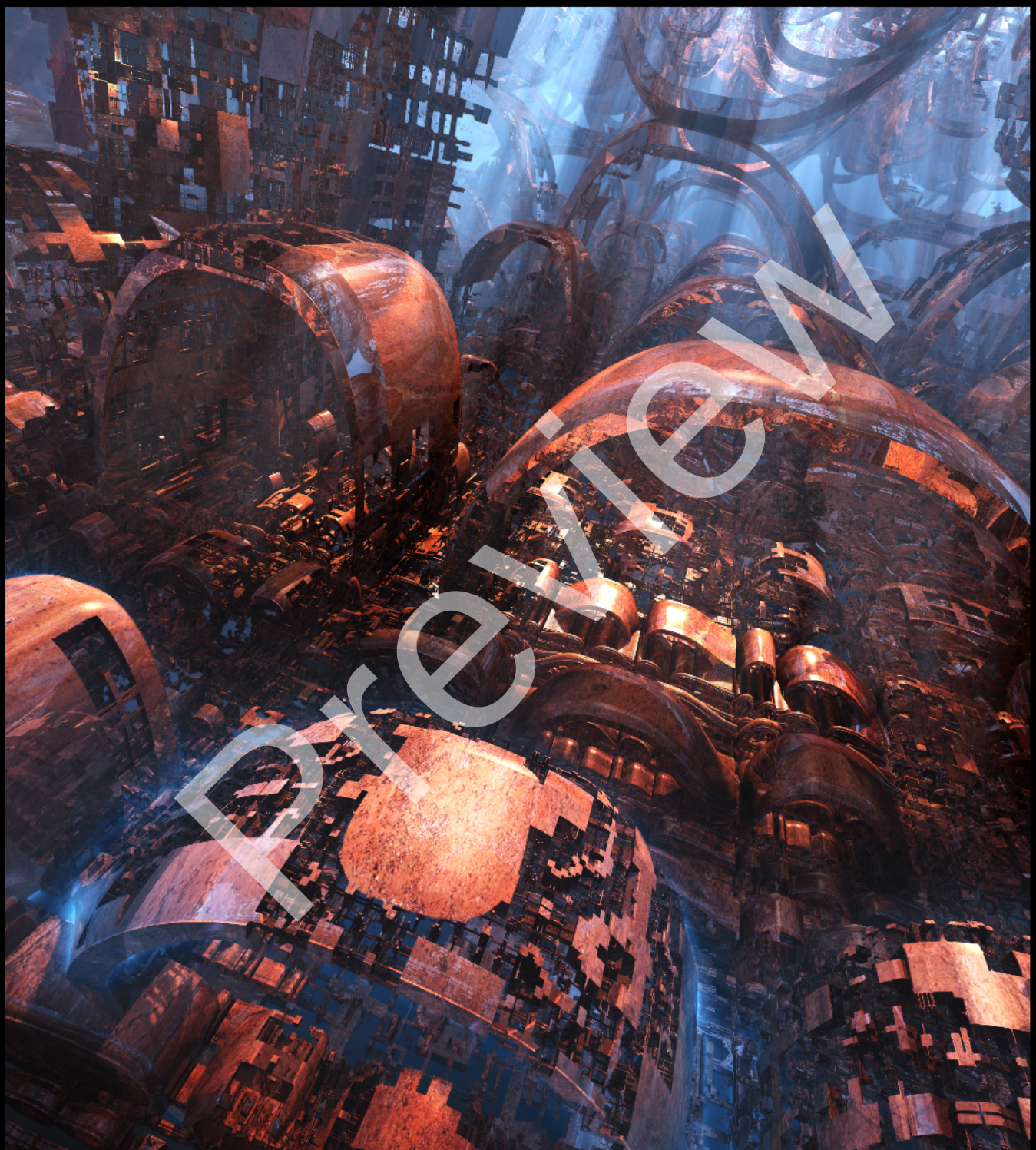
C++ Data Structures from Scratch, Vol. 1



Robert MacGregor

From databases and operating systems to simulations and graphics, data structures are an integral part of all programming domains. Designed for complete beginners as well as those with prior programming experience, *C++ Data Structures from Scratch, Vol. 1* covers everything, from basic C++ language concepts, to creating fully functional, STL-style implementations of common data structures and sorting algorithms.

Key features:

- 170+ complete source code files, with detailed line-by-line analysis and diagrams
- 60 sample programs directly illustrating key concepts from each chapter
- Free sample content and online support at the official website, *cppdatastructures.com*

Major topics:

- Step-by-step instructions for setting up an IDE (integrated development environment)
- Thorough coverage of fundamental C++ language concepts:
    - Datatypes, variables, and arithmetic
    - Logic, functions, and program structure
    - Pointers, arrays, and references
    - Object-oriented programming (classes) and operator overloading
    - Template metaprogramming, in the style of the Standard Template Library (STL)
    - Dynamic memory allocation
- A comprehensive, line-by-line guide to implementing:
    - Bubble sort, insertion sort, and quicksort
    - Allocators
    - Dynamic arrays (STL *vector*)
    - Doubly-linked lists (STL *list*)
    - Single-block deques (double-ended queues)
    - Multi-block deques (STL *deque*)
    - Non-balancing binary search trees
    - AVL (self-balancing) trees (STL *map*)
    - Bidirectional / random access, *const*, and reverse iterators for each data structure

About the author:

- Robert MacGregor is the developer of a C++ API for financial market trading systems. He is also a CTA (Commodity Trading Advisor) in the National Futures Association, and a Chartered Market Technician in the CMT Association.

Cover illustration: *SelfSimilarCircuitry* by Mark J. Brady (*markjaybeefractal.com*)

C++ Data Structures from Scratch, Vol. 1

Robert MacGregor

Published by South Coast Books

For errata, supplementary material, and contact / purchase information, visit www.cppdatastructures.com

Cover illustration: *SelfSimilarCircuitry* by Mark J. Brady (www.markjaybeefractal.com)

ISBN-10: 0-9962115-8-6
ISBN-13: 978-0-9962115-8-1

1st Printing, November 2023

Dedicated to Milagros "Mila" Oronce Reyes
(1935-2014)

Table of Contents

Part 6: Classes and Operator Overloading

Part 7: Templates and Function Objects

Part 8: Dynamic Memory Allocation

Part 9: Dynamic Arrays

Part 10: Linked Lists

Part 11: Reverse Iterators

Part 12: Single-Block Double-Ended Queues

Part 13: Multi-Block Double-Ended Queues

Part 14: Non-Balancing Binary Search Trees

Part 15: Self-Balancing Binary Search Trees

Part 16: Time Complexity

Appendix: Standard Library Equivalents

Index

Introduction and Getting Started

Chapter outline

- *What are data structures, and why do we need to know how they work?*
- *Who this book is for*
- *A brief overview of the major topics*

The manipulation of data is a fundamental task performed by nearly all types of software. An address book application, for example, alphabetically sorts names, while a web browser maintains a chronological history of visited sites. A *data structure* is a software component that stores and organizes a set of data.

Although different types of data structures can perform many of the same tasks, they do so with varying degrees of efficiency. By knowing how data structures work, we can better understand their relative strengths and weaknesses, thereby helping us design more effective software.

No prior programming experience is required for this book. We'll begin by setting up the software required to create C++ programs, and introducing the most basic building blocks of the language. By mastering the material herein, you'll acquire a strong foundation in data structures and C++, as well as a solid conceptual framework for programming in a wide variety of languages and domains.

Here's a brief overview of what we'll cover in Volume 1:

In Part 1, we'll set up a *development environment* (software used to create C++ programs), and obtain the required *source code* for all of the programs in this book. We'll then create our first program, introducing *variables* and *datatypes*.

In Part 2, we'll cover basic *arithmetic operations*, such as addition, subtraction, multiplication, and division.

In Part 3, we'll introduce basic tools for managing *control flow*, the order in which a program performs its instructions.

In Part 4, we'll introduce basic tools for organizing and reusing source code, such as *functions*, *namespaces*, and *header files*.

In Part 5, we'll introduce the concept of *indirection*, covering *pointers*, *arrays*, and *references*. To demonstrate these tools, we'll implement the *bubble sort* and *insertion sort algorithms*.

In Part 6, we'll introduce *object-oriented programming*, *classes*, and *operator overloading*.

In Part 7, we'll introduce *template metaprogramming* and *function objects*. We'll also implement the *quicksort algorithm*, introducing the concept of *recursion*.

In Part 8, we'll introduce *dynamic memory allocation*, completing the foundation for Parts 9-15.

In Parts 9-15, we'll implement a fully-functional set of common data structures:

- Dynamic array (STL *vector*)
- Doubly-linked list (STL *list*)
- Single-block deque (double-ended queue)
- Multi-block deque (STL *deque*)
- Non-balancing binary search tree
- AVL tree (self-balancing binary search tree) (STL *map*)

For each of these data structures, we'll also implement the applicable *iterators* (bidirectional / random access, *const*, and reverse).

In Part 16, we'll introduce the concept of *time complexity*, and demonstrate how to optimize a single function for multiple types of iterators.

C++ Data Structures from Scratch, Vol. 1

Part 1: Creating Our First Program

1.1: Setting Up a Development Environment

Chapter outline

- *Obtaining and setting up the tools required to create C++ programs*

Creating a C++ program requires three tools: a word processor, compiler, and linker.

A *word processor* is a program used to edit plain text (*.txt*) files, such as Microsoft Notepad or Notepad++. We use the word processor to write *source code*, a set of plain text files with the extensions *.h* and *.cpp*. The source code, written in C++, contains the instructions to be performed by our program.

Computers, however, don't natively understand the C++ language; the source code must be translated into *object code* (*.obj* files). Object code is written in *machine language*, the native instruction set of the *CPU* (central processing unit / microchip). To perform this translation, we use a program called a *compiler.*

Finally, we use a program called a *linker*, which combines the various object (*.obj*) files into a single *executable* (*.exe* file). The executable is our finished program, which we can run from the command line or desktop.

The following diagram summarizes the entire process:

```
Source Code (.h and .cpp files, written in C++)
     |
     |      (Compilation)
     v
Object Code (.obj files, written in machine language)
     |
     |      (Linking)
     v
Executable  (.exe file, finished program)
```

Rather than use a separate program for each step, we can streamline the process by using an *IDE* (integrated development environment). An IDE is a single program that combines the functionality of a word processor, compiler, and linker. This allows us to write source code, then generate the executable with a single click.

To download, install, and set up the IDE, follow the instructions at *www.cppdatastructures.com* before proceeding.

1.2: Obtaining the Required Source Code

Chapter outline

- *Obtaining the required source code for all of the programs in this book*
- *How to use the source code listing for each chapter*
- *Recommended study approach*

Each chapter of this book is a line-by-line walk-through of a small program, designed to illustrate the key concepts as simply and directly as possible.

To obtain the source code for all of these programs, visit *www.cppdatastructures.com* before proceeding.

The relevant source files and / or folders are listed at the beginning of each chapter. The root folder (*dss*) is omitted. If a folder is listed without specific filenames, it indicates that the chapter uses all of the files in that folder. The listing for Chapter 5.2, for example,

> *Source files and folders*
>
> - *passByReference*
> - *swapInts*

indicates that Chapter 5.2 uses:

- All of the files in the folder *dss/passByReference*
- All of the files in the folder *dss/swapInts*

Similarly, the listing for Chapter 7.3,

> *Source files and folders*
>
> - *Array/1*
> - *Array/common/memberFunctions_1.h*
> - *quickSort*

indicates that Chapter 7.3 uses:

- All of the files in the folder *dss/Array/1*
- The file *dss/Array/common/memberFunctions_1.h*
- All of the files in the folder *dss/quickSort*

The recommended study approach is as follows:

- At the beginning of each chapter, compile the included source code and run the program.
- Read the chapter, following along with the included source code.
- Read the chapter again, duplicating the included source code from scratch.
- Compile the duplicated source code and run the program, verifying that you've achieved the same result.

1.3: Standard Output, Variables, and Datatypes

Source files and folders

- *hello*

Chapter outline

- *Generating an executable*
- *Displaying messages on the screen*
- *Using variables to store and modify values*
- *Basic datatypes: integers, real numbers, booleans, and character strings*

Our first program will introduce some basic building blocks of the C++ language. Before analyzing the source code, however, we'll generate and run the executable. This will allow us to view the source code and its results, side-by-side.

For instructions on how to generate an executable, visit *www.cppdatastructures.com* before proceeding.

Now that we've run the program, let's examine the source code. Line 4 of *main.cpp*,

```
int main()
```

specifies a function called *main*. A *function*, also called a *subroutine* or *method*, is a named set of instructions, or *statements*.

A function's statements are collectively referred to as the *function body*, which is enclosed in *curly braces* ({ }). The opening and closing braces of *main* are located at lines 5 and 36, respectively. When the program is run, the system *calls* (executes) the function *main*, running through each statement in the body (lines 6-35).

The type of output value generated by a function is called its *return type*. The return type of *main* is *int* (line 4), which is short for *integer.* This indicates that, upon completion, *main* returns an integer value to the *function caller* (the executor of the function). Line 8,

```
cout << "Hello" << endl;
```

displays the message

```
Hello
```

on the screen, where

- *cout* (pronounced "C out") is the *standard output stream*, used to display messages.

- << is the *stream insertion operator*, used to indicate what we want to display.

- *endl* (pronounced "end L") is short for "end line." This represents a *carriage return*, a special character used to indicate the end of the current line. An *endl*, in other words, indicates that the next message displayed will begin on a new line.

The entire statement ("C out *Hello*, end L") thus inserts the character string *Hello* into the standard output stream, followed by a carriage return. The semicolon following *endl* indicates the end of the statement.

A *variable* is a value that we can refer to by name. Lines 10-14,

```
int i = 0;
double d = 3.14;
bool b = true;
char c = '?';
string s = "Independence Day";
```

declare and initialize the variables

- *i*, of type *int* (integer), initialized to *0*
- *d*, of type *double* (real number), initialized to *3.14*
- *b*, of type *bool* (*boolean*, true-or-false value), initialized to *true*
- *c*, of type *char* (single character), initialized to *?*
- *s*, of type *string* (character string), initialized to *Independence Day*

Note that values of type *char* must be enclosed in single-quotation marks (line 13). Values of type *string* must be enclosed in double-quotation marks (line 14). Line 16,

```
cout << "i is " << i << endl;
```

prints the character string "i is ", followed by the current value of *i*, followed by a carriage return, generating the output

```
i is 0
```

Similarly, lines 17-20,

```
cout << "d is " << d << endl;
cout << "b is " << b << endl;
cout << "c is " << c << endl;
cout << "s is " << s << endl << endl;
```

generate the output

```
d is 3.14
b is 1
c is ?
```

```
s is Independence Day
```

Note that, when using *cout* to print values of type *bool* (as in line 18), *true* and *false* are displayed as the integer values *1* and *0*, respectively. In lines 22-26,

```
i = 9;
d = 2.71;
b = false;
c = '!';
s = "July 4th, 1776";
```

we assign a new value to each variable. The equal sign (=) is called the *assignment operator.* Once again, note that we use single-quotation marks for values of type *char* (line 25), and double-quotation marks for values of type *string* (line 26). Lines 28-32,

```
cout << "i is " << i << endl;
cout << "d is " << d << endl;
cout << "b is " << b << endl;
cout << "c is " << c << endl;
cout << "s is " << s << endl;
```

print the value of each variable once more. The corresponding output,

```
i is 9
d is 2.71
b is 0
c is !
s is July 4th, 1776
```

reflects the new values. Line 33,

```
cout << "Goodbye\n";
```

is equivalent to

```
cout << "Goodbye" << endl;
```

where *\n* is a special single-character value, representing a carriage return. It can be used as part of a character string, or as a single-character value. Given the values

```
string k = "ice\ncream\ncone\n";
char p = '\n';
```

for example, the statements

```
cout << k;
cout << "ice" << p << "cream" << p << "cone" << p;
```

generate the output

```
ice
cream
cone
ice
cream
cone
```

As mentioned earlier, the return type of *main* is *int* (line 4). We must therefore terminate *main* by returning an integer value to the caller. Line 35,

```
return 0;
```

terminates *main*, returning a value of 0. Line 38,

```
// single-line comment
```

demonstrates the syntax of a single-line comment. A *comment* is a portion of text to be ignored by the compiler, used to further explain or document the code. All text following a pair of forward slashes (//) to the end of the line will be ignored by the compiler. Lines 40-43,

```
/*
  multiple-line
  comment
*/
```

demonstrate the syntax of a multiple-line comment. In this case, the compiler ignores all text from the /* to the */.

cout, *endl*, and the *string* type are provided by the *C++ Standard Library*, a uniform set of tools often included with compilers and IDEs. In order to use Standard Library components, the appropriate *header files* must be included. Lines 1-2,

```
#include <iostream>
#include <string>
```

are *include directives*. Before compilation, a separate program, called the *preprocessor*, replaces each include directive with the text of the specified file, as if we had written it ourselves. In this case, the preprocessor inserts the text of <*iostream*> at line 1, followed by the the text of <*string*>. The <*iostream*> header provides *cout* and *endl*, while the <*string*> header provides the *string* type.

The statement in line 6,

```
using namespace std;
```

is a *using directive*. Without this statement, we would've had to write every instance of *cout*, *endl*, and *string* (lines 8, 14, 16-20, 28-33) as *std::cout*, *std::endl*, and *std::string*. Lines 8 and 14, for example, would've had to be written as

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

```
std::string s = "Independence Day";
```

We'll discuss the meaning of *namespace* and *std* in Part 4. For now, all you need to know is that the statement

```
using namespace std;
```

allows us to omit the *std::* when referring to Standard Library components, such as *cout*, *endl*, and *string*.

Before moving on, there's an additional point to note regarding variables. We can declare a variable, without initializing it to a specific value. We could have, for example, written lines 10-14 as

```
int i;
double d;
bool b;
char c;
string s;
```

In this case, the initial values of *i*, *d*, *b*, and *c* would be random, while the initial value of *s* would be *empty string* (a string containing zero characters). More generally speaking, when declaring a variable *v* without specifying its value,

- If *v* is an *int*, *double*, *bool*, or *char*, then *v* will be initialized to a random value.
- If *v* is a *string*, then *v* will be initialized as an empty string.

Part 2: Arithmetic Operations and User Input

2.1: Basic Arithmetic

Source files and folders

- *basicArithmetic*

Chapter outline

- *Performing addition, subtraction, multiplication, and division*
- *Calculating a remainder using modulus division*

In this chapter, we'll demonstrate how to perform some basic arithmetic. Our program (*main.cpp*) begins by initializing an integer *a* to a value of 1 (line 7). The next statement (line 8)

```
int b = a + 2;
```

then initializes another integer *b*, to the value of (*a + 2*). The current value of *a* is 1, so the expression

```
a + 2
```

returns (evaluates to) 3, which is then assigned to *b*. The plus sign (+) is called the *addition operator*. We then print the values of *a* and *b* (lines 10-11), generating the output

```
a = 1
b = 3
```

In line 13, the statement

```
a = 7 - b;
```

sets the value of *a* to (*7 – b*). The current value of *b* is 3, so the expression

```
7 - b
```

returns 4, which is then assigned to *a*. The minus sign (–) is called the *subtraction operator*. In lines 15-16, we print the values of *a* and *b*, generating the output

```
a = 4
b = 3
```

The statement in line 18,

```
b = a * 3;
```

sets the value of *b* to (*a * 3*). The current value of *a* is 4, so the expression

```
a * 3
```

returns 12, which is then assigned to *b*. The asterisk (*) is called the *multiplication operator.* In lines 20-21, we print the values of *a* and *b*, generating the output

```
a = 4
b = 12
```

In line 23, the statement

```
a = b / 4;
```

sets the value of *a* to (*b / 4*). The current value of *b* is 12, so the expression

```
b / 4
```

returns 3. The forward slash (/) is called the *division operator.* In lines 25-26, we print the values of *a* and *b*, generating the output

```
a = 3
b = 12
```

In line 28, the statement

```
b = a % 2;
```

sets the value of *b* to (*a % 2*). The percent sign (%) is called the *modulo operator.* The modulo operator performs *modulus division*, which finds the remainder when dividing one value by another. The current value of *a* is 3, so the expression

```
a % 2
```

returns the remainder of (*a / 2*), which is 1. This value is then assigned to *b*. In lines 30-31, we print the values of *a* and *b* once more, generating the output

```
a = 3
b = 1
```

To demonstrate another example of modulus division, we'll use the expression

```
27 % 8
```

which returns the remainder of (*27 / 8*). When dividing 27 by 8, the remainder is 4, so the expression (*27 % 8*) returns 4.

2.2: Standard Input

Source files and folders

- *standardInput*

Chapter outline

- *Obtaining user input*

Our program (*main.cpp*) begins by declaring two *string*s, *firstName* and *lastName* (lines 8-9), and an *int age* (line 10). We then display the message (line 12)

```
Please enter your first name:
```

Line 13,

```
cin >> firstName;
```

prompts the user to enter a *string* value, and assigns it to the variable *firstName*, where

- *cin* (pronounced "C in") is the *standard input stream*, which obtains input from the user.
- >> is the *stream extraction operator*, used to specify the destination.

The entire statement ("C in, *firstname*") thus extracts data from the standard input stream and writes it to the variable *firstName*.

cin, like *cout*, is provided by the <*iostream*> header, which we've included in line 1.

We then ask the user for their last name and age. To do so, we begin by displaying the message (line 15)

```
Please enter your last name and age (separated by whitespace):
```

Used singularly, the term *whitespace* refers to a single empty space; plurally, it refers to a sequence of empty spaces. In the above message, we're therefore asking the user to separate their last name and age with one or more spaces.

Line 16,

```
cin >> lastName >> age;
```

prompts the user to enter a *string* and an *int* (separated by whitespace), and assigns them to *lastName* and *age*. This demonstrates how we can obtain multiple input values in a single statement, by writing >> before each destination variable.

Now that we've obtained the values for *firstName*, *lastName*, and *age*, we can perform some operations with the data. Line 18,

```
cout << "\nHello, " << firstName << " " << lastName << ".\n";
```

displays the message

```
Hello, <firstName> <lastName>.
```

where <*firstName*> and <*lastName*> will be the values of *firstName* and *lastName* respectively. This demonstrates how, when using *cout*, we can print multiple items in a single statement. Recall from Chapter 1.3 that *\n* is a special character, used to represent a carriage return. The expression (line 18)

```
cout << "\nHello, "
```

is therefore equivalent to

```
cout << endl << "Hello, "
```

The third item that we print in line 18 (after *firstName*),

```
" "
```

is a character string consisting of a single whitespace, which separates the values of *firstName* and *lastName* in the displayed message.

Line 19,

```
cout << "In 5 years, you will be " << age + 5 << " years old.\n";
```

generates the output

```
In 5 years, you will be <age + 5> years old.
```

where <*age + 5*> will be the value of (*age + 5*).

Given the values *Alex*, *Turner*, and *37*, for example, our program generates the output

```
Please enter your first name: Alex
Please enter your last name and age (separated by whitespace): Turner 37

Hello, Alex Turner.
In 5 years, you will be 42 years old.
```

2.3: Increment Operator

Source files and folders

- *incrementOperator*

Chapter outline

- *Prefix vs. postfix increment*

Incrementing a variable is the process of increasing its value by a given amount. Given

```
int x = 5;
```

for example, we can increment *x* by 1 via the statement

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

The current value of *x* is 5, so the expression

```
x + 1
```

returns 6, which is then assigned as the new value of *x*. Similarly, given

```
int y = 7;
```

we can increment *y* by 2 via the statement

```
y = y + 2;
```

The current value of *y* is 7, so the expression

```
y + 2
```

returns 9, which is then assigned as the new value of *y*.

Incrementing a variable by 1, as shown in the first example, is such a common operation that C++ provides a shorthand way of doing so. Our program (*main.cpp*) will demonstrate this.

We begin by initializing two *int*s, *a* and *b*, to 7 and 3 respectively (lines 7-11). We then print the values of *a* and *b* (lines 13-14), generating the output

```
a = 7
b = 3
```

In line 16, the statement

```
++a;
```

increments *a* by 1 (from 7 to 8), and is equivalent to

```
a = a + 1;
```

The ++ is called the *increment operator*, which increments a variable by 1. When the ++ appears before the target variable (as in line 16), it's called the *prefix increment operator*. After incrementing *a*, we print its new value (line 17), generating the output

```
a = 8
```

Line 19,

```
b = ++a;
```

demonstrates how the prefix increment operator behaves within a longer statement. In this statement, the expression

```
++a
```

increments *a* by 1 (from 8 to 9), and returns the *new* value of *a* (9), which we then assign to *b*. The value of *b* thus becomes 9. Printing the new values of *a* and *b* (lines 20-21) generates the output

```
a = 9
b = 9
```

When the ++ appears after the target variable, it's called the *postfix increment operator*. The statement in line 23, for example,

```
a++;
```

increments *a* by 1 (from 9 to 10). Printing the new value of *a* (line 24) generates the output

```
a = 10
```

Line 26,

```
b = a++;
```

demonstrates how the postfix increment operator behaves within a longer statement. In this statement, the expression

```
a++
```

increments *a* by 1 (from 10 to 11), and returns the *original* value of *a* (10), which we then assign to *b*. The value of *b* thus becomes 10. Printing the new values of *a* and *b* (lines 27-28) generates the output

```
a = 11
```

```
b = 10
```

Before moving on, it's worth reiterating the key difference between prefix and postfix increment:

- A prefix increment expression, such as (++*k*), increments *k*, and returns the *new* (incremented) value of *k*.

- A postfix increment expression, such as (*k*++), increments *k*, and returns the *original* (pre-increment) value of *k*.

The statement (line 19)

```
b = ++a;
```

for example, increments *a*, and sets *b* to the *new* (incremented) value of *a*. Conversely, the statement (line 26)

```
b = a++;
```

increments *a*, and sets *b* to the *original* (pre-increment) value of *a*. In *both* statements, the value of *a* increases by 1.

The complete output of our program (*main.cpp*) is

```
a = 7     // Initial values of a and b
b = 3

a = 8     // Result of ++a;

a = 9     // Result of b = ++a;
b = 9

a = 10    // Result of a++;

a = 11    // Result of b = a++;
b = 10
```

Knowing the subtle difference between prefix and postfix increment allows us to write more concise code. The statement (line 16)

```
b = ++a;
```

for example, is a shorthand way of writing

```
++a;
b = a;
```

Similarly, the statement (line 26)

```
b = a++;
```

is a shorthand way of writing

```
b = a;
a++;
```

Statements such as

```
b = ++a;
b = a++;
```

however, are more difficult to understand than their long-form counterparts. We'll therefore favor the long-form versions throughout this book.

As a sidenote, the increment operator provides some insight into the name of the C++ language. The ++ represents the fact that C++, created by Bjarne Stroustrup in 1982, is an extension of C, a language created by Dennis Ritchie in 1972.

2.4: Decrement Operator

Source files and folders

- *decrementOperator*

Chapter outline

- *Prefix vs. postfix decrement*

Decrementing a variable is the process of decreasing its value by a given amount. Given

```
int x = 5;
```

for example, we can decrement *x* by 1 via the statement

```
x = x - 1;
```

which sets *x* to 4. Similarly, given

```
int y = 7;
```

we can decrement *y* by 2 via the statement

```
y = y - 2;
```

which sets *y* to 5.

Given

```
int k = 3;
```

the statement

```
--k;
```

is equivalent to

```
k = k - 1;
```

The – – is called the *decrement operator*, which decrements a variable by 1. The decrement operator follows the same pattern as the increment operator:

- When the – – appears before the target variable (– –*k*), it's called the *prefix decrement operator*.
- When the – – appears after the target variable (*k*– –), it's called the *postfix decrement operator*.

- A prefix decrement expression, such as (– –*k*), decrements *k*, and returns the *new* (decremented) value of *k*.

- A postfix decrement expression, such as (*k*– –), decrements *k*, and returns the *original* (pre-decrement) value of *k*.

Our program (*main.cpp*) begins by initializing two integers, *a* and *b*, to 9 and 5 respectively (lines 7-8).

Line 13,

```
--a;
```

decrements *a* by 1 (from 9 to 8). After printing the new value of *a* (line 14), the statement (line 16)

```
b = --a;
```

decrements *a* by 1 (from 8 to 7), and sets *b* to the new value of *a* (7). After printing the new values of *a* and *b* (lines 17-18), the statement (line 20)

```
a--;
```

decrements *a* by 1 (from 7 to 6). After printing the new value of *a* (line 21), the statement (line 23)

```
b = a--;
```

decrements *a* by 1 (from 6 to 5), and sets *b* to the original value of *a* (6). We then print the new values of *a* and *b* once more (lines 24-25).

The complete output of our program (*main.cpp*) is

```
a = 9     // Initial values of a and b
b = 5

a = 8     // Result of --a;

a = 7     // Result of b = --a;
b = 7

a = 6     // Result of a--;

a = 5     // Result of b = a--;
b = 6
```

The statement (line 16)

```
b = --a;
```

is shorthand for

```
--a;
b = a;
```

Similarly, the statement (line 23)

```
b = a--;
```

is shorthand for

```
b = a;
a--;
```

As discussed in the previous chapter, statements such as

```
b = --a;
b = a--;
```

are more difficult to understand than their long-form counterparts. We'll therefore favor the long-form versions throughout this book.

2.5: Compound Assignment Operators

Source files and folders

- *compoundAssignment*

Chapter outline

- *Modifying variables using the compound assignment operators (addition, subtraction, multiplication, division, and modulo)*

Given (*main.cpp*, lines 7-8)

```
int a = 3;
int b = 5;
```

the statement (line 12)

```
a += 8;          // Increment the value of a by 8 (from 3 to 11)
```

is shorthand for

```
a = a + 8;
```

The += is called the *addition assignment operator*, which increments a variable by a given amount.

An addition assignment expression returns the target variable. The statement (line 15)

```
b = (a += 8);    // Increment the value of a by 8 (from 11 to 19),
                 // then set the value of b to the value of a (19)
```

for example, is equivalent to

```
a += 8;
b = a;
```

The statement (line 18)

```
a -= 4;          // Decrement the value of a by 4 (from 19 to 15)
```

is shorthand for

```
a = a - 4;
```

The –= is called the *subtraction assignment operator*, which decrements a variable by a given amount.

A subtraction assignment expression returns the target variable. The statement (line 21)

```
b = (a -= 4);   // Decrement the value of a by 4 (from 15 to 11),
                // then set b to the value of a (11)
```

for example, is equivalent to

```
a -= 4;
b = a;
```

The statement (line 24)

```
a *= 3;         // Multiply the value of a by 3 (from 11 to 33)
```

is shorthand for

```
a = a * 3;
```

The *****= is called the *multiplication assignment operator*, which multiplies a variable by a given value.

A multiplication assignment expression returns the target variable. The statement (line 27)

```
b = (a *= 3);   // Multiply the value of a by 3 (from 33 to 99),
                // then set b to the value of a (99)
```

for example, is equivalent to

```
a *= 3;
b = a;
```

The statement (line 30)

```
a /= 3;         // Divide the value of a by 3 (from 99 to 33)
```

is equivalent to

```
a = a / 3;
```

The /= is called the *division assignment operator*, which divides a variable by a given value.

A division assignment expression returns the target variable. The statement (line 33)

```
b = (a /= 3);   // Divide the value of a by 3 (from 33 to 11),
                // then set b to the value of a (11)
```

for example, is equivalent to

```
a /= 3;
b = a;
```

The statement (line 36)

```
a %= 4;          // Calculate (a % 4) (the remainder of a / 4, which is 3),
                 // then set the value of a to the result (3)
```

is shorthand for

```
a = a % 4;
```

The *%=* is called the *modulo assignment operator*, which modulus divides a variable by a given amount.

A modulo assignment expression returns the target variable. The statement (line 39)

```
b = (a %= 2);    // Calculate (a % 2) (the remainder of a / 2, which is 1),
                 // set the value of a to the result (1),
                 // then set the value of b to the value of a (1)
```

for example, is equivalent to

```
a %= 2;
b = a;
```

For the sake of simplicity, we'll avoid writing statements such as

```
b = (a += 8);
```

in favor of their long-form counterparts,

```
a += 8;
b = a;
```

Our program (*main.cpp*) generates the output

```
a = 3, b = 5
a = 11                // Result of a += 8;
a = 19, b = 19        // Result of b = (a += 8);
a = 15                // Result of a -= 4;
a = 11, b = 11        // Result of b = (a -= 4);
a = 33                // Result of a *= 3;
a = 99, b = 99        // Result of b = (a *= 3);
a = 33                // Result of a /= 3;
a = 11, b = 11        // Result of b = (a /= 3);
a = 3                 // Result of a %= 4;
a = 1, b = 1          // Result of b = (a %= 2);
```

Part 3: Control Flow

3.1: Relational Operators and Conditional Statements

Source files and folders

- *conditionalStatements*

Chapter outline

- *Comparing values using the relational operators*
- *Controlling program behavior using the if and else keywords*

In this section, we'll introduce some basic tools for managing *control flow*, the order in which statements are executed.

Our program (*main.cpp*) begins by prompting the user for an integer value, which we store as the variable *x* (lines 7-10). Lines 12-15,

```
if (x == 0)
{
  cout << "You entered 0\n";
}
```

can be read as "If the value of *x* is 0, print *You entered 0*." This is an example of an *if* statement. An *if statement* is a body of statements to be run, if and only if the given condition is true. The *condition* is an expression that returns a boolean (true-or-false) value. In this case, the condition is the expression

```
x == 0
```

which returns *true* if the value of *x* is 0, or *false* if the value of *x* isn't 0. The pair of equal signs (==) is the *equal to operator*, which checks whether the left operand (*x*) is equal to the right operand (0). The body is (line 14)

```
cout << "You entered 0\n";
```

If the body consists of a single statement, as it does here, we can omit the opening and closing braces (lines 13, 15). We could therefore write lines 12-15 as

```
if (x == 0)
  cout << "You entered 0\n";
```

The syntax of an *if* statement is thus

// Continued on next page

```
if (condition)
{
  body
}
```

Lines 17-20,

```
if (x % 2 == 0)
  cout << "You entered an even number\n";
else
  cout << "You entered an odd number\n";
```

can be read as "If the remainder of (*x / 2*) is 0, print *You entered an even number*; otherwise, print *You entered an odd number*." This is an example of an *if-else statement*, which uses the syntax

```
if (condition)
{
  body
}
else
{
  alternative body
}
```

If the condition is *true*, the body will be run; if the condition is *false*, the alternative body will be run. In this case, the condition is the expression

```
x % 2 == 0
```

where (*x % 2*) returns the remainder of (*x / 2*). Because *x* is an integer, (*x % 2*) will either be 0 or 1.

- If (*x % 2*) is 0, then (*x % 2 == 0*) will be *true*, and the body (line 18) will be run.
- If (*x % 2*) is 1, then (*x % 2 == 0*) will be *false*, and the alternative body (line 20) will be run.

Lines 22-27,

```
if (x > 0)
  cout << "You entered a positive number\n";
else if (x < 0)
  cout << "You entered a negative number\n";
else
  cout << "The number you entered is neither positive nor negative\n";
```

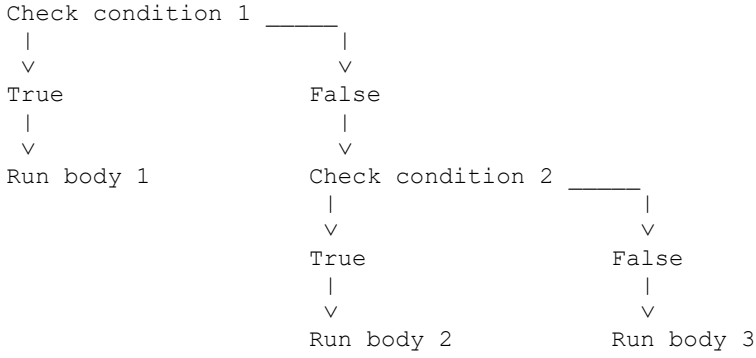
can be read as "If *x* is greater than 0, print *You entered a positive number.* Otherwise, if *x* is less than 0, print *You entered a negative number.* Otherwise, print *The number you entered is neither positive nor negative*."

The > symbol is the *greater than operator*, which checks whether the left operand (*x*) is greater than the right operand (0). The < symbol is the *less than operator*, which checks whether the left operand is less than the right.

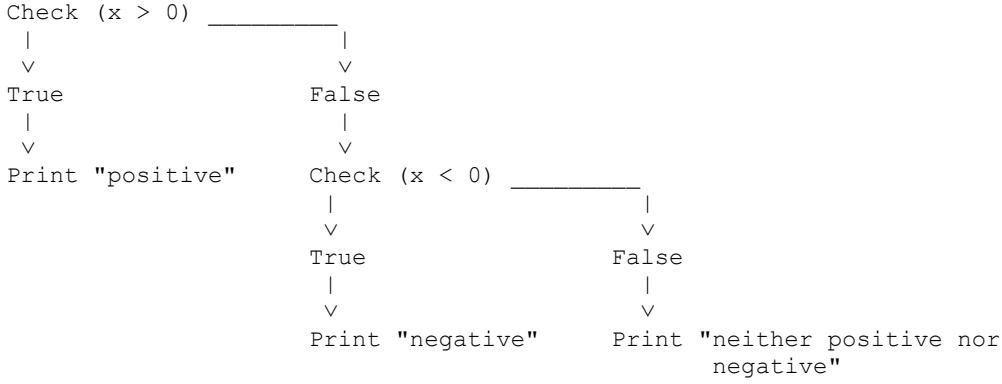
Lines 22-27 are an example of an *if-else if statement*, which uses the syntax

```
if (condition 1)
{
  body 1
}
else if (condition 2)
{
  body 2
}
else
{
  body 3
}
```

We can depict the above syntax using a flow chart:



The logic of lines 22-27 can thus be depicted as

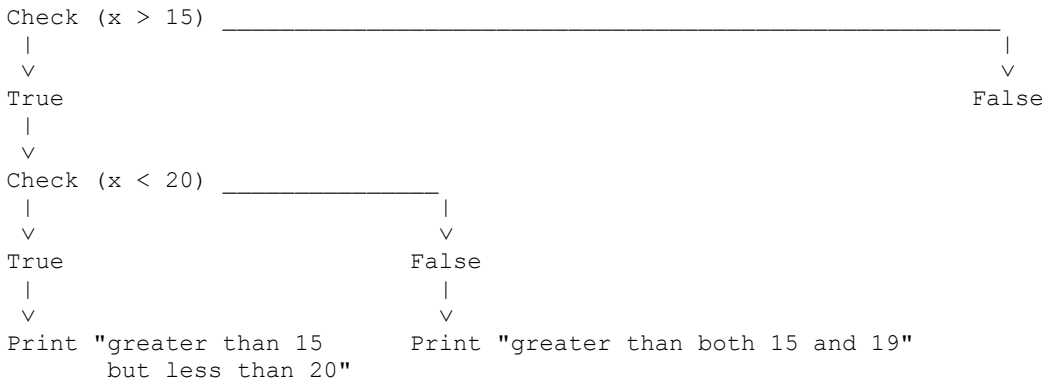


In lines 29-35,

// Continued on next page

```
if (x > 15)
{
  if (x < 20)
    cout << "You entered a number greater than 15 but less than 20\n";
  else
    cout << "You entered a number greater than both 15 and 19\n";
}
```

we check whether *x* is greater than 15. If it is, we then check whether *x* is also less than 20. Once again, we can depict the logic using a flowchart:



These types of statements (*if*, *if-else*, *if-else if*) are collectively known as *conditional statements*, or *conditionals*. A *nested conditional statement*, or *nested conditional*, is a conditional that exists within the body of another conditional. The *if-else* statement in lines 31-34, for example, is a nested conditional, as it exists within the body of another conditional (the *if* statement beginning in line 29).

Given the values 0, 17, -8, and 25, our program generates the output

```
Enter a number (integer): 0
You entered 0
You entered an even number
The number you entered is neither positive nor negative
```

```
Enter a number (integer): 17
You entered an odd number
You entered a positive number
You entered a number greater than 15 but less than 20
```

```
Enter a number (integer): -8
You entered an even number
You entered a negative number
```

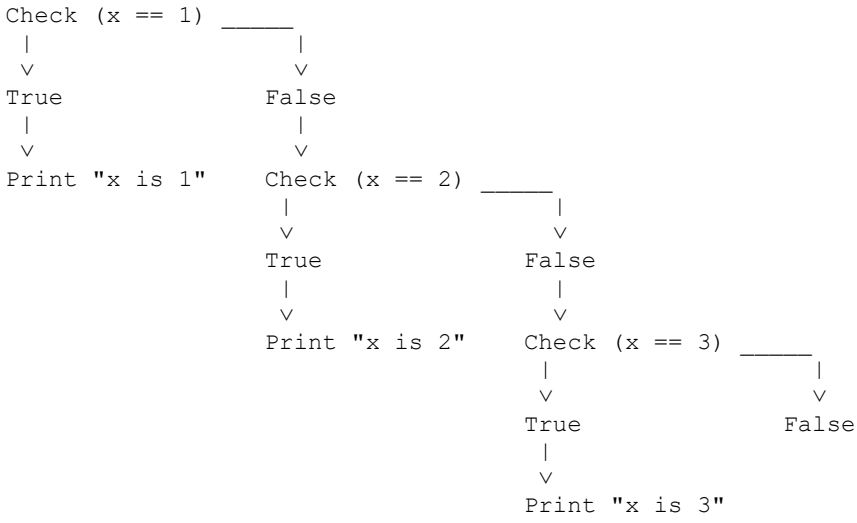
```
Enter a number (integer): 25
You entered an odd number
```

```
You entered a positive number
You entered a number greater than both 15 and 19
```

An *if-else if* statement can contain any number of *else if* clauses, and need not contain a final *else* clause. We could, for example, write

```
if (x == 1)
  cout << "x is 1\n";
else if (x == 2)
  cout << "x is 2\n";
else if (x == 3)
  cout << "x is 3\n";
```

In this case, if *x* isn't 1, 2, or 3, then no message will be printed:



Finally, the *comparison operators* (>, <, etc.) are also known as the *relational operators*, summarized in the following table:

Symbol	Name	Given two operands, left (L) and right (R), returns true if
==	Equal to	L is equal to R
!=	Not equal to	L is not equal to R
<	Less than	L is less than R
>	Greater than	L is greater than R
<=	Less than or equal to	L is less than or equal to R
>=	Greater than or equal to	L is greater than or equal to R

3.2: Logical Operators

Source files and folders

- *logicalOperators*

Chapter outline

- *Using the logical operators to form compound boolean expressions*

A *boolean expression* is an expression that returns a boolean (*true* / *false*) value, such as

```
x > 0     // Return true if x is greater than 0 (otherwise, return false)
x < 10    // Return true if x is less than 10 (otherwise, return false)
```

A *compound boolean expression* is a multi-part boolean expression. To form a compound boolean expression, we combine two or more boolean expressions, using a logical operator. The three *logical operators* are *and* (&&), *or* (||), and *not* (!).

To demonstrate this, our program (*main.cpp*) begins by prompting the user for an integer value, and storing it as the variable *x* (lines 7-10). We then check whether *x* is greater than 0 and less than 10 (line 12), via the expression

```
x > 0 && x < 10
```

The pair of ampersands (&&) is the *and operator*, which returns *true* if and only if the left and right operands are both *true*. In this case, the left operand is (*x > 0*), and the right operand is (*x < 10*).

- If (*x > 0*) and (*x < 10*) are both *true*, then (*x > 0 && x < 10*) returns *true*
- If either (*x > 0*) is *false*, or (*x < 10*) is *false*, then (*x > 0 && x < 10*) returns *false*

If the expression (*x > 0 && x < 10*) returns *true* (line 12), then (*x > 0*) and (*x < 10*) are both *true*. We therefore print the message *Your number is positive and less than 10* (line 13).

If the expression (*x > 0 && x < 10*) returns *false* (line 14), then either (*x > 0*) is *false*, or (*x < 10*) is *false*. We therefore print the message *Your number is either nonpositive or greater than 9* (line 15).

We then check whether *x* is either odd, negative, or both, via the expression (line 17)

```
x % 2 == 1 || x < 0
```

The pair of vertical lines (||) is the *or operator*, which returns *true* if either the left operand is *true*, or right operand is *true*. In this case, the left operand is (*x % 2 == 1*), which returns *true* if *x* is odd (by checking whether the remainder of (*x / 2*) is 1). The right operand is (*x < 0*).

- If either (*x % 2 == 1*) is *true*, or (*x < 0*) is *true*, then (*x % 2 == 1 || x < 0*) returns *true*
- If (*x % 2 == 1*) and (*x < 0*) are both *false*, then (*x % 2 == 1 || x < 0*) returns *false*

If the expression (*x % 2 == 1 || x < 0*) returns *true* (line 17), we print the message *Your number is either odd, negative, or both* (odd and negative) (line 18).

If the expression (*x % 2 == 1 || x < 0*) returns *false* (line 19), then (*x % 2 == 1*) and (*x < 0*) are both *false*. We therefore print the message *Your number is neither odd nor negative* (line 20).

We can use parentheses to write more complex expressions, such as (line 22)

```
(x < -5 && x > -10) || (x > 5 && x < 10)
```

In this case, the left operand is

```
(x < -5 && x > -10)
```

which returns *true* if *x* is between -5 and -10. The right operand is

```
(x > 5 && x < 10)
```

which returns *true* if *x* is between 5 and 10. The expression

```
(x < -5 && x > -10) || (x > 5 && x < 10)
```

thus returns *true* if *x* is between -5 and -10, *or* if *x* is between 5 and 10.

If it returns *true* (line 22), we print the message *Your number is either between -5 and -10, or between 5 and 10* (line 23).

If it returns *false* (line 24), then the left and right operands are both *false*. We therefore print the message *Your number is neither between -5 and -10, nor between 5 and 10* (line 25).

Finally, we determine whether *x* is *not* less than 0, via the expression (line 27)

```
!(x < 0)
```

The exclamation point (!) is the *not operator*, which returns the inverse (opposite) boolean value of its operand. In this case, the operand is (*x < 0*).

- If (*x < 0*) is *true*, then !(*x < 0*) returns *false*
- If (*x < 0*) is *false*, then !(*x < 0*) returns *true*

If !(*x < 0*) returns *true* (line 27), then *x* is not less than 0, so we print *Your number is not negative* (line 28).

If !(*x < 0*) returns *false* (line 29), then *x* is less than 0, so we print *Your number is negative* (line 30).

If, for example, *x* is 2, then *x* is not less than 0. The expression !(*x* < *0*) (*x* is *not* less than 0) will therefore return *true*:

```
!(x < 0) = !(2 < 0)
         = !(false)
         = true       // If x is 2, then "x is not less than 0" is true
```

If, on the other hand, *x* is -2, then *x* is less than 0. The expression !(*x* < *0*) (*x* is *not* less than 0) will therefore return *false*:

```
!(x < 0) = !(-2 < 0)
         = !(true)
         = false      // If x is -2, then "x is not less than 0" is false
```

Given the values 3, -6, -8, 18, 0, and 9, our program generates the following output:

```
Enter a number (integer): 3
Your number is positive and less than 10
Your number is either odd, negative, or both
Your number is neither between -5 and -10, nor between 5 and 10
Your number is not negative

Enter a number (integer): -6
Your number is either nonpositive or greater than 9
Your number is either odd, negative, or both
Your number is either between -5 and -10, or between 5 and 10
Your number is negative

Enter a number (integer): -8
Your number is either nonpositive or greater than 9
Your number is either odd, negative, or both
Your number is either between -5 and -10, or between 5 and 10
Your number is negative

Enter a number (integer): 18
Your number is either nonpositive or greater than 9
Your number is neither odd nor negative
Your number is neither between -5 and -10, nor between 5 and 10
Your number is not negative

Enter a number (integer): 0
Your number is either nonpositive or greater than 9
Your number is neither odd nor negative
Your number is neither between -5 and -10, nor between 5 and 10
Your number is not negative

Enter a number (integer): 9
Your number is positive and less than 10
Your number is either odd, negative, or both
Your number is either between -5 and -10, or between 5 and 10
Your number is not negative
```

3.3: Loops

Source files and folders

- *loops*

Chapter outline

- *Performing repetition, using the for and while keywords*

A *loop* is a set of statements to be repeatedly run, as long as a given condition returns *true*. The two most common types of loops are *for* loops and *while* loops. Our program (*main.cpp*) begins with the *for* loop (lines 7-8)

```
for (int i = 0; i != 5; ++i)
  cout << i << endl;
```

which runs the statement

```
cout << i << endl;
```

for *i* = {*0, 1, 2, 3, 4*}. A *for loop* uses the following syntax:

```
for (initializer; condition; post body)
{
  body;
}
```

The *initializer* is a single statement, run once (and only once). In this case, the initializer is (*int i = 0;*).

The *condition* is a boolean expression, checked before each execution of the body. In this case, the condition is (*i != 5*).

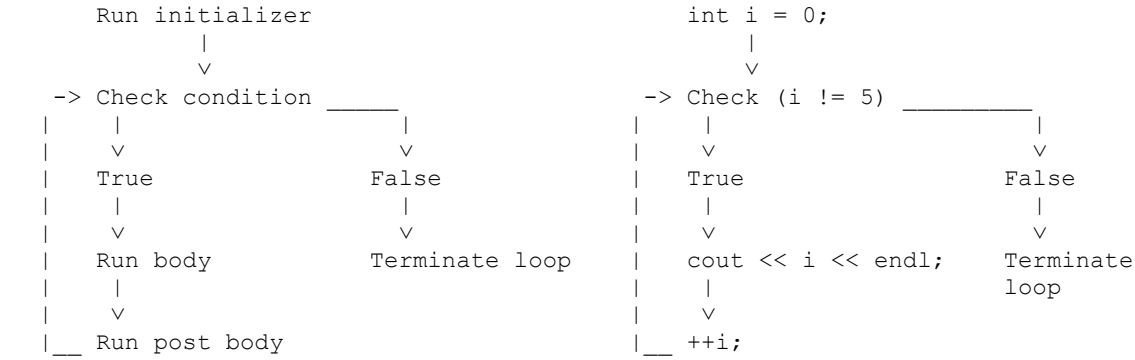
- If the condition returns *true*, the body is run once, followed by the post body.
- If the condition returns *false*, the loop terminates.

The *body* is a set of statements to be run, each time the condition returns *true*. If the body consists of a single statement, we can omit the braces ({}). In this case, the body is (*cout << i << endl;*).

The *post body* is a single statement to be run, immediately following each execution of the body. In this case, the post body is (*++i*).

We can depict these operations using a flow chart:

// Continued on next page



One *iteration* of a loop is a single execution of the body. The loop in lines 7-8 performs a total of 5 iterations:

```
int i = 0;                // Initializer

i != 5 is true            // 0 != 5 is true
  Iteration 1:
    cout << i << endl;    // Print 0;
    ++i;                  // i = 1;

i != 5 is true            // 1 != 5 is true
  Iteration 2:
    cout << i << endl;    // Print 1;
    ++i;                  // i = 2;

i != 5 is true            // 2 != 5 is true
  Iteration 3:
    cout << i << endl;    // Print 2;
    ++i;                  // i = 3;

i != 5 is true            // 3 != 5 is true
  Iteration 4:
    cout << i << endl;    // Print 3;
    ++i;                  // i = 4;

i != 5 is true            // 4 != 5 is true
  Iteration 5:
    cout << i << endl;    // Print 4;
    ++i;                  // i = 5;

i != 5 is false           // 5 != 5 is false
  Terminate loop;
```

All variables created within a loop are automatically destroyed when the loop terminates. In line 7, for example, we create *i* in the initializer. Once the loop ends, this particular *i* no longer exists.

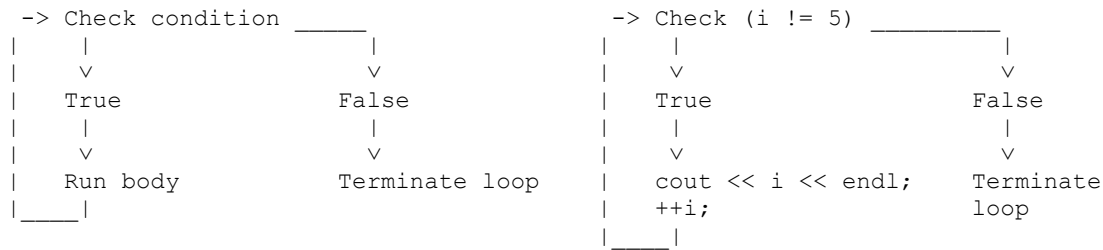
In line 12, we create a new integer *i* and initialize it to 0. We then demonstrate an example of a *while loop* (lines 14-18), which uses the following syntax:

```
while (condition)
{
  body
}
```

The *condition* and *body* have the same definitions as those of a *for* loop, described earlier. In this case, the condition is (*i != 5*), and the body is

```
cout << i << endl;
++i;
```

Once again, we can depict these operations using a flow chart:



Because we initialized *i* to 0 (line 12), this loop (lines 14-18) performs a total of 5 iterations, just like the previous one (lines 7-8).

Note, however, that this new *i* (line 12) wasn't created within a loop. It will therefore remain in existence for the duration of *main*.

A *nested loop* is a loop that exists within the body of another loop. The loop in lines 29-30, for example, is a nested loop. The surrounding loop (lines 22-31) is called the *outer loop* (or *main loop*), and the nested loop (lines 29-30) is called the *inner loop*.

The outer loop performs a total of 5 iterations, for *i* = {*0, 1, 2, 3, 4*}. In each iteration of the outer loop,

- We check whether *i* is even or odd, and print the result (lines 24-27).

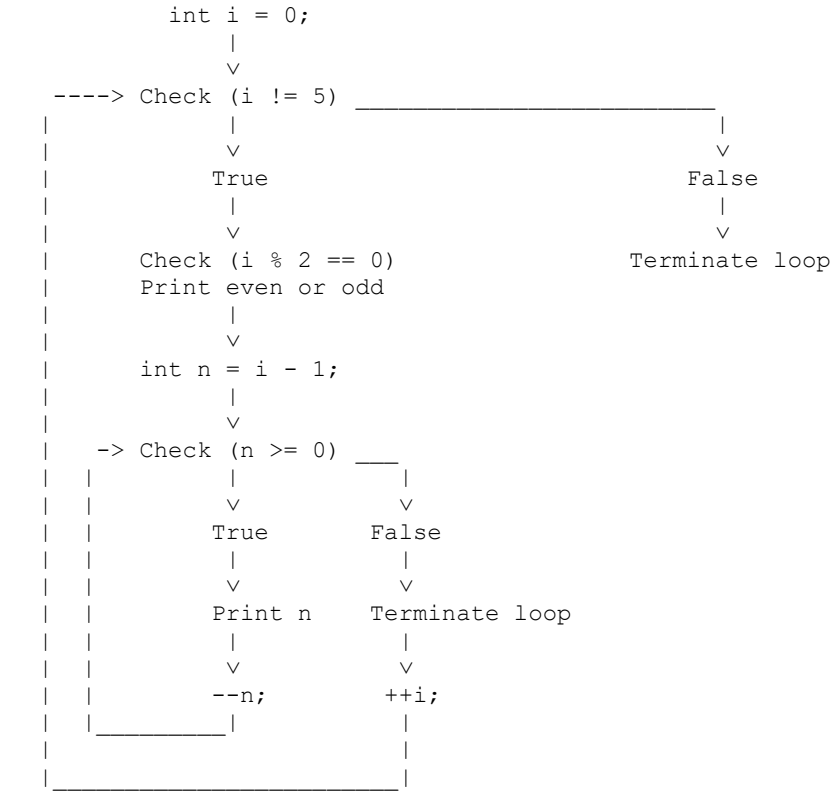
- We then run the inner loop. The inner loop performs a total of *i* iterations, for *n* = (*i – 1*) down to 0 (line 29). In each iteration of the inner loop, we print two whitespaces, followed by *n* (line 30).

  - When *i* is 0, the inner loop performs 0 iterations.
  - When *i* is 1, the inner loop performs 1 iteration, for *n* = {*0*}.
  - When *i* is 2, the inner loop performs 2 iterations, for *n* = {*1, 0*}.
  - When *i* is 3, the inner loop performs 3 iterations, for *n* = {*2, 1, 0*}.
  - When *i* is 4, the inner loop performs 4 iterations, for *n* = {*3, 2, 1, 0*}.

The entire process (lines 22-31) thus generates the output

```
0 is even    // Outer loop iteration 1 (i = 0)
1 is odd     // Outer loop iteration 2 (i = 1)
  0          //   Inner loop iteration 1 (n = 0)
2 is even    // Outer loop iteration 3 (i = 2)
  1          //   Inner loop iteration 1 (n = 1)
  0          //   Inner loop iteration 2 (n = 0)
3 is odd     // Outer loop iteration 4 (i = 3)
  2          //   Inner loop iteration 1 (n = 2)
  1          //   Inner loop iteration 2 (n = 1)
  0          //   Inner loop iteration 3 (n = 0)
4 is even    // Outer loop iteration 5 (i = 4)
  3          //   Inner loop iteration 1 (n = 3)
  2          //   Inner loop iteration 2 (n = 2)
  1          //   Inner loop iteration 3 (n = 1)
  0          //   Inner loop iteration 4 (n = 0)
```

Below is a flow chart that illustrates the logic, followed by a walk-through of the entire procedure:



```
int i = 0;
```

```
// Continued on next page
```

```
i != 5 is true
  Iteration 1
  {
    i % 2 == 0 is true
      Print "<i> is even";

    int n = i - 1;              // n = -1;

    n >= 0 is false
      Terminate inner loop;

    ++i;                        // i = 1;
  }
```

```
i != 5 is true
  Iteration 2
  {
    i % 2 == 0 is false
      Print "<i> is odd";

    int n = i - 1;              // n = 0;

    n >= 0 is true
      Iteration 1
        Print n;
        --n;                    // n = -1;

    n >= 0 is false
      Terminate inner loop;

    ++i;                        // i = 2;
  }
```

```
i != 5 is true
  Iteration 3
  {
    i % 2 == 0 is true
      Print "<i> is even";

    int n = i - 1;              // n = 1;

    n >= 0 is true
      Iteration 1
        Print n;
        --n;                    // n = 0;

    n >= 0 is true
      Iteration 2
        Print n;
        --n;                    // n = -1;

    // Continued on next page
```

```
    n >= 0 is false
      Terminate inner loop;

    ++i;                        // i = 3;
  }
```

```
i != 5 is true
  Iteration 4
  {
    i % 2 == 0 is false
      Print "<i> is odd";

    int n = i - 1;              // n = 2;

    n >= 0 is true
      Iteration 1
        Print n;
        --n;                    // n = 1;

    n >= 0 is true
      Iteration 2
        Print n;
        --n;                    // n = 0;

    n >= 0 is true
      Iteration 3
        Print n;
        --n;                    // n = -1;

    n >= 0 is false
      Terminate inner loop;

    ++i;                        // i = 4;
  }
```

```
i != 5 is true
  Iteration 5
  {
    i % 2 == 0 is true
      Print "<i> is even";

    int n = i - 1;              // n = 3;

    n >= 0 is true
      Iteration 1
        Print n;
        --n;                    // n = 2;

    n >= 0 is true
      Iteration 2
        Print n;
        --n;                    // n = 1;
```

```
    n >= 0 is true
      Iteration 3
        Print n;
        --n;                    // n = 0;

    n >= 0 is true
      Iteration 4
        Print n;
        --n;                    // n = -1;

    n >= 0 is false
      Terminate inner loop;

    ++i;                        // i = 5;
  }
```

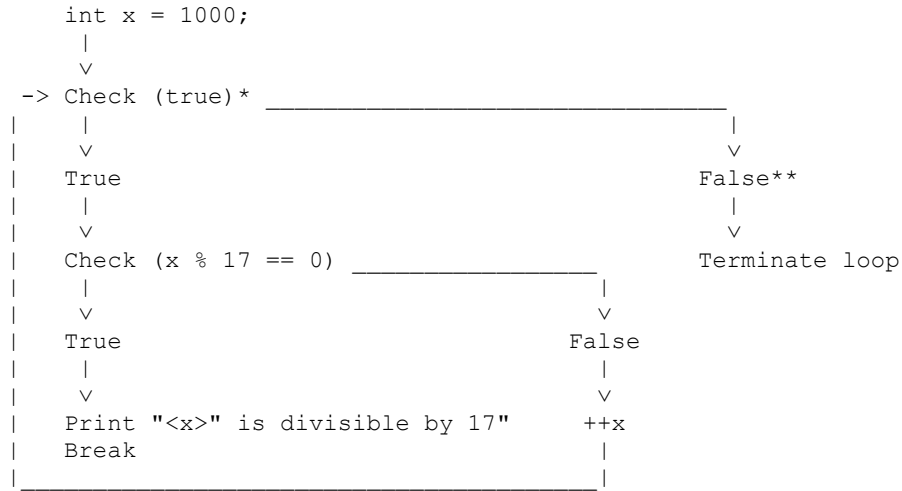
```
i != 5 is false
  Terminate outer loop;
```

Our final example is a loop in which the condition is the boolean value *true* (line 35). In this case, the termination condition resides within the body (lines 37-41). In each iteration, we check whether the current value of *x* is divisible by 17 (line 37).

- If so, we print the message *<x> is divisible by 17* (line 39). We then terminate the loop, via the *break* keyword (line 40).

- If not, the post body increments *x* and we begin the next iteration.

By initializing *x* to 1000, this loop (lines 35-42) finds the first integer greater than or equal to 1000 that is divisible by 17. The following flow chart illustrates the logic:



The loop condition (*true*) (marked with a * in the above diagram) will always return *true*. The first *false* branch (marked with a **) will therefore never be taken. The loop performs a total of 4 iterations, for *x* = {*1000, 1001, 1002, 1003*}:

```
  int x = 1000;
_____________________________________________________________________________

  true is true
    Iteration 1
      x % 17 == 0 is false;
      ++x;                      // x = 1001
_____________________________________________________________________________

  true is true
    Iteration 2
      x % 17 == 0 is false;
      ++x;                      // x = 1002
_____________________________________________________________________________

  true is true
    Iteration 3
      x % 17 == 0 is false;
      ++x;                      // x = 1003
_____________________________________________________________________________

  true is true
    Iteration 4
      x % 17 == 0 is true
        Print "<x> is divisible by 17";
        break;
```

The *while* loop equivalent of lines 35-42 is

```
int x = 1000;

while (true)
{
  if (x % 17 == 0)
  {
    cout << x << " is divisible by 17\n"
    break;
  }

  ++x;
}
```

The advantage of the *while* version is that *x* is created outside the loop, so we can continue using *x* after the loop ends. In the *for* version (lines 35-42), *x* is created within the loop, so *x* will be destroyed upon termination of the loop.

When a *break* statement occurs in a nested loop, execution resumes at the next line of the surrounding loop. To demonstrate this, we'll use the following nested loop:

```
for (int k = 0; k != 3; ++k)    // Surrounding loop (Outer loop)
{
  cout << "A\n";

  for (int y = 5; true; ++y)    // Nested loop (Inner loop)
  {
    if (y == 7)
    {
      cout << y << endl;
      break;
    }
  }

  cout << "B\n";
}
```

The outer loop performs 3 iterations, for *k* = {*0, 1, 2*}. In each iteration of the outer loop, we print the letter *A* (*cout << "A\n";*), then execute the inner loop.

We begin the inner loop by initializing *y* to 5. In each iteration of the inner loop, we check if *y* is 7:

- If *y* is 7, we print the value of *y* (7), then *break* out of (terminate) the inner loop. This brings us to the next line of the outer loop, in which we print the letter *B* (*cout << "B\n";*)

- If *y* isn't 7, we increment *y* and prepare for the next iteration of the inner loop.

Because *y* is initialized to 5 every single time, the inner loop will always perform 3 iterations, for *y* = {*5*, *6*, *7*}.

The entire process thus generates the output

```
A    // Outer loop iteration 1 (k = 0)
7    //   Nested loop iteration 3 (y = 7)
B
A    // Outer loop iteration 2 (k = 1)
7    //   Nested loop iteration 3 (y = 7)
B
A    // Outer loop iteration 3 (k = 2)
7    //   Nested loop iteration 3 (y = 7)
B
```

The key point here is that when a *break* statement occurs in a nested loop, it only terminates the nested loop, *not* the surrounding loop.

The full output of our program (*main.cpp*) is

```
0                             // for loop
1
2
3
```

```
4

0                           // while loop
1
2
3
4

0 is even                   // Outer loop iteration 1 (i = 0)
1 is odd                    // Outer loop iteration 2 (i = 1)
  0                         //   Inner loop iteration 1 (n = 0)
2 is even                   // Outer loop iteration 3 (i = 2)
  1                         //   Inner loop iteration 1 (n = 1)
  0                         //   Inner loop iteration 2 (n = 0)
3 is odd                    // Outer loop iteration 4 (i = 3)
  2                         //   Inner loop iteration 1 (n = 2)
  1                         //   Inner loop iteration 2 (n = 1)
  0                         //   Inner loop iteration 3 (n = 0)
4 is even                   // Outer loop iteration 5 (i = 4)
  3                         //   Inner loop iteration 1 (n = 3)
  2                         //   Inner loop iteration 2 (n = 2)
  1                         //   Inner loop iteration 3 (n = 1)
  0                         //   Inner loop iteration 4 (n = 0)

1003 is divisible by 17     // Inner loop iteration 4 (x = 1003)
```

3.4: Boolean Variables

Source files and folders

- *booleanVariables*

Chapter outline

- *Using boolean variables to increase code readability*

As discussed in Chapter 1.3, a variable of type *bool* stores a boolean (*true* / *false*) value. This allows us to store the result of a boolean expression, such as (*x > 0 && x < 10*), in a *bool* variable.

To demonstrate this, our program (*main.cpp*) begins by prompting the user for an integer value, which we store as the variable *k* (lines 7-10). We then create a *bool* variable, *isPositive* (line 12), initializing its value to the result of the expression (*k > 0*).

- If (*k > 0*) returns *true*, then *isPositive* will be *true*.
- If (*k > 0*) returns *false*, then *isPositive* will be *false*.

Similarly, we initialize *isNegative* (line 13) to the result of the expression (*k < 0*):

- If (*k < 0*) returns *true*, then *isNegative* will be *true*.
- If (*k < 0*) returns *false*, then *isNegative* will be *false*.

Lastly, we initialize *isEven* (line 14) to the result of the expression (*k % 2 == 0*):

- If (*k % 2 == 0*) returns *true*, then *isEven* will be *true*.
- If (*k % 2 == 0*) returns *false*, then *isEven* will be *false*.

We can now use these variables (*isPositive*, *isNegative*, *isEven*) to form boolean expressions, improving the readability of our conditional statements:

- If (*isEven*) is *true* (line 16), we print *You entered an even number* (line 17).

- If (*!isEven*) ("is not even") is *true* (line 19), we print *You entered an odd number* (line 20).

- If (*!isPositive*) ("is not positive") and (*!isNegative*) ("is not negative") are both *true* (line 22), we print *You entered 0* (line 23).

Given the values -3, 0, and 5, our program generates the following output:

```
Enter a number (integer): -3
You entered an odd number
```

```
Enter a number (integer): 0
You entered an even number
You entered 0
```

```
Enter a number (integer): 5
You entered an odd number
```

3.5: Putting It All Together

Source files and folders

- *retirementAge*

Chapter outline

- *Writing a small program that combines the key concepts from Parts 1-3*

Our program in this chapter combines all of the key concepts introduced thus far. It begins by prompting the user for their current age, as well as their planned age of retirement. It then displays the user's future age and retirement status for the next 25 years, in 5-year intervals. Given the values 43 and 60, for example, the program generates the following output:

```
What is your current age? 43

At what age do you plan to retire? 60

In 5 years, you will be 48 years old.
  You will have 12 years before retirement.

In 10 years, you will be 53 years old.
  You will have 7 years before retirement.

In 15 years, you will be 58 years old.
  You will have 2 years before retirement.

In 20 years, you will be 63 years old.
  You will have been retired for 3 years.

In 25 years, you will be 68 years old.
  You will have been retired for 8 years.
```

We begin by obtaining the user's *currentAge* and *retirementAge* (*main.cpp*, lines 7-14). The loop (line 16) then performs 5 iterations, for *futureYears* = {*5, 10, 15, 20, 25*}. The variable *futureYears* represents the number of years in the future, relative to the current year. In each iteration,

- We calculate the user's *futureAge*, by adding the number of *futureYears* to their *currentAge* (line 18).

- We determine whether or not the user will be retired at that point, by checking whether their *futureAge* is greater than or equal to their *retirementAge* (line 19).

  - If (*futureAge* >= *retirementAge*) is *true*, then *isRetired* will be *true*.
  - If (*futureAge* >= *retirementAge*) is *false*, then *isRetired* will be *false*.

- We print the message *In <futureYears> years, you will be <futureAge> years old* (lines 21-22).

- If *isRetired* is *true* (line 24), then the number of years spent in retirement is equal to (*futureAge – retirementAge*). We thus print the message *You will have been retired for <futureAge – retirementAge> years* (lines 26-27).

- If *isRetired* is *false* (line 29), then the number of years remaining until retirement is equal to (*retirementAge – futureAge*). We thus print the message *You will have <retirementAge – futureAge> years before retirement* (lines 31-32).

Given a *currentAge* of 43 and a *retirementAge* of 60, for example, the loop performs the following operations:

```
int futureYears = 5;
```

```
futureYears <= 25 is true
  Iteration 1
  {
    futureAge = currentAge + futureYears
              = 43 + 5
              = 48;

    isRetired = (futureAge >= retirementAge)
              = (48 >= 60)
              = false;

    Print "In <5> years, you will be <48> years old."

    isRetired is false
      Print "You will have <60 - 48> years before retirement."

    futureYears += 5
                 = 10;
  }
```

```
futureYears <= 25 is true
  Iteration 2
  {
    futureAge = currentAge + futureYears
              = 43 + 10
              = 53;

    isRetired = (futureAge >= retirementAge)
              = (53 >= 60)
              = false;

    Print "In <10> years, you will be <53> years old."
```

```
    isRetired is false
      Print "You will have <60 - 53> years before retirement."

    futureYears += 5
                 = 15;
  }
```

```
futureYears <= 25 is true
  Iteration 3
  {
    futureAge = currentAge + futureYears
              = 43 + 15
              = 58;

    isRetired = (futureAge >= retirementAge)
              = (58 >= 60)
              = false;

    Print "In <15> years, you will be <58> years old."

    isRetired is false
      Print "You will have <60 - 58> years before retirement."

    futureYears += 5
                 = 20;
  }
```

```
futureYears <= 25 is true
  Iteration 4
  {
    futureAge = currentAge + futureYears
              = 43 + 20
              = 63;

    isRetired = (futureAge >= retirementAge)
              = (63 >= 60)
              = true;

    Print "In <20> years, you will be <63> years old."

    isRetired is true
      Print "You will have been retired for <63 - 60> years."

    futureYears += 5
                 = 25;
  }
```

// Continued on next page

```
futureYears <= 25 is true
  Iteration 5
  {
    futureAge = currentAge + futureYears
              = 43 + 25
              = 68;

    isRetired = (futureAge >= retirementAge)
              = (68 >= 60)
              = true;

    Print "In <25> years, you will be <68> years old."

    isRetired is true
      Print "You will have been retired for <68 - 60> years."

    futureYears += 5
                 = 30;
  }
```

```
futureYears <= 25 is false
  Terminate loop;
```

Part 4: Functions and Scope

4.1: Functions

Source files and folders

- *functions*

Chapter outline

- *Declaring, defining, and calling functions*
- *The difference between arguments and parameters*

As discussed in Chapter 1.3, a *function*, also called a *subroutine* or *method*, is a set of instructions that can be *called* (executed) by name. A *function call* is the act of *running* (executing) a function.

Functions allow us to divide large, complex tasks into smaller, simpler tasks. They also let us reuse code, thereby reducing unnecessary code duplication. This, in turn, makes our code more manageable and easier to follow.

To create a function, we begin by declaring it. A *function declaration* is a statement that provides the information required to call a function: its name, parameters, and return type. The syntax of a function declaration is

```
return_type function_name(parameter_list);
```

where

- *return_type* is the the function's return type. As discussed in Chapter 1.3, the *return type* is the type of value (*int*, *double*, etc.) returned to the *caller* (executor) of the function, upon completion. If there is no return value, we use a return type of *void*.

- *function_name* is the name of the function.

- *parameter_list* is a comma-separated list of *parameters* (input values). Each parameter must have a type (*int*, *double*, etc.) and name, separated by a space. If there are no parameters, we leave the list empty.

Our program (*main.cpp*), for example, begins by declaring a function called *salesTax* (line 3), which returns the amount of sales tax (in dollars) on an item. The two parameters are

- *price* (of type *double*): the price of the item, in dollars
- *rate* (of type *double*): the sales tax rate, as a percentage of the item price (e.g. 3.5, for 3.5%)

The return type is *double*. Recall from Chapter 1.3 that the type *double* represents a real number (e.g. 3.14159).

We then declare another function, *shippingCost* (line 4), which returns the shipping cost (in dollars) of an item. The parameter *weight* (of type *double*) is the weight of the item, in pounds. The return type (like that of *salesTax*) is also *double*.

Once we declare a function, we need to define it. A *function definition*, also called an *implementation*, specifies the function body. The *function body* is the set of instructions to be run when the function is called. The syntax of a function definition is

```
return_type function_name(parameter_list)
{
  function_body
}
```

The *return_type*, *function_name*, and *parameter_list* must match those of the corresponding function declaration. Note, however, that a function definition doesn't contain a semicolon (;) at the end of the *parameter_list* (unlike a function declaration).

To *implement* (define) the *salesTax* function, we simply return (*price* * (*rate / 100*)) (lines 32-35).

To implement *shippingCost*, we begin by creating *cost* (line 39), a variable of type *double*, which we'll use to store the final shipping cost.

- If the item *weight* is under 5 pounds (line 41), we'll set the *cost* to 7.99 (line 42), which represents a flat rate of $7.99.

- If the *weight* is 5 pounds or more (line 43), we'll set the *cost* to (*2.00 * weight*) (line 44), which represents a cost of $2 per pound.

After calculating the *cost*, we return it to the caller (line 46).

Now that we've completed *salesTax* and *shippingCost*, we'll call them from within *main*. We begin by prompting the user for three values: *retailPrice* (the price of an item), *salesTaxRate*, and *itemWeight* (lines 10-21).

We then compute the *totalCost* of the item, by adding the *retailPrice*, *salesTax*, and *shippingCost* (lines 23-25). The expression (line 24)

```
salesTax(retailPrice, salesTaxRate)
```

calls the *salesTax* function. When calling a function, the values that we *pass* (send) to the function are called *arguments*. In line 24, *retailPrice* and *salesTaxRate* are the arguments being *passed* (sent) to the *salesTax* function. If, for example, *retailPrice* is 24.99 and *salesTaxRate* is 6.25, then *salesTax* will return 1.561875 (24.99 x 6.25/100).

Similarly, in the function call (line 25)

```
shippingCost(itemWeight)
```

itemWeight is the argument being passed to *shippingCost*. If, for example, *itemWeight* is 2.50, then *shippingCost* will return 7.99. Conversely, if *itemWeight* is 8.25, then *shippingCost* will return 16.50 (2.00 x 8.25).

After calculating the *totalCost*, we print the message (line 27)

```
The total cost is $<totalCost>
```

Below is the output of two sample runs of our program. The first demonstrates an example of a flat rate *shippingCost* (when the *itemWeight* is under 5 pounds). The second demonstrates an example of a per-pound *shippingCost* (when the *itemWeight* is 5 pounds or above).

```
Enter retail price (e.g. for $24.99, enter 24.99): 29.99
Enter sales tax rate (e.g. for 6.25%, enter 6.25): 7.50
Enter item weight (e.g. for 2.50 lbs, enter 2.50): 3.00

The total cost is $40.2293
```

```
Enter retail price (e.g. for $24.99, enter 24.99): 14.99
Enter sales tax rate (e.g. for 6.25%, enter 6.25): 4.75
Enter item weight (e.g. for 2.50 lbs, enter 2.50): 8.50

The total cost is $32.702
```

Before moving on, there are a few important points to note regarding functions.

In order to call a function, the declaration must appear before the *call site* (the location at which the function is called). In our program, for example, the declaration of *salesTax* (line 3) appears before the call site (line 24). If we try to call a function before declaring it, our program won't compile.

We can, however, declare the same function multiple times. This is often necessary in programs consisting of multiple source files. If, for example, our program consisted of two *.cpp* files, both of which contained calls to *salesTax*, we would need to declare *salesTax* in both files.

Every function must have one (and only one) definition. If we try to define the same function multiple times, our program won't compile.

A function definition, however, does *not* need to appear before the call(s). In our program, for example, the definition of *salesTax* (lines 32-35) appears after the call (line 24).

Function parameters are separate variables, distinct from arguments. Each time a function is called, its parameters are initialized to the values of the corresponding arguments. When we call *shippingCost* (line 25), for example, the parameter *weight* (line 37) is a separate variable, initialized to the value of

the argument *itemWeight* (line 25).

As a general guideline, the body of any one function shouldn't exceed 40 lines or so. This is mainly for the sake of code readability, not due to any technical constraints. If you find yourself drastically going over this limit, you're likely trying to perform too much work within a single function. In such cases, try dividing the task among shorter, more focused functions.

On a related note, maintaining a line limit of 80 characters can also improve readability. A *line limit* of 80 characters means that no single line of code exceeds 80 characters. If a single statement exceeds 80 characters (as in lines 23-25 of *main.cpp*), we divide it among multiple lines. Additionally, we use indentation to show that the separate lines are all part of the same statement.

At the beginning of the chapter, I mentioned that in a function declaration, we must specify the type and name of each parameter. Technically, however, it is possible to omit the parameter names in a function declaration. We could have, for example, written the declaration of *salesTax* (line 13) as

```
double salesTax(double, double);
```

instead of

```
double salesTax(double price, double rate);
```

Though technically legal, this would be considered bad form, as it omits essential information.

In any case, this technicality does *not* apply to function definitions. If, for example, we attempt to omit the parameter names from the definition of *salesTax* (lines 79-82), as in

```
double salesTax(double, double)
{
  return price * (rate / 100);
}
```

then our code won't compile, because the compiler won't know what *price* and *rate* are in the function body (line 81).

4.2: Namespaces

Source files and folders

- *namespaces*

Chapter outline

- *Declaring and using namespaces*

Our program in this chapter is functionally identical to that of the previous chapter. The code, however, is reorganized to demonstrate the use of namespaces.

A *namespace* is a named region in which we can group together conceptually related components. The syntax of a namespace declaration is

```
namespace n
{
members
};
```

where *n* is the name of the namespace, and *members* are the functions, variables, etc. that reside in namespace *n*. Note that a namespace declaration contains a semicolon (;) after the closing brace.

Our program, for example, begins by declaring three functions, *getRetailPrice*, *getSalesTaxRate*, and *getItemWeight* (*main.cpp*, lines 5-7), in a namespace called *userInterface* (lines 3-4, 8). These three functions are said to be "declared in namespace *userInterface*." The name, *userInterface*, indicates that its members are responsible for handling user interaction.

Because we declared these functions in *userInterface*, we must also define them in *userInterface* (lines 33-34, 70):

- *getRetailPrice* prompts the user for the retail price and returns it to the caller (lines 35-45).
- *getSalesTaxRate* prompts the user for the sales tax rate and returns it to the caller (lines 47-57).
- *getItemWeight* prompts the user for the item weight and returns it to the caller (lines 59-69).

Next, we'll create another namespace, called *feeCalculator* (lines 10-11, 15), whose members are responsible for computing the various fees:

- *salesTax* (line 13) returns the sales tax, using the given *price* and *rate*.
- *shippingCost* (line 14) returns the shipping cost, using the given *weight*.
- *totalFees* (line 12) returns (*salesTax* + *shippingCost*), using the given *itemPrice*, *salesTaxRate*, and *itemWeight*.

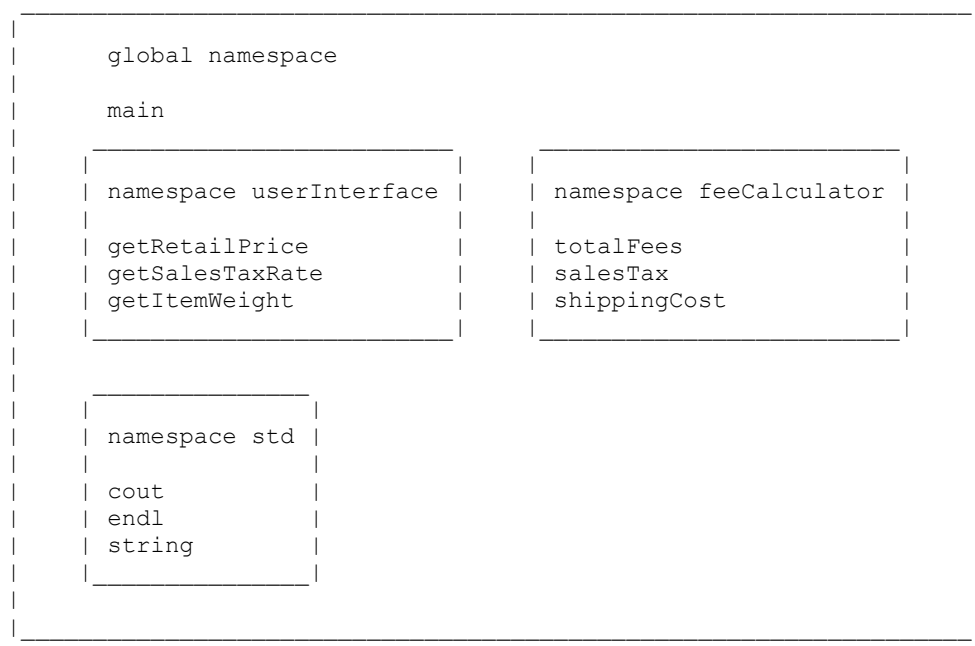
Because we declared these functions in *feeCalculator*, we must also define them in *feeCalculator* (lines

72-73, 95):

- *salesTax* and *shippingCost* (lines 79-82, 84-94) are unchanged from the previous chapter.
- *totalFees* (lines 74-77) simply calls *salesTax* and *shippingCost*, returning the sum.

Standard Library components, such as *cout*, *endl*, and *string*, are declared in namespace *std*. Although *std* is short for "standard," some people pronounce it as "stud" or "stood" (rather than "STD") for the sake of brevity.

The *main* function resides in the global namespace. The *global namespace* is the nameless, outermost namespace, which encloses all other namespaces. The following diagram illustrates the relationships between the various namespaces in our program:



To describe relationships between namespaces, we use the terms *inner*, *outer*, and *adjacent*. Given three namespaces, *x*, *a*, and *b*, where *a* and *b* are members of *x*,

- *x* is an *outer namespace* of *a* and *b* (relative to *a* and *b*, *x* is an outer namespace).
- *a* and *b* are *inner namespaces* of *x* (relative to *x*, *a* and *b* are inner namespaces).
- *a* and *b* are *adjacent namespaces* (relative to each other).

In the above diagram, for example,

- The global namespace is an outer namespace of *userInterface*, *feeCalculator*, and *std*.
- *userInterface*, *feeCalculator*, and *std* are inner namespaces of the global namespace.

- *userInterface*, *feeCalculator*, and *std* are adjacent namespaces.

When referring to a member of an inner or adjacent namespace, we must, by default, use its *fully qualified name* (the name of the member, including its namespace). To do so, the syntax is

```
n::m
```

where *m* is the member, and *n* is the namespace in which *m* resides. The pair of colons (::) is called the *scope resolution operator*.

We begin *main*, for example, by calling *getRetailPrice*, *getSalesTaxRate*, and *getItemWeight* (lines 21-23):

```
double retailPrice = userInterface::getRetailPrice();
double salesTaxRate = userInterface::getSalesTaxRate();
double itemWeight = userInterface::getItemWeight();
```

Relative to *main* (which resides in the global namespace), these three functions (*getRetailPrice*, *getSalesTaxRate*, and *getItemWeight*) reside in an inner namespace (*userInterface*). We must therefore use their fully qualified names (*userInterface::getRetailPrice*, *userInterface::getSalesTaxRate*, *userInterface::getItemWeight*).

Similarly, when calling *totalFees* from within *main* (lines 25-26), we must use the fully qualified name (*feeCalculator::totalFees*). This is because relative to *main* (the global namespace), *totalFees* resides in an inner namespace (*feeCalculator*).

Finally, we print the *totalCost* using *cout* and *endl* (line 28), which, as mentioned earlier, reside in namespace *std*. We don't, however, need to use the fully qualified names (*std::cout*, *std::endl*), due to the statement (line 19)

```
using namespace std;
```

As mentioned in Chapter 1.3, this statement is called a *using directive*. The syntax of a using directive is

```
using namespace n;
```

where *n* is the namespace that we'd like to use. This allows us to omit the *n::* when referring to members of *n*.

The *using* directive in line 19, for example, allows us to omit the *std::* when referring to any member of namespace *std*. Note, however, that this only applies to the body of the surrounding function (*main*, in this case). To avoid writing *std::* inside the other functions (*getRetailPrice*, *getSalesTaxRate*, *getItemWeight*), we must repeat the *using* directive (lines 37, 49, 61).

We can refer to any member of the same namespace without using its fully qualified name. In line 76, for example, we call *salesTax* and *shippingCost*, without using the namespace name (*feeCalculator::*).

This is because we're calling these functions from within *totalFees*, which also resides in *feeCalculator*.

A *using declaration* is similar to a *using* directive, but only applies to a specific member. The syntax of a using declaration is

```
using n::m;
```

where *n* is the namespace name, and *m* is the member that we'd like to exempt. This allows us to refer to *m*, without using its fully qualified name (*n::m*). The *using* declaration

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

for example, allows us to omit the *std::* when referring to *cout* (and only *cout*). For all other members of *std*, we still need to use the fully qualified names, as in

```
int main()
{
  using std::cout;

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

  return 0;
}
```

As another example, below is an alternative version of *main* (lines 17-31). The *using* directive for *userInterface* allows us to omit the *userInterface::* from the calls to *getRetailPrice*, *getSalesTaxRate*, and *getItemWeight*. The *using* declaration for *totalFees* allows us to omit the *feeCalculator::* when calling *totalFees*.

```
int main()
{
  using namespace std;
  using namespace userInterface;
  using feeCalculator::totalFees;

  double retailPrice = getRetailPrice();
  double salesTaxRate = getSalesTaxRate();
  double itemWeight = getItemWeight();

  double totalCost = retailPrice +
    totalFees(retailPrice, salesTaxRate, itemWeight);

  cout << "\nThe total cost is $" << totalCost << endl;

  return 0;
}
```

There may be times when two or more namespaces have a member with the same name, as in

```
void f();

namespace x
{
void f();
};

namespace y
{
void f();
};
```

Here, there are three versions of the function *f*: one in the global namespace, one in namespace *x*, and another in namespace *y*. Recall from the previous chapter that the return type *void* means that *f* doesn't have a return value.

In this case, even if we write *using* directives for *x* and / or *y*, we'll still need to use the fully qualified name for *f*. Otherwise, the compiler won't know which *f* we're referring to:

```
int main()
{
  using namespace x;
  using namespace y;

  ::f();                // Calling the f in the global namespace ("global f")
  x::f();               // Calling the f in namespace x
  y::f();               // Calling the f in namespace y

  return 0;
}
```

As shown above, to explicitly refer to a member *m* of the global namespace, the syntax is

```
::m
```

There is no namespace name because (as mentioned earlier) the global namespace is nameless.

4.3: Lifetime, Visibility, and Scope

Source files and folders

- *scope*

Chapter outline

- *How the location of a variable's declaration determines its lifetime and visibility*

The *lifetime* of a variable *v* describes when *v* is created and destroyed, while the *visibility* of *v* describes which portions of code can directly access *v*. The lifetime and visibility of *v* are determined by where *v* is declared. Given

```
void f(int p);

int main()
{
  int i = 7;
  std::cout << "i is " << i << std::endl;
  f(i);

  return 0;
}

void f(int p)
{
  if (p % 2 == 0)
  {
    std::string s = " is even\n";
    std::cout << p << s;
  }
}
```

for example,

- *i* is created in the first line of *main*, and destroyed at the end of *main*.
- *p* is created at the beginning of the call to *f*, and destroyed at the end of *f*.
- *s* is created in the conditional body, and destroyed at the end of the conditional body.

- *i* is *visible* (directly accessible) to the body of *main*.
- *p* is visible to the body of *f*.
- *s* is visible to the conditional body.

The lifetime and visibility of a variable are sometimes referred to as its *scope*. As we discuss the program in this chapter, we'll examine the scope of each variable. We begin by declaring *pi* (*main.cpp*, line 4), a variable of type *const double*, initialized to π (3.14159). The keyword *const* is short for

constant (non-modifiable). The type *const double* thus represents a non-modifiable value of type *double*. This means that the value of *pi*, once initialized, can't be changed. If we try to assign a new value, as in

```
pi = 1.618;
```

then our program won't compile. This makes the *const* keyword indispensable when declaring mathematical constants and other fixed values.

Because *pi* isn't declared inside of a particular function, *pi* will be created at the very beginning of the program (before the execution of *main*). It will remain visible to every function at all times, and won't be destroyed until the end of the program (after the termination of *main*). These types of variables, which have the broadest possible scope (lifetime and visibility), are called *global variables*.

Next, we declare the function *calculateArea* (line 6), which returns the area of a circle with the given radius *r*. To implement *calculateArea*, we use the formula (πr^2). The expression (line 46)

```
pow(r, 2)
```

returns r^2, by calling the Standard Library function *pow*, which is short for *power*. This function, declared as

```
double pow(double base, int exponent);
```

returns the value of the *base* raised to the power of the *exponent*. In order to use *pow*, we need to include the <*cmath*> header (line 1).

Each time *calculateArea* is called, the parameter *r* (line 44) is created and initialized. It will be visible for the duration of the body (line 46), and destroyed when the function ends.

Our next function, *calculateCircumference* (line 7), returns the circumference of a circle with the given radius *r*. To implement *calculateCircumference*, we use the formula (π * *diameter*). We begin by calculating the *diameter* (line 51), which is equal to twice the radius. We then return the value of (*pi* * *diameter*) (line 53).

Each time *calculateCircumference* is called, the parameter *r* (line 49) is created and initialized. It will be visible for the duration of the body (lines 51-53), and destroyed when the function ends. Similarly, after we declare *diameter* (line 51), it will be visible for the duration of the body (lines 52-53), and destroyed when the function ends.

To implement *main*, we begin by printing the message (lines 13-14)

```
This program calculates the area and circumference of 3 circles,
as well as running totals of the area and circumference.
```

We then declare and initialize two *double*s (lines 16-17),

- *sumAreas*, the running total (sum) of the circles' areas
- *sumCircumferences*, the running total (sum) of the circles' circumferences

Both of these variables will be visible for the duration of the body (lines 18-41), and destroyed at the end of *main*.

Next, we calculate the area and circumference of each circle, and update the running totals while doing so. The loop (lines 19-39) performs 3 iterations, for $c = \{1, 2, 3\}$. The variable *c*, declared in the initializer (line 19), will be only visible to the loop (lines 19-39), and destroyed when the loop ends.

In each iteration of the loop (lines 21-38),

- We declare a variable *radius* (line 21), which is the radius of the current circle. We then prompt the user for the length of the *radius* (lines 23-27). This variable will only be visible for the remainder of the current iteration (lines 22-38), after which it will be destroyed.

- We declare a variable *area* (line 29), which is the area of the current circle. To initialize the value of *area*, we call *calculateArea*, using the *radius*. This variable (*area*) will only be visible for the remainder of the current iteration (lines 30-38), after which it will be destroyed.

- We declare the variable *circumference* (line 30), which is the circumference of the current circle. To initialize the value of *circumference*, we call *calculateCircumference*, using the *radius*. This variable (*circumference*) will only be visible for the remainder of the current iteration (lines 31-38), after which it will be destroyed.

- We update the running total of the area (*sumAreas*) (line 32), by incrementing *sumAreas* by the current *area*.

- We update the running total of the circumference (*sumCircumferences*) (line 33), by incrementing *sumCircumferences* by the current *circumference*.

- We print the *area* and *circumference* of the current circle (lines 35-36), followed by the running totals of the area and circumference (lines 37-38).

Given *radius* values of 7, 4, and 9, our program generates the following output:

```
This program calculates the area and circumference of 3 circles,
as well as running totals of the area and circumference.

Enter the radius of circle 1 (e.g. for 3.5 m, enter 3.5): 7

Area of circle 1 = 153.938 sq m
Circumference of circle 1 = 43.9823 m
Sum of areas = 153.938 sq m
Sum of circumferences = 43.9823 m
```

```
Enter the radius of circle 2 (e.g. for 3.5 m, enter 3.5): 4

Area of circle 2 = 50.2654 sq m
Circumference of circle 2 = 25.1327 m
Sum of areas = 204.203 sq m
Sum of circumferences = 69.115 m

Enter the radius of circle 3 (e.g. for 3.5 m, enter 3.5): 9

Area of circle 3 = 254.469 sq m
Circumference of circle 3 = 56.5486 m
Sum of areas = 458.672 sq m
Sum of circumferences = 125.664 m
```

Before moving on, we'll briefly summarize the rules regarding scope. Given a variable *v*,

- If *v* is declared within the body of a function, loop, or conditional statement, then *v*'s scope will be limited to the *block* (pair of braces) surrounding *v*'s declaration site. If the surrounding braces form a loop body, then *v* will be destroyed at the end of each iteration.

- If *v* is declared in the initializer of a *for* loop, then *v*'s scope will cover the entire loop (all iterations).

- If *v* is a parameter of a function *f*, then *v*'s scope will cover the entire body of *f*.

- If *v* is a global variable, then *v*'s scope will cover the entire program.

Finally, to maximize code readability and reduce the chances of *bugs* (errors / unintended behavior), variables should be declared with the narrowest possible scope. If, for example, we need to write a function *f* that prints the numbers 1 through 5, we should prefer

```
void f()
{
  for (int i = 1; i != 6; ++i)      // Narrowest possible scope (i's scope
    std::cout << i << std::endl;    // is limited to the loop)
}
```

as opposed to

```
void f()
{
  int i = 1;                        // Broader scope (i's scope covers the
                                    // entire body of f)
  while (i != 6)
  {
    std::cout << i << std::endl;
    ++i;
  }
}
```

or worse yet,

```
int i = 1;                              // Broadest possible scope (i's scope
                                        // covers the entire program)
void f()
{
  while (i != 6)
  {
    std::cout << i << std::endl;
    ++i;
  }
}
```

Variables, in other words, should be declared as close as possible to where they are used. As shown above, *i* is merely a loop counter, so we need not expose it to the rest of *f*, or the entire program. Doing so would make it easier to create bugs, as well as reduce efficiency, because *i* would remain in memory longer than necessary.

Exceptions to this rule can be made, however, for frequently-used constants, such as *pi* in our program (*main.cpp*, line 4).

4.4: Function Overloading

Source files and folders

- *functionOverloading*

Chapter outline

- *Creating multiple functions with the same name*

Function overloading is the process of creating an overloaded function. An *overloaded function* is a function for which there are multiple versions with the same name, differing by their number and / or types of parameters.

Each version of an overloaded function *f* is called an *overload* of *f*. Our program (*main.cpp*), for example, begins by declaring three overloads of a function called *area* (lines 4-6):

- The first (line 4) returns the area of a square with the given *length*, by calculating $length^2$ (lines 33-36).

- The second (line 5) returns the area of a rectangle with the given *length* and *width*, by multiplying the *length* by the *width* (lines 38-41).

- The third (line 6) returns the area of a cuboid (rectangular prism) with the given *length*, *width*, and *height*, by summing the area of all 6 sides (lines 43-48).

We begin *main* by prompting the user for the *length*, *width*, and *height* (lines 12-23). We then call each overload of *area* (lines 25-28), printing the result. Based on the number and types of arguments that we provide when calling *area*, the compiler selects the appropriate overload. This process is called *overload resolution*.

Given the values 4, 6, and 3.5, our program generates the following output:

```
Enter length (e.g. for 2.5 m, enter 2.5): 4
Enter width (e.g. for 2.5 m, enter 2.5): 6
Enter height (e.g. for 2.5 m, enter 2.5): 3.5

The area of a square is 16 sq m
The area of a rectangle is 24 sq m
The area of a cuboid is 118 sq m
```

The *pow* function (line 35) is an example of overloading by parameter type. The Standard Library provides several overloads of pow, including

```
double pow(double base, int exponent);
double pow(double base, double exponent);
```

Given

```
double x = 1.618;
int y = 3;
double z = 4.5;
```

for example, suppose that we make the function call

```
pow(x, y)
```

In this case, the arguments *x* and *y* are of type *double* and *int*, so the compiler would select the first overload, *pow(double base, int exponent)*. Similarly, if we make the function call

```
pow(x, z)
```

the arguments *x* and *z* are both of type *double*, so the compiler would select the second overload, *pow(double base, double exponent)*.

Overloads can also differ by return type, as long as their number and / or types of parameters are different. We could, for example, overload a function *f* as

```
int f(double d);
double f(int i);
bool f(double d, int i);
```

This is legal because every overload of *f* has a different number of parameters, and / or different types of parameters.

We cannot, however, overload a function by return type *only*. The following, for example, won't compile:

```
int g(bool b, char c);
double g(bool b, char c);
bool g(int i);
```

This is illegal because the first two overloads of *g* differ only by their return type. Given the expression

```
g(true, 'x')
```

for example, the compiler won't be able to tell whether we intend to call the first or second overload of *g*.

4.5: Header Files and Inline Functions

Source files and folders

- *headerFiles*

Chapter outline

- *Splitting a program among multiple source files*

In Chapter 1.1, we outlined the process by which source code is transformed into an executable:

```
Source Code (.h and .cpp files, written in C++)
     |
     |      (Compilation)
     v
Object Code (.obj files, written in machine language)
     |
     |      (Linking)
     v
Executable  (.exe file, finished program)
```

Source files with the extension *.h* are called *header files*, commonly referred to as *headers*. Standard Library headers, such as <*iostream*> and <*cmath*>, don't have the *.h* extension, in order to distinguish them from non-standard headers.

As we discussed in Chapter 1.1, *compilation* is the process of translating source code (*.h* and *.cpp* files) into object code (*.obj* files). A *translation unit* consists of the source code in a single *.cpp* file, and all of its included headers.

All of our programs thus far have consisted of a single translation unit (*main.cpp*, and one or more Standard Library headers). In this chapter, we'll take the source code from Chapter 4.1 and divide it into multiple translation units.

We begin by creating the header file *shippingCost.h*, which contains the declaration of the *shippingCost* function (line 1). We then create the source file *shippingCost.cpp*, containing the definition (lines 1-11).

Next, we create the header file *salesTax.h*, which contains the declaration of the *salesTax* function (line 4), as well as the definition (lines 6-9). The *inline* keyword (line 6), placed before the return type in the definition, designates *salesTax* as an *inline function*. This instructs the compiler to insert the entire body of *salesTax* (lines 7-9) at each call site, thereby eliminating the overhead of a function call.

Suppose, for example, that we call *salesTax* from within *main*, as in

// Continued on next page

```
int main()
{
  double p = 19.99;
  double r = 7.50;
  double s = salesTax(p, r);    // Call site of salesTax

  return 0;
}
```

Normally, this would entail a function call to *salesTax*. Because *salesTax* is an inline function, however, the compiler inserts the body of *salesTax* directly at the call site, generating the code

```
int main()
{
  double p = 19.99;
  double r = 7.50;
  double s = p * (r / 100);

  return 0;
}
```

This eliminates the call to *salesTax* altogether, thereby improving the performance of *main*. This process, by which the compiler inserts the body of an inline function directly at the call site, is called *inline expansion*.

For inline expansion to occur, the function definition must appear before the call site. This is why *salesTax.h* contains both the declaration and the definition. The definition, however, can only appear once within a given translation unit.

Suppose, for example, that there are two header files, *checkout.h* and *invoice.h*, both of which contain calls to *salesTax*. Both of these files must therefore include *salesTax.h*:

```
checkout.h                  invoice.h
{                           {
  #include "salesTax.h"       #include "salesTax.h"
}                           }
```

If *main.cpp* then includes *checkout.h* and *invoice.h*, as in

```
main.cpp
{
  #include "checkout.h"
  #include "invoice.h"
}
```

then the function definition for *salesTax* will appear twice, preventing compilation.

The solution to this is to create an include guard for *salesTax.h*. An *include guard* is a set of preprocessor commands that prevents a body of text from appearing more than once in the same translation unit. The syntax of an include guard is

```
#ifndef HEADER_ID
#define HEADER_ID

// Source code

#endif
```

where *HEADER_ID* is a unique identifier for the header file, customarily in all-caps. All of the text between *#ifndef* and *#endif* will appear no more than once in a given translation unit.

In *salesTax.h*, lines 1, 2, and 11 form the include guard. This ensures that lines 3-10 will appear only once in a given translation unit, even if *salesTax.h* is included multiple times.

We can now include the headers for *salesTax* and *shippingCost* in *main.cpp* (lines 3-4). The function definition for *main* (lines 6-30) is unchanged from Chapter 4.1. Our program now consists of two translation units:

```
Translation Unit 1
{
  <iostream>
  salesTax.h
  shippingCost.h
  main.cpp
}

Translation Unit 2
{
  shippingCost.cpp
}
```

At *compile time* (when the source code is compiled), the compiler works on each translation unit separately. *Unit 1* is compiled into the object file *main.obj*, and *Unit 2* is compiled into the object file *shippingCost.obj*. These two object files are then linked to form the executable (*.exe*) file.

By isolating the function definition of *shippingCost* in its own translation unit, we reduce the impact of making future changes to the source code. To modify the definition of *shippingCost*, for example, we would only need to recompile *Unit 2*, then link the updated *shippingCost.obj* file with the existing (unchanged) *main.obj* file. None of the source code in *Unit 1* would have to be recompiled.

Before moving on, here's a summary of the key points regarding header files and inline functions:

- For non-inline functions, place the declaration in a header (*.h*) file, and the definition in a source (*.cpp*) file. An include guard in the header file isn't required. It will, however, speed up compilation if the header is included multiple times in the same translation unit.

- For inline functions, place the declaration and the definition in a header file. The header file must contain an include guard, to prevent the definition from appearing more than once in the same translation unit.

- Defining a function as inline doesn't guarantee inline expansion at every call site. The *inline* keyword, in other words, is only a suggestion to the compiler, not a strict command. This is because there is a downside to inline expansion, in the form of increased code size. At each call site, the compiler performs a cost-benefit analysis. If inline expansion is deemed to be suboptimal, the compiler will generate a normal function call instead.

- Because inline expansion increases code size, the best functions to *inline* are those with particularly short bodies (around 5 lines or less).

Part 5: Pointers, Arrays, and References

5.1: Pointers

Source files and folders

- *pointers*

Chapter outline

- *Bytes, memory addresses, and hexadecimal numbers*
- *Indirectly accessing a variable, by taking its address*

Memory usage and storage capacity are measured in units called *bytes* (*B*). You've likely encountered the terms *kilobyte* (*kB*), *megabyte* (*MB*), *gigabyte* (*GB*), and *terabyte* (*TB*), all of which represent multiples of 1 byte:

```
1 kB = 1000 B                          (1 kilobyte = 1 thousand bytes)
1 MB = 1000 kB = 1,000,000 B           (1 megabyte = 1 million bytes)
1 GB = 1000 MB = 1,000,000,000 B       (1 gigabyte = 1 billion bytes)
1 TB = 1000 GB = 1,000,000,000,000 B (1 terabyte = 1 trillion bytes)
```

On a *RAM* (random access memory) chip, each byte has a numerical address, indicating its physical location. Memory addresses are commonly represented using hexadecimal numbers.

For the purposes of this book, you won't need to know how to read or write hexadecimal values. We will, however, briefly describe the process, if only to provide some basic understanding and context.

The *decimal*, or *base-10* numbering system, uses 10 digits. This is the system that we use most often in daily life. To compute the value of a base-10 number, we multiply each digit by the corresponding power of 10, and take the sum:

```
Digit  0   1   2   3   4   5   6   7   8   9
Value  0   1   2   3   4   5   6   7   8   9

7    = (7 x 10^0)
29   = (2 x 10^1) + (9 x 10^0)
943  = (9 x 10^2) + (4 x 10^1) + (3 x 10^0)
9484 = (9 x 10^3) + (4 x 10^2) + (8 x 10^1) + (4 x 10^0)
```

The *hexadecimal*, or *base-16* system, uses 16 digits. To compute the value of a base-16 number, we multiply each digit by the corresponding power of 16, and take the sum. The base-16 numbers {*7*, *1D*, *3AF*, *250C*}, for example, are equivalent to the base-10 numbers {*7*, *29*, *943*, *9484*}:

// Continued on next page

```
Digit  0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
Value  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15

7    = (7 x 16^0)                                          =  7
1D   = (1 x 16^1) + (D x 16^0)                             =  29
3AF  = (3 x 16^2) + (A x 16^1) + (F x 16^0)                =  943
250C = (2 x 16^3) + (5 x 16^2) + (0 x 16^1) + (C x 16^0) =  9484
```

At *runtime* (when a program is run), variables are stored in memory. To create a variable, the system *allocates* (reserves) the required number of bytes, at a particular address. The number of bytes required to store a variable depends on its type (*int*, *double*, etc.), the hardware platform, and the compiler.

The size of an *int*, for example, may be 4 bytes on one system, and 8 bytes on another. Note, however, that on the *same* system, all *int*s will be the same *size*, regardless of *value*. If, for example, the size of an *int* on a given system is 4 bytes, then every *int* on that system will use 4 bytes of memory, no more and no less. An *int k*, whose value is 7, and an *int p*, whose value is 7,000,000, will both use 4 bytes.

Although memory addresses are usually represented using hexadecimal numbers, we'll use base-10 for the sake of simplicity. Suppose, for example, that our program has access to 30 bytes of memory, with the addresses {*01*, *02*, *03...30*}:

```
   01    02    03    04    05    06    07    08    09    10
|     |     |     |     |     |     |     |     |     |     |

   11    12    13    14    15    16    17    18    19    20
|     |     |     |     |     |     |     |     |     |     |

   21    22    23    24    25    26    27    28    29    30
|     |     |     |     |     |     |     |     |     |     |
```

Let's also suppose that on this system, the size of an *int* is 2 bytes. Our program (*main.cpp*) begins by creating two *int*s, *x* and *y* (lines 7-8), initialized to 0 and 1 respectively. At runtime, the system allocates 2 bytes for *x*, and 2 bytes for *y*. The following diagram depicts a hypothetical scenario, in which *x* resides at address *03* (with a value of 0), and *y* resides at address *17* (with a value of 1):

```
   01    02    03    04    05    06    07    08    09    10
|     |     |x(0)---------|     |     |     |     |     |     |

   11    12    13    14    15    16    17    18    19    20
|     |     |     |     |     |     |y(1)---------|     |     |

   21    22    23    24    25    26    27    28    29    30
|     |     |     |     |     |     |     |     |     |     |
```

A *pointer* is a variable whose value is a memory address. In line 10, for example,

```
int* p = nullptr;
```

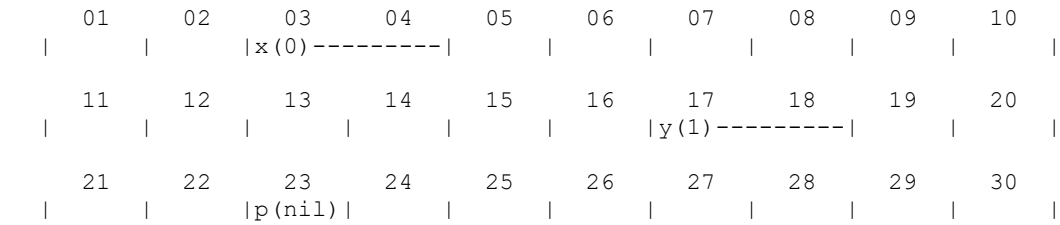
we create a pointer *p*, of type *int**, with an initial value of *nullptr*. The asterisk (*) means "pointer to," and the type *int** is read right-to-left, as "pointer to *int*." The type *int** thus represents the address of an *int*.

The keyword *nullptr* is short for "null pointer," a special value that means "no address." Although *ptr* is short for "pointer," some people pronounce it as "putter" (rather than "PTR") for the sake of brevity.

At this point, there are a few different ways of describing *p*, all of which mean the same thing:

- "*p* is a null pointer" (A *null pointer* is a pointer with a value of no address)
- "*p* is null"
- "*p* is nil" (*nil*, which rhymes with "Bill," is a synonym for *null*)

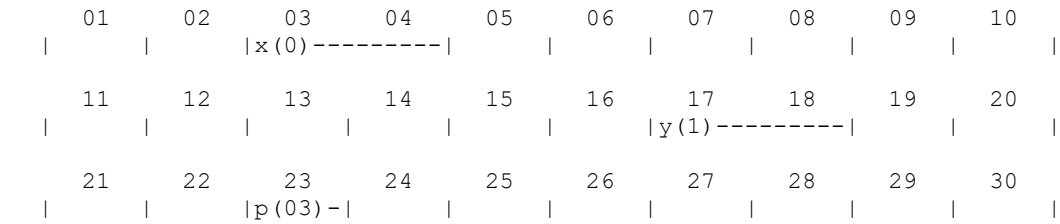
Suppose that on this system, the size of an *int** is 1 byte, and that *p* resides at address *23*. The runtime memory layout thus becomes



After printing the current values of *p*, *x*, and *y* (lines 12-14), we modify the value to *p* (line 16), via the statement

```
p = &x;
```

The ampersand (&) is the *address-of operator*, which returns the address of its operand. Given the above diagram, the expression (*&x*) returns *03*, the address of *x*. The entire statement (line 16) thus sets the value of *p* to address *03*:



To describe *p*, we can now say "*p* points to *x*," or "*p* refers to *x*," because the value of *p* is the address of *x*.

The *referent object* (or *referent*) of a pointer is the object to which the pointer refers. We can therefore describe *x* by saying "*x* is the referent object of *p*," or "*x* is the referent of *p*."

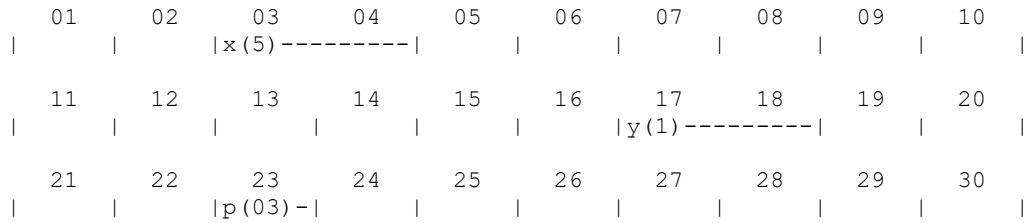
Because *p* points *x*, we can indirectly access *x* via *p*. The statement (line 17)

```
*p = 5;
```

for example, is equivalent to

```
x = 5;
```

The asterisk (*) is called the *indirection operator* or *dereference operator*, which returns the referent of its operand. In this case, the expression (**p*) returns *x*, the referent of *p*. The entire statement (line 17) thus indirectly sets the value of *x* to 5:

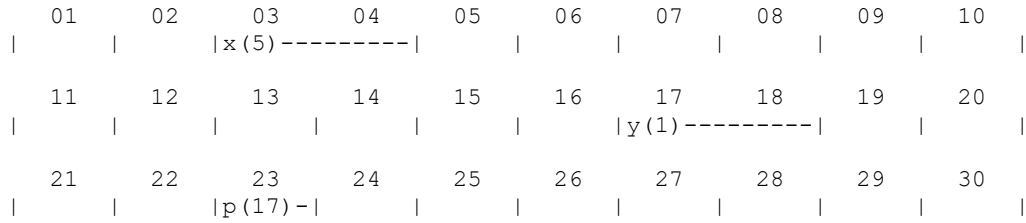


The process of accessing a pointer's referent object is called *dereferencing*. We can thus read the expression (**p*) as "dereference *p*," or more commonly, "the referent of *p*." We can therefore read the statement (**p = 5;*) as "Set the referent of *p* to 5."

In lines 19-22, we print *p* (the address of *x*), **p* (the value of *p*'s referent *x*), *x*, and *y*. We then modify the value of *p* once again, via the statement (line 24)

```
p = &y;
```

which sets *p* to the address of *y*:

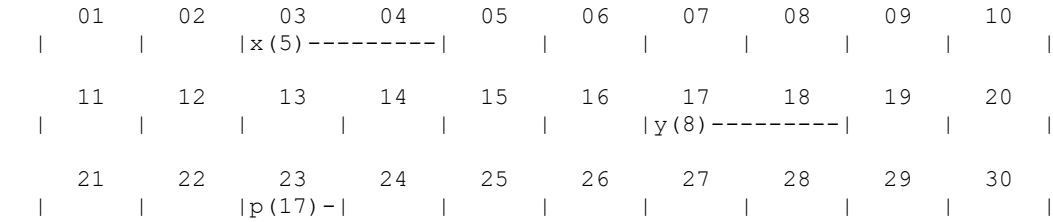


The *referent address* of a pointer is the address to which the pointer refers. In the above diagram, for example, *p*'s referent address is *17*.

The process of changing a pointer's referent address is called *reseating*. The statement in line 24 thus reseats *p* from *03* (the previous referent address) to *17* (the new referent address). Now that *p* points to *y*, the expression (**p*) returns *y*. The statement (line 25)

```
*p = 8;
```

therefore indirectly sets the value of *y* to 8:



Finally, we print *p*, **p*, *x*, and *y* once more (lines 27-30).

Using the above diagram, here's a brief summary of the notation introduced in this chapter:

- The value of *x* is 5
- The value of *y* is 8
- The value of *p* is address *17* (*p*'s type is *int** (pointer to *int*))

- *&x* (the address of *x*) is *03*
- *&y* (the address of *y*) is *17*
- *&p* (the address of *p*) is *23*

- **p* (the referent of *p*) is *y* (the *int* at address *17*)

On my machine, the program generates the following output. Memory addresses are represented using 8-digit hexadecimal values:

```
p = nullptr = 00000000
x = 0
y = 1

p = &x = 0031F7DC
*p = 5
x = 5
y = 1

p = &y = 0031F7E0
*p = 8
x = 5
y = 8
```

Before moving on, there are a few additional points regarding pointers.

Just as *int** means "pointer to *int*," the same pattern applies to other types, such as *double** (pointer to *double*), *std::string** (pointer to *std::string*), etc. When setting the value of a pointer, however, the referent type (*int*, *double*, etc.) must match the pointer type (*int**, *double**, etc.):

// Continued on next page

```
int i = 9;
double d = 3.14;

int* pi = &i;        // Ok: Set pi (pointer to an int) to
                     // &i (the address of an int)

double* pd = &d;     // Ok: Set pd (pointer to a double) to
                     // &d (the address of a double)

pi = &d;             // Compiler error: Can't set pi (pointer to an int) to
                     // &d (the address of a double)

pd = &i;             // Compiler error: Can't set pd (pointer to a double) to
                     // &i (the address of an int)
```

Additionally, dereferencing a pointer without a valid referent will result in *undefined behavior* (unpredictable consequences at runtime, such as a program crash or invalid output):

```
double* q = nullptr;
string* r;

*q = 1.618;              // Undefined behavior: Dereferencing a null pointer

cout << *r << endl;      // Undefined behavior: Dereferencing a pointer without
                         // a valid referent
```

5.2: Pass by Reference

Source files and folders

- *passByReference*
- *swapInts*

Chapter outline

- *Pointers as function parameters*
- *Pass-by-reference vs. pass-by-value*

Now that we understand the basics of using pointers, we'll demonstrate how to use them as function parameters. To do so, we begin by declaring the function (*swap.h*, line 3)

```
void swap(int* a, int* b);
```

which has two parameters, *a* and *b*, of type *int** (pointer to *int*). As discussed previously, the return type *void* indicates that the function doesn't return a value. We've declared the function in namespace *dss* (lines 1-2, 4), which stands for the title of this book (*Data Structures from Scratch*).

The *swap* function exchanges the value of *a*'s referent with that of *b*'s referent. Suppose, for example, that *a* points to an *int x*, whose value is 17, and *b* points to an *int y*, whose value is 43:

```
a -> x(17)     // The arrow symbol (->) means "points to"
b -> y(43)
```

The function call *swap(a, b)* exchanges the values of *x* and *y*, at which point *x* becomes 43 and *y* becomes 17:

```
a -> x(43)
b -> y(17)
```

To implement *swap*, we begin by creating a temporary *int c* (*swap.cpp*, line 5), initialized to the value of *a*'s referent. This variable *c* saves the original value of *a*'s referent, prior to the exchange. We then set *a*'s referent to the value of *b*'s referent (line 6). Finally, we set *b*'s referent to *c*, which contains the original value of *a*'s referent (line 7).

In the above example, lines 5-7 perform the following operations:

```
int c = x;      // c = 17;
x = y;          // x = 43;
y = c;          // y = 17;
```

We begin *main* by obtaining the values of *x* and *y* from the user (*main.cpp*, lines 9-16). In each iteration of the loop (lines 18-34),

- We print the current values of *x* and *y* (lines 20-21).
- We prompt the user for a single character, which we store in the variable *userCommand* (lines 23-28). A lowercase *s* represents a command to swap the values of *x* and *y*, and a lowercase *q* represents a command to quit.
- If the user entered *s* (line 30), we call *swap* (line 31), passing the addresses of *x* and *y*. We then begin the next iteration.
- If the user entered a character other than *s* (line 32), we terminate the loop (line 33) and exit *main* (line 36).

Using the values 17 and 43, our program generates the following output:

```
Enter an integer value for x: 17
Enter an integer value for y: 43

The value of x is now 17
The value of y is now 43

Enter s to swap the values of x and y, or q to quit (case-sensitive): s

The value of x is now 43
The value of y is now 17

Enter s to swap the values of x and y, or q to quit (case-sensitive): s

The value of x is now 17
The value of y is now 43

Enter s to swap the values of x and y, or q to quit (case-sensitive): q
```

Passing the address of a variable, as in (*passByReference/main.cpp*, line 31)

```
dss::swap(&x, &y);
```

is called *pass-by-reference*. In this case, we're passing the addresses of *x* and *y* to *swap*, so *x* and *y* are said to be “passed by reference.”

Conversely, passing the value of a variable, as in (*functions/main.cpp*, line 25)

```
shippingCost(itemWeight)
```

is called *pass-by-value*. In this case, we're passing the value of *itemWeight* to *shippingCost*, so *itemWeight* is said to be “passed by value.”

5.3: Arrays and Bubble Sort

Source files and folders

- *arraySubscript*

Additional .cpp files (must be compiled and linked with main.cpp)

- *swapInts/swap.cpp*

Chapter outline

- *Declaring arrays*
- *Accessing individual elements*
- *Arrays as function parameters*
- *Implementing the bubble sort algorithm*

An *object* is a specific instance of a type. Given

```
int i = 7;
double d = 3.14;
string s = "Ayrshire";
```

for example, *i* is an object of type *int*, *d* is an object of type *double*, and *s* is an object of type *string*.

An *array* is a set of objects of a single type, stored as a contiguous sequence in memory. Arrays are the most primitive type of data structure. Each object in an array is called an *element*. More broadly speaking, however, the term *element* applies to any object stored in any type of data structure, not just an array.

The following diagram depicts an array *x*, of 3 *int*s, at address *15*. In this example, the size of an *int* is 2 bytes:

```
    11     12     13     14     15     16     17     18     19     20
|      |      |      |      |x[0](4)------|x[1](8)------|x[2](3)------|

The first element,  x[0], is at address 15, and has a value of 4
The second element, x[1], is at address 17, and has a value of 8
The third element,  x[2], is at address 19, and has a value of 3
```

As shown above, the syntax of accessing a particular element is

```
array_name[index]
```

The pair of brackets ([ ]) is called the *array subscript operator*, and the *index* is the position of the desired element. The index of the first element is always 0, so to access the n^{th} element, we use an

index of (*n – 1*). Another way to think of this is that the index represents the *offset*, or number of elements away, from the first. In the above diagram, for example,

- The first element (*x[0]*) is 0 elements away from the first, so its index is 0.
- The second element (*x[1]*) is 1 element away from the first, so its index is 1.
- The third element (*x[2]*) is 2 elements away from the first, so its index is 2.

The elements in an array are functionally identical to the variables that we've been working with all along:

```
cout << x[0] << endl;     // Print the value of element x[0]
x[0] = 7;                 // Set the value of element x[0] to 7
++x[1];                   // Increment the value of element x[1]
int* p = &x[2];           // p points to element x[2] (address 19, above)
*p = 6;                   // Set the value of element x[2] to 6
```

The syntax of declaring an array is

```
element_type array_name[size];
```

where *size* is the total number of elements. The *size* must be a *compile-time constant* (a fixed value, known at compile-time). This type of array is called a *static array* or *fixed-size array*, because the size cannot be set (or modified) at runtime:

```
int x[3];       // Declares an array x, of 3 ints
double y[1];    // Declares an array y, of 1 double
string z[5];    // Declares an array z, of 5 strings
```

Our program in this chapter prompts the user to enter 5 integers, stores them in an array, and sorts the values in ascending order. We begin by declaring a compile-time constant *totalNumbers* (*main.cpp*, line 15), with a value of 5. We then use this value to declare *numbers*, an array of 5 *int*s (line 16). The first element is *numbers[0]*, and the fifth element is *numbers[4]*.

The loop (lines 18-22) performs 5 iterations, for *i* = {*0*, *1*, *2*, *3*, *4*}. In each iteration, we prompt the user for an integer value, and store it in the element at index *i* (*numbers[i]*) (lines 20-21).

We then print the message *Performing bubble sort...* (line 24), and call the function (lines 25, 7)

```
void printArray(int a[], int size);
```

which prints each element in the given array *a*, of the given *size* (total number of elements). As shown above, the syntax of declaring an array as a function parameter is

```
element_type array_name[]
```

We must include the size as a separate parameter, otherwise we won't know how many elements to print.

To implement *printArray*, we perform a total of *size* iterations, for $i = 0$, to $i = (size - 1)$ (line 52). If, for example, *size* is 5, then we perform 5 iterations, for $i = \{0, 1, 2, 3, 4\}$. In each iteration, we print the value of the element at index *i* (*a[i]*), followed by a whitespace (line 53). After printing the final element, we insert a new line before returning (line 55).

After printing the array (line 25), we run the *bubble sort algorithm* (lines 27-38), which rearranges the values from least to greatest. The term *bubble* describes how the algorithm works: the largest value gradually rises to the top, like a bubble in a glass of water. The next largest value then gradually rises to its proper place, and the process repeats until the entire sequence is sorted.

Before demonstrating the algorithm, we need to introduce *sequence notation* (the notation used to describe a sequence of elements). In sequence notation, a sequence is defined by two bounding elements *A* and *B*, separated by a comma. It also contains two symbols, one before *A* and one after *B*.

- A left bracket before *A* indicates that the sequence includes *A*.
- A left parenthesis before *A* indicates that the sequence doesn't include *A*.

- A right bracket after *B* indicates that the sequence includes *B*.
- A right parenthesis after *B* indicates that the sequence doesn't include *B*.

The sequence

```
[numbers[0], numbers[4])
```

for example, contains every element from *numbers[0]* to *numbers[4]*, including *numbers[0]*, but not *numbers[4]*. The sequence thus contains {*numbers[0]*, *numbers[1]*, *numbers[2]*, *numbers[3]*} (elements 0 to 3). Here are a few more examples:

```
[numbers[1], numbers[3]] = {numbers[1], numbers[2], numbers[3]}
[numbers[1], numbers[3]) = {numbers[1], numbers[2]}
(numbers[1], numbers[3]] = {numbers[2], numbers[3]}
(numbers[1], numbers[3]) = {numbers[2]}
```

Returning to the bubble sort algorithm, suppose that *numbers* contains the values {3, 1, 5, 4, 2}:

```
index | 0 | 1 | 2 | 3 | 4 |
value | 3 | 1 | 5 | 4 | 2 |
```

The algorithm works by examining the following sequences:

```
[numbers[0], numbers[4]) (elements 0 to 3)    // Pass 1
[numbers[0], numbers[3]) (elements 0 to 2)    // Pass 2
[numbers[0], numbers[2]) (elements 0 to 1)    // Pass 3
[numbers[0], numbers[1]) (element 0)          // Pass 4
```

In each pass, we *traverse* (check each element in) the sequence, where *i* is the index value of the current element, and $(i + 1)$ is the index value of the next element. If the current element (*numbers[i]*)

is greater than the next element (*numbers[i + 1]*), we swap their values. In the following outline, *end* is the index value of the final element in the sequence.

In Pass 1, for example, we traverse the sequence [*numbers[0]*, *numbers[4]*). The final element in the sequence is *numbers[3]*, so *end* is 3. There are 4 elements in the sequence, so we perform 4 iterations, for *i* = {*0*, *1*, *2*, *3*}:

```
Pass 1: Traverse [numbers[0], numbers[4]) (elements 0 to 3)    // end = 3
  If numbers[0] > numbers[1], then swap them                   // i = 0
  If numbers[1] > numbers[2], then swap them                   // i = 1
  If numbers[2] > numbers[3], then swap them                   // i = 2
  If numbers[3] > numbers[4], then swap them                   // i = 3
```

At the end of Pass 1, *numbers[4]* contains the largest value.

```
Pass 2: Traverse [numbers[0], numbers[3]) (elements 0 to 2)    // end = 2
  If numbers[0] > numbers[1], then swap them                   // i = 0
  If numbers[1] > numbers[2], then swap them                   // i = 1
  If numbers[2] > numbers[3], then swap them                   // i = 2
```

At the end of Pass 2, *numbers[3]* contains the second-largest value.

```
Pass 3: Traverse [numbers[0], numbers[2]) (elements 0 to 1)    // end = 1
  If numbers[0] > numbers[1], then swap them                   // i = 0
  If numbers[1] > numbers[2], then swap them                   // i = 1
```

At the end of Pass 3, *numbers[2]* contains the third-largest value.

```
Pass 4: Traverse [numbers[0], numbers[1]) (element 0)          // end = 0
  If numbers[0] > numbers[1], then swap them                   // i = 0
```

At the end of Pass 4, *numbers[1]* contains the fourth-largest value. Because there are 5 elements in the array, *numbers[0]* is left with the fifth-largest (i.e. the smallest) value.

Each iteration of the outer loop (line 27) represents a single pass, where *end* is the index value of the final element in the current sequence. The loop performs (*totalNumbers – 1*) iterations, for *end* = (*totalNumbers – 2*), down to 0. If, for example, *totalNumbers* is 5, the loop performs 4 iterations, for *end* = {*3*, *2*, *1*, *0*} (as shown above).

The inner loop (lines 29-37) traverses the current sequence, [*numbers[0]*, *numbers[end]*], where *i* is the index value of the current element. In each iteration, *numbers[i]* is the current element, and *numbers[i + 1]* is the next element.

- We begin by printing the message *If (current element) > (next element), then swap* (line 31).
- If the current element is greater than the next (line 33), we swap their values (line 34).
- Finally, we print the entire array (line 36), displaying the result.

Upon completion of the outer loop, we print the fully sorted array (lines 40-41).

Given the sequence {3, 1, 5, 4, 2}, for example, the algorithm performs the following operations:

```
Iteration 1 (end = 3)                                  // Pass 1
{
  Iteration 1.1 (i = 0)
  {
            i          end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 3 | 1 | 5 | 4 | 2 |

    numbers[0] is greater than numbers[1]
      swap(&numbers[0], &numbers[1]);                  // swap 3 and 1

            i          end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 5 | 4 | 2 |
  }

  Iteration 1.2 (i = 1)
  {
              i      end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 5 | 4 | 2 |

    numbers[1] is not greater than numbers[2];
  }

  Iteration 1.3 (i = 2)
  {
                    i  end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 5 | 4 | 2 |

    numbers[2] is greater than numbers[3]
      swap(&numbers[2], &numbers[3]);                  // swap 5 and 4

                    i  end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 4 | 5 | 2 |
  }

  Iteration 1.4 (i = 3)
  {
                        i
                      end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 4 | 5 | 2 |

    numbers[3] is greater than numbers[4]
      swap(&numbers[3], &numbers[4]);                  // swap 5 and 2

    // Continued on next page
```

```
                          i
                        end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 4 | 2 | 5 |
  }
}

// The largest value (5) is now at the correct position
```

```
Iteration 2 (end = 2)                              // Pass 2
{
  Iteration 2.1 (i = 0)
  {
            i      end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 4 | 2 | 5 |

    numbers[0] is not greater than numbers[1];
  }

  Iteration 2.2 (i = 1)
  {
              i  end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 4 | 2 | 5 |

    numbers[1] is not greater than numbers[2];
  }

  Iteration 2.3 (i = 2)
  {
                  i
                end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 4 | 2 | 5 |

    numbers[2] is greater than numbers[3]
      swap(&numbers[2], &numbers[3]);              // swap 4 and 2

                  i
                end
    index | 0 | 1 | 2 | 3 | 4 |
    value | 1 | 3 | 2 | 4 | 5 |
  }
}

//  The second-largest value (4) is now at the correct position
```

// Continued on next page

```
  Iteration 3 (end = 1)                                  // Pass 3
  {
    Iteration 3.1 (i = 0)
    {
              i  end
      index | 0 | 1 | 2 | 3 | 4 |
      value | 1 | 3 | 2 | 4 | 5 |

      numbers[0] is not greater than numbers[1];
    }

    Iteration 3.2 (i = 1)
    {
                 i
                end
      index | 0 | 1 | 2 | 3 | 4 |
      value | 1 | 3 | 2 | 4 | 5 |

      numbers[1] is greater than numbers[2]
        swap(&numbers[1], &numbers[2]);                  // swap 3 and 2

                 i
                end
      index | 0 | 1 | 2 | 3 | 4 |
      value | 1 | 2 | 3 | 4 | 5 |
    }
  }

  // The third-largest value (3) is now at the correct position
```

```
  Iteration 4 (end = 0)                                  // Pass 4
  {
    Iteration 4.1 (i = 0)
    {
              i
             end
      index | 0 | 1 | 2 | 3 | 4 |
      value | 1 | 2 | 3 | 4 | 5 |

      numbers[0] is not greater than numbers[1];
    }
  }

  // The fourth-largest value (2) is now at the correct position

  // There are 5 values altogether, and the 4 largest ones (5, 4, 3, 2) are at
  // the correct positions

  // The only remaining value (1, the fifth-largest) is therefore also at the
  // correct position
```

Given the sequence {3, 1, 5, 4, 2}, our program generates the following output:

```
Enter a number (integer): 3
Enter a number (integer): 1
Enter a number (integer): 5
Enter a number (integer): 4
Enter a number (integer): 2

Performing bubble sort...
3 1 5 4 2
If 3 > 1, then swap
1 3 5 4 2
If 3 > 5, then swap
1 3 5 4 2
If 5 > 4, then swap
1 3 4 5 2
If 5 > 2, then swap
1 3 4 2 5
If 1 > 3, then swap
1 3 4 2 5
If 3 > 4, then swap
1 3 4 2 5
If 4 > 2, then swap
1 3 2 4 5
If 1 > 3, then swap
1 3 2 4 5
If 3 > 2, then swap
1 2 3 4 5
If 1 > 2, then swap
1 2 3 4 5

The sorted sequence is:
1 2 3 4 5
```

Before moving on, there's an additional point to note regarding the use of arrays as function parameters. When passing an array to a function, as in (*main.cpp*, lines 25, 36)

```
printArray(numbers, totalNumbers);
```

the argument (*numbers*) is automatically passed by reference. Within the body of *printArray* (lines 50-55), the parameter *a* is *not* a copy of *numbers*; rather, the name *a* refers to the exact same array as *numbers*, just by a different name. The reason for this will become clear in the next chapter, when we discuss the relationship between pointers and arrays. For now, here's a summary of the key points we've covered so far:

- An array is a set of objects, of a single type, organized as a contiguous sequence in memory.
- To access a particular element, we use the array subscript operator.
- The first element has an index of 0, so to access the n^{th} element, we use an index of $(n - 1)$.
- The size of an array must be a compile-time constant; we cannot set or change it at runtime.
- A function with an array parameter must also include a separate parameter for the size.
- When passing an array to a function, it is automatically passed by reference; the array parameter is not a separate copy of the array passed by the caller.