a program into what is called *object code* through a process called *compiling*. This code is stored in an object file. Next, a linker transforms the object file into an executable program that can be run on an operating system.

Because interpreters read the code as it is written and execute the code on the fly, they're easy for programmers to work with. Compilers require the more inconvenient extra steps of compiling and linking programs. The benefit to this approach is that the programs run significantly faster than programs run by an interpreter.

For many years, the principal goal of programmers was to write short pieces of code that would execute quickly. Programs needed to be small because memory was expensive, and they needed to be fast because processing power also was expensive. As computers have become cheaper, faster, and more powerful and the cost and capacity of memory has fallen, these priorities diminished in importance.

Today, the greatest expense in programming is the cost of a programmer's time. Modern languages such as C++ make it faster to produce well-written, easy-to-maintain programs that can be extended and enhanced.

Styles of Programming

As programming languages have evolved, languages have been created to cater to different styles of programming.

In procedural programming, programs are conceived of as a series of actions performed on a set of data. Structured programming was introduced to provide a systematic approach to organizing these procedures and managing large amounts of data.

The principal idea behind structured programming is to divide and conquer. Take a task that needs to be accomplished in a program, and if it is too complex, break it down into a set of smaller component tasks. If any of those tasks is still too complicated, break it down into even smaller tasks. The end goal is tasks that are small and self-contained enough to be easily understood.

As an example, pretend you've been asked by this publisher to write a program that tracks the average income of its team of enormously talented and understatedly charismatic computer book authors. This job can be broken down into these subtasks:

1. Find out what each author earns.
2. Count how many authors the publisher has.

3. Total all their income.
4. Divide the total by the number of authors.

Totaling the income can be broken down into the following:

1. Get each author's personnel record.
2. Access the author's book advances and royalties.
3. Deduct the cost of morning coffee, corrective eyewear and chiropractic care.
4. Add the income to the running total.
5. Get the next author's record.

In turn, obtaining each author's record can be broken down into these subtasks:

1. Open the file folder of authors.
2. Go to the correct record.
3. Read the data from disk.

Although structured programming has been widely used, this approach has some drawbacks. The separation of data from the tasks that manipulate the data becomes harder to comprehend and maintain as the amount of data grows. The more things that must be done with data, the more confusing a program becomes.

Procedural programmers often find themselves reinventing new solutions to old problems instead of producing reusable programs. The idea behind reusability is to build program components that can be plugged into programs as needed. This approach is modeled after the physical world, where devices are built out of individual parts that each perform a specific task and have already been manufactured. A person designing a bicycle doesn't have to create a brake system from scratch. Instead, she can incorporate an existing brake into the design and take advantage of its existing functionality.

This component-based approach became available to computer programmers for the first time with the introduction of object-oriented programming.

C++ and Object-Oriented Programming

C++ helped popularize a revolutionary style of programming with a funny acronym: OOP.

The essence of object-oriented programming is to treat data and the procedures that act upon the data as a single object—a self-contained entity with an identity and characteristics of its own.

The C++ language fully supports object-oriented programming, including three concepts that have come to be known as the pillars of object-oriented development: encapsulation, inheritance, and polymorphism.

Encapsulation

When the aforementioned bike engineer creates a new bicycle, she connects together component pieces such as the frame, handlebars, wheels, and a headlight (baseball card in the spokes optional). Each component has certain properties and can accomplish certain behaviors. She can use the headlight without understanding the details of how it works, as long as she knows what it does.

To achieve this, the headlight must be self-contained. It must do one well-defined thing and it must do it completely. Accomplishing one thing completely is called *encapsulation*.

All the properties of the headlight are encapsulated in the headlight object. They are not spread out through the bicycle.

C++ supports the properties of encapsulation through the creation of user-defined types called *classes*. A well-defined class acts as a fully encapsulated entity that is used as an entire unit or not at all. The inner workings of the class should be hidden on the principle that the programs which use a well-defined class do not need to know how the class works. They only need to know is how to use it. You learn how to create classes in Hour 8, "Creating Basic Classes."

Inheritance and Reuse

Now we're starting to learn a little more about our bike engineer. Let's call her Penny Farthing. Penny needs her new bicycle to hit the market quickly—she has run up enormous gambling debts to people who are not known for their patience.

Because of the urgency, Penny starts with the design of an existing bicycle and enhances it with cool add-ons like a cup holder and mileage counter. Her new bicycle is conceived as a kind of bicycle with added features. She reused all the features of a regular bicycle while adding capabilities to extend its utility.

C++ supports the idea of reuse through inheritance. A new type can be declared that is an extension of an existing type. This new subclass is said to derive from the existing type. Penny's bicycle is derived from a plain old bicycle and thus inherits all its qualities but adds additional features as needed. Inheritance and its application in C++ are discussed in Hour 16, "Extending Classes with Inheritance."

Polymorphism

As its final new selling point, Penny Farthing's Amazo-Bicycle behaves differently when its horn is squeezed. Instead of honking like an anguished duck, it sounds like a car when lightly pressed

and roars like a foghorn when strongly squashed. The horn does the right thing and makes the proper sound based on how it is used by the bicycle's rider.

C++ supports this idea that different objects do the right thing through a language feature called *function polymorphism* and *class polymorphism*. *Polymorphism* refers to the same thing taking many forms, and is discussed during Hour 17, "Using Polymorphism and Derived Classes."

You will learn the full scope of object-oriented programming by learning C++. These concepts will become familiar to you by the time you've completed this full 24-hour ride and begun to develop your own C++ programs.

Disclaimer: You won't learn how to design bicycles or get out of gambling debt.

The Parts of a Program

The program you created during the first hour, `Motto.cpp`, contains the basic framework of a C++ program. Listing 2.1 reproduces the source code of this program so that it can be explored in more detail.

When typing this program in to a programming editor such as NetBeans, remember not to include the line numbers in the listing. They are included solely for the purpose of referring to specific lines in this book.

LISTING 2.1 The Full Text of `Motto.cpp`

```
1:  #include <iostream>
2:
3:  int main()
4:  {
5:      std::cout << "Solidum petit in profundis!\n";
6:      return 0;
7:  }
```

This program produces a single line of output, the motto of Aarhus University:

```
Solidum petit in profundis!
```

On line 1 of Listing 2.1 a file named `iostream` is included in the source code. This line causes the compiler to act as if the entire contents of that file were typed at that place in `Motto.cpp`.

Preprocessor Directives

A C++ compiler's first action is to call another tool called the preprocessor that examines the source code. This happens automatically each time the compiler runs.

The first character in line 1 is the # symbol, which indicates that the line is a command to be handled by the preprocessor. These commands are called *preprocessor directives*. The preprocessor's job is to read source code looking for directives and modify the code according to the indicated directive. The modified code is fed to the compiler.

The preprocessor serves as an editor of code right before it is compiled. Each directive is a command telling that editor what to do.

The `#include` directive tells the preprocessor to include the entire contents of a designated filename at that spot in a program. As you learned in Hour 1, "Writing Your First Program," C++ includes a standard library of source code that can be used in your programs to perform useful functionality. The code in the `iostream` file supports input and output tasks such as displaying information onscreen and taking input from a user.

The < and > brackets around the filename `iostream` tell the preprocessor to look in a standard set of locations for the file. Because of the brackets, the preprocessor looks for the `iostream` file in the folder that holds header files for the compiler. These files also are called *include files* because they are included in a program's source code.

The full contents of `iostream` are included in place of line 1.

NOTE

Header files traditionally ended with the filename extension `.h` and also were called *h file*s, so they used a directive of the form `include <iostream.h>`.

Modern compilers don't require that extension, but if you refer to files using it, the directive might still work for compatibility reasons. This book omits the extraneous .h in include files.

The contents of the file `iostream` are used by the `cout` command in line 5, which displays information to the screen.

There are no other directives in the source code, so the compiler handles the rest of `Motto.cpp`.

Source Code Line by Line

Line 3 begins the actual program by declaring a function named `main()`. *Functions* are blocks of code that perform one or more related actions. Functions do some work and then return to the spot in the program where they were called.

Every C++ program has a `main()` function. When a program starts, `main()` is called automatically.

All functions in C++ must return a value of some kind after their work is done. The `main()` function always returns an integer value. Integers are specified using the keyword `int`.

Functions, like other blocks of code in a C++ program, are grouped together using the brace marks { and }. All functions begin with an opening brace { and end with a closing brace }.

The braces for the `main()` function of `Motto.cpp` are on lines 4 and 7, respectively. Everything between the opening and closing braces is part of the function.

In line 5, the `cout` command is used to display a message on the screen. The object has the designation `std::` in front of it, which tells the compiler to use the standard C++ input/output library. The details of how this works are too complex for this early hour and likely will cause you to throw the book across the room if introduced here. For the safety of others in your vicinity, they are explained later. For now, treat `std::cout` as the name of the object that handles output in your programs and `std::cin` as the object that handles user input.

The reference to `std::cout` in line 5 is followed by `<<`, which is called the *output redirection operator*. *Operators* are characters in lines of code that perform an action in response to some kind of information. The `<<` operator displays the information that follows it on the line. In line 5, the text `"Solidum petit in profundis!\n"` is enclosed within double quotes. This displays a string of characters on the screen followed by a special character specified by `"\n"`, a newline character that advances the program's output to the beginning of the next line.

On line 6, the program returns the integer value 0. This value is received by the operating system after the program finishes running. Typically, a program returns the value 0 to indicate that it ran successfully. Any other number indicates a failure of some kind.

The closing braces on line 7 ends the `main()` function, which ends the program. All of your programs use the basic framework demonstrated by this program.

Comments

As you are writing your own programs for the first time, it will seem perfectly clear to you what each line of the source code does. But as time passes and you come back to the program to fix a bug or add a new feature, you may find yourself completely mystified by your own work.

To avoid this predicament and help others understand your program, you can document your source code with comments. *Comments* are lines of text that explain what a program is doing. The compiler ignores them, so they are strictly for benefit of humans reading the code.

There are two types of comments in C++. A single-line comment begins with two slash marks (`//`) and causes the compiler to ignore everything that follows the slashes on the same line. Here's an example:

```
// The next line is a kludge (ugh!)
```

A multiple-line comment begins with the slash and asterisk characters (`/*`) and ends with the same characters reversed (`*/`). Everything within the opening `/*` and the closing `*/` is a comment, even if it stretches over multiple lines. If a program contains a `/*` that is not followed by a `*/` somewhere, that's an error likely to be flagged by the compiler. Here's a multiline comment:

```
/* This part of the program doesn't work very well. Please remember to
   fix this before the code goes live -- or else find a scapegoat you can
   blame for the problem. The new guy Curtis would be a good choice. */
```

In the preceding comment, the text on the left margin is lined up to make it more readable. This is not required. Because the compiler ignores everything within the `/*` and `*/`, anything can be put there—grocery lists, love poems, secrets you've never told anybody in your life, and so on.

CAUTION

An important thing to remember about multiline comments is that they do not nest inside each other. If you use one `/*` to start a comment and then use another `/*` a few lines later, the first `*/` mark encountered by the compiler will end all multiline comments. The second `*/` mark will result in a compiler error. Most C++ programming editors display comments in a different color to make clear where they begin and end.

The next project that you create includes both kinds of comments. Write lots of comments in your programs. The more time spent writing comments that explain what's going on in source code, the easier that code will be to work on weeks, months or even years later.

Functions

The `main()` function is unusual among C++ functions because it's called automatically when a program begins running.

A program is executed line by line in source code, beginning with the start of `main()`. When a function is called, the program branches off to execute the function. After the function has done its work, it returns control to the line where the function was called. Functions may or may not return a value, with the exception of `main()`, which always returns an integer.

Functions consist of a header and a body. The header consists of three things:

- The type of data the function returns
- The function's name
- The parameters received by the function

The *function name* is a short identifier that describes its purpose.

When a function does not return a value, it uses data type `void`, which means nothing. To clarify: `void` isn't meaningless. It means "nothing," like how stars in space are separated by a ginormous amount of nothing called "the void."

Arguments are data sent to the function that control what it does. These arguments are received by the function as *parameters*. A function can have zero, one, or more parameters. The next program that you create has a function called `add()` that adds two numbers together. Here's how it is declared:

```
int add(int x, int y)
{
    // body of function goes here
}
```

The parameters are organized within parentheses marks as a list separated by commas. In this function, the parameters are integers named `x` and `y`.

The name of a function, its parameters and the order of those parameters is called its *signature*. Like a person's signature, the function's signature uniquely identifies it.

A function with no parameters has an empty set of parentheses, as in this example:

```
int getServerStatus()
{
    // body of function here
}
```

Function names cannot contain spaces, so the `getServerStatus()` function capitalizes the first letter of each word after the first one. This naming convention is common among C++ programmers and adopted throughout this book.

The body of a function consists of an opening brace, zero or more statements, and a closing brace. A function that returns a value uses a `return` statement, as you've seen in the Motto program:

```
return 0;
```

The `return` statement causes a function to exit. If you don't include at least one `return` statement in a function, it automatically returns `void` at the end of the function's body. In that situation, `void` must be specified as the function's return type.

Using Arguments with Functions

The `Calculator.cpp` program in Listing 2.2 fleshes out the aforementioned `add()` function, using it to add a pair of numbers together and display the results. This program demonstrates how to create a function that takes two integer arguments and returns an integer value.

LISTING 2.2 The Full Text of `Calculator.cpp`

```
 1: #include <iostream>
 2:
 3: int add(int x, int y)
 4: {
 5:     // add the numbers x and y together and return the sum
 6:     std::cout << "Running calculator ...\n";
 7:     return (x+y);
 8: }
 9:
10: int main()
11: {
12:     /* this program calls an add() function to add two different
13:        sets of numbers together and display the results. The
14:        add() function doesn't do anything unless it is called by
15:        a line in the main() function. */
16:     std::cout << "What is 867 + 5309?\n";
17:     std::cout << "The sum is " << add(867, 5309) << "\n\n";
18:     std::cout << "What is 777 + 9311?\n";
19:     std::cout << "The sum is " << add(777, 9311) << "\n";
20:     return 0;
21: }
```

This program produces the following output:

```
What is 867 + 5309?
Running calculator ...
The sum is 6176

What is 777 + 9311?
Running calculator ...
The sum is 10088
```

The `Calculator` program includes a single-line comment on line 5 and a multi-line comment on lines 12–15. All comments are ignored by the compiler.

The `add()` function takes two integer parameters named `x` and `y` and adds them together in a `return` statement (lines 3–8).

The program's execution begins in the `main()` function. The first statement in line 16 uses the object `std::cout` and the redirection operator `<<` to display the text `"What is 867 + 5309?"` followed by a newline.

The next line displays the text `"The sum is"` and calls the `add()` function with the arguments `777` and `9311`. The execution of the program branches off to the `add()` function, as you can tell in the output by the text `"Running calculator...."`

The integer value returned by the function is displayed along with two more newlines.

The process repeats for a different set of numbers in lines 18–19.

The formula (`x+y`) is an expression. You learn how to create these mathematical workhorses in Hour 4, "Using Expressions, Statements, and Operators."

Summary

During this hour, you were shown how C++ evolved from other styles of computer languages and embraced a methodology called object-oriented programming. This methodology has been so successful in the world of computing that the language remains as contemporary today as it did when it was invented in 1979.

I wish the mullet haircut I sported in college had survived the test of time as well. Instead, it lives on in Facebook photos that friends share to shock and awe.

In the two programs that you developed during this hour, you made use of three parts of a C++ program: preprocessor directives, comments, and functions.

All the programs that you will create in C++ employ the same basic framework as the Motto and Calculator programs. They just become more sophisticated as they make use of more functions, whether you write them from scratch or call functions from header files included with the `#include` directive.

Q&A

Q. What does the `#` character do in a C++ program?

A. The `#` symbol signals that the line is a preprocessor directive, a command that is handled before the program is compiled. The `#include` directive includes the full text of a file at that position in the program. The compiler never sees the directive. Instead, it acts as if the contents of the file were typed in with the rest of the source code.

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

A. The comments that start with `//` are single-line comments that end with the end of the line on which they appear. The `/*` comments are multi-line comments that don't end until a `*/` is encountered. The end of a function won't even cause a multi-line comment to be ended. You must put in the closing `*/` mark or the compiler will fail with an error.

Q. What's the difference between function arguments and function parameters?

A. The terms are two sides of the same process when a function is called. Arguments are the information sent to the function. Parameters are the same information received by the function. You call a function with arguments. Within a function, those arguments are received as parameters.

Q. What is a kludge?

A. A kludge is an ugly solution to a problem that's intended to be replaced later with something better. The term was popularized by Navy technicians, computer programmers, and aerospace engineers and spread to other technical professions.

In a computer program a kludge is source code that works but would have been designed better if there had been more time. Kludges have a tendency to stick around a lot longer than expected.

The astronauts on the Apollo 13 mission created one of the greatest kludges of all time: a system cobbled together from duct tape and socks that filtered carbon dioxide from the air on the spacecraft and helped them make it back to Earth.

The first known usage of the term was in a 1962 article in *Datamation* magazine by Jackson W. Granholm, who gave it an elegant definition that has stood the test of time: "An ill-assorted collection of poorly matching parts, forming a distressing whole."

Workshop

Now that you've learned about some of the pieces of a C++ program, you can answer a couple of questions and complete a couple of exercises to firm up your knowledge.

Quiz

1. What data type does the `main` function return?

   A. `void`

   B. `int`

   C. It does not return a type.

2. What do the braces do in a C++ program?

   A. Indicate the start and end of a function

   B. Indicate the start and end of a program

   C. Straighten the program's teeth

3. What is not part of a function's signature?

   A. Its name

   B. Its arguments

   C. Its return type

Answers

1. **B.** The `main` function returns an `int` (integer).
2. **A.** Braces mark the start and end of functions and other blocks of code you learn about in upcoming hours.
3. **C.** A function signature consists of its name, parameters, and the precise order of those parameters. It does not include its return type.

Activities

1. Rewrite the Motto program to display the Aarhus University motto in a function.
2. Rewrite the Calculator program to add a third integer called `z` in the `add()` function and call this function with two sets of three numbers.

To see solutions to these activities, visit this book's website at http://cplusplus.cadenhead.org.

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

J-K

L

M

N

O

P

Q-R

S

T

U

V

W-X-Y-Z